Dissertation

Zuverlässige modusbasierte Kommunikation und Virtual Prototyping in der Entwicklung verteilter Echtzeitsysteme

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation
von

TOBIAS BRAUN

Abgabedatum	:	08.07.2015
Wissenschaftliche Aussprache	:	22.01.2016
Dekan	:	Prof. Dr. rer. nat. Klaus Schneider
Promotionskommission		
Vorsitzender	:	Prof. Dr.-Ing. Karsten Berns
Berichterstatter	:	Prof. Dr.-Ing. Reinhard Gotzhein
	:	Prof. Dr.-Ing. Jens B. Schmitt

D 386

Danksagung

An dieser Stelle möchte ich die Gelegenheit ergreifen, mich bei allen zu bedanken, die mich direkt oder indirekt sowohl bei meinem Studium als auch bei der Arbeit an der Dissertation unterstützten. Zuerst möchte ich mich bei den Mitgliedern der Promotionskomission – Prof. Reinhard Gotzhein, Prof. Jens Schmitt und Prof. Karsten Berns – für ihr Engagement und die von ihnen investierte Zeit bedanken.

Hierbei gilt mein besonderer Dank meinem Betreuer Prof. Reinhard Gotzhein, der mir nach dem Diplom, die Promotion in seiner Arbeitsgruppe *Vernetzte Systeme* ermöglichte. Besonders bedanke ich mich für die zahlreichen fachlichen sowie persönlichen Diskussionen und Anregungen, welche maßgeblich zum erfolgreichen Abschluss dieser Arbeit beigetragen haben. Ohne die von ihm eingeräumten forscherischen Freiheiten und das entgegengebrachte Vertrauen, wäre die Arbeit in dieser Form nicht zustande gekommen. Prof. Jens Schmitt danke ich, dass er sich als Gutachter zur Verfügung gestellt hat, auch wenn die Arbeit etwas umfangreicher ausgefallen ist, als zunächst geplant.

Ich bedanke mich ebenfalls bei meinen ehemaligen und gegenwärtigen Kollegen Philipp Becker, Dennis Christmann, Markus Engel, Christopher Kramer, Marc Krämer, Anuschka Igel und Thomas Kuhn für die anregenden Diskussionen sowie die gute Zusammenarbeit in den gemeinsamen Projekten und Publikationen. Insbesondere danke ich Marc für die intensive Einarbeitung, als ich neu in die Arbeitsgruppe kam, und die eingebrachten Kenntnisse bei der Entwicklung experimenteller Schaltungen und Platinen. Anuschka, Dennis und Thomas danke ich für die gute Zusammenarbeit bei der Entwicklung und Erweiterung von FERAL sowie den in diesem Kontext entstandenen gemeinsamen Veröffentlichungen. Ein weiterer Dank geht an Markus, der die verwendete *Imote 2* Plattform wie kein anderer kennt und mit seinen Ratschlägen sowie Interpretationen von GCC-Fehlermeldungen stets hilfreiche Hinweise bei der Fehlersuche lieferte. Ein besonderer Dank geht auch an Dr. Joachim Thees für viele interessante Diskussionen während der Promotionsphase und seine wertvollen Hinweise und Korrekturvorschläge in der Endphase.

Ich möchte meinen Eltern dafür danken, dass sie mir das Studium der Informatik ermöglichten und mich sowohl während des Studiums als auch bei der Promotion stets mit Rat und Tat unterstützten. Ebenfalls bedanke ich mich bei meinem Bruder Manuel für seine Hilfe und das Durchlesen dieser Arbeit sowie die hiermit verbundenen Hinweise. Abschließend möchte ich mich noch bei allen meinem Freunden und Verwandten bedanken, auch insbesondere für ihr Verständnis, wenn die Zeit einmal in Folge einer wichtigen Deadline etwas knapp wurde.

Kaiserslautern, den 18.02.2016 Tobias Braun

Zusammenfassung

In der aktuellen technologischen Entwicklung spielen verteilte eingebettete Echtzeitsysteme eine immer zentralere Rolle und werden zunehmend zum Träger von Innovationen. Durch den hiermit verbundenen steigenden Funktionsumfang der verteilten Echtzeitsysteme und deren zunehmenden Einsatz in sicherheitsrelevanten Anwendungsgebieten stellt die Entwicklung solcher Systeme eine immer größere Herausforderung dar. Hierbei handelt es sich einerseits um Herausforderungen bezogen auf die Kommunikation hinsichtlich Echtzeitfähigkeit und effizienter Bandbreitennutzung, andererseits werden geeignete Methoden benötigt, um den Entwicklungsprozess solcher komplexen Systeme durch Tests und Evaluationen zu unterstützen und zu begleiten. Die hier vorgestellte Arbeit adressiert diese beiden Aspekte und ist entsprechend in zwei Teile untergliedert.

Der erste Teil der Arbeit beschäftigt sich mit der Entwicklung neuer Kommunikationslösungen, um den gestiegenen Kommunikationsanforderungen begegnen zu können. So erfordert die Nutzung verteilter Echtzeitsysteme im Kontext sicherheitsrelevanter Aufgaben den Einsatz zeitgetriggerter Kommunikationssysteme, die in der Lage sind, deterministische Garantien bezüglich der Echtzeitfähigkeit zu gewähren. Diese klassischen auf exklusiven Reservierungen basierenden Ansätze sind jedoch gerade bei (seltenen) sporadischen Nachrichten sehr ineffizient in Bezug auf die Nutzung der Bandbreite. Das in dieser Arbeit verwendete *Mode-Based Scheduling with Fast Mode-Signaling* (*modusbasierte Kommunikation*) ist ein Verfahren zur Verbesserung der Bandbreitennutzung zeitgetriggerter Kommunikation, bei gleichzeitiger Gewährleistung der Echtzeitfähigkeit. Um dies zu ermöglichen, erlaubt *Mode-Based Scheduling* einen kontrollierten, slotbasierten Wettbewerb, welcher durch eine schnelle Modussignalisierung (*Fast Mode-Signaling*) aufgelöst wird. Im Zuge dieser Arbeit werden verschiedene robuste, zuverlässige und vor allem deterministische Realisierungen von *Mode-Based Scheduling with Fast Mode-Signaling* auf Basis existierender drahtgebundener Kommunikationsprotokolle (TTCAN und FlexRay) vorgestellt sowie Konzepte präsentiert, welche eine einfache Integration in weitere Kommunikationstechnologien (wie drahtlose Ad-Hoc-Netze) ermöglichen.

Der zweite Teil der Arbeit konzentriert sich nicht nur auf Kommunikationsaspekte, sondern stellt einen Ansatz vor, den Entwicklungsprozess verteilter eingebetteter Echtzeitsysteme durch kontinuierliche Tests und Evaluationen in allen Entwicklungsphasen zu unterstützen und zu begleiten. Das im Kontext des Innovationszentrums für *Applied Systems Modeling* mitentwickelte und erweiterte FERAL (ein Framework für die Kopplung spezialisierter Simulatoren) bietet eine ideale Ausgangsbasis für das Virtual Prototyping komplexer verteilter eingebetteter Echtzeitsysteme und ermöglicht Tests und Evaluationen der Systeme in einer realistisch simulierten Umgebung. Die entwickelten Simulatoren für aktuelle Kommunikationstechnologien ermöglichen hierbei realistische Simulationen der Interaktionen innerhalb des verteilten Systems. Durch die Unterstützung von Simulationssystemen mit Komponenten auf unterschiedlichen Abstraktionsstufen kann FERAL in allen Entwicklungsphasen eingesetzt werden. Anhand einer Fallstudie wird gezeigt, wie FERAL verwendet werden kann, um ein Simulationssystem zusammen mit den zu realisierenden Komponenten schrittweise zu verfeinern. Auf diese Weise steht während jeder Entwicklungsphase ein ausführbares Simulationssystem für Tests zur Verfügung. Die entwickelten Konzepte und Simulatoren für FERAL ermöglichen es, Designalternativen zu evaluieren und die Wahl einer Kommunikationstechnologie durch die Ergebnisse von Simulationen zu stützen.

Inhaltsverzeichnis

1 Einführung

In der aktuellen technologischen Entwicklung spielen verteilte eingebettete Echtzeitsysteme eine immer zentralere Rolle und werden zunehmend zum Träger von Innovationen. Der hiermit verbundene stetig wachsende Funktionsumfang führt dazu, dass die Komplexität dieser Systeme immer weiter zunimmt. Hinzu kommt, dass verteilte eingebettete Echtzeitsysteme immer häufiger auch sicherheitskritische Funktionen erbringen.

Belege für diesen Trend finden sich im Bereich der Automobilentwicklung in Form zahlreicher elektronischer Assistenz- und Komfortsysteme (ABS, ESP, Brems- und Spurassistenten, Einparkautomatik), deren Funktionen durch verteilte eingebettete Echtzeitsysteme umgesetzt werden. Um die Interaktion zwischen den einzelnen Komponenten dieser (zum Teil sicherheitskritischen) Systeme sowie den Systemen untereinander zu ermöglichen, kommen mehrere Feldbusse basierend auf unterschiedlichen Technologien (CAN, LIN, FlexRay) zum Einsatz, die über Gateways miteinander verbunden sind. Insgesamt muss ein modernes Auto daher heute als Beispiel für ein hochkomplexes verteiltes eingebettetes Echtzeitsystem betrachtet werden. Weitere Anwendungsgebiete komplexer verteilter Systeme finden sich im Bereich der Avionik sowie der Überwachung und Steuerung von Industrieanlagen.

Aufgrund der wachsenden Komplexität verteilter eingebetteter Echtzeitsysteme und deren zunehmenden Einsatz in sicherheitsrelevanten Anwendungsgebieten stellt die Entwicklung solcher Systeme eine immer größere Herausforderung dar. Hierbei handelt es sich einerseits um Herausforderungen bezogen auf die Kommunikation hinsichtlich Echtzeitfähigkeit und effizienter Bandbreitennutzung, andererseits werden geeignete Methoden benötigt, um den Entwicklungsprozess solcher komplexen verteilten Systeme durch Tests und Evaluationen zu unterstützen und zu begleiten. Die hier vorgestellte Arbeit konzentriert sich auf diese beiden Teilaspekte: Die Entwicklung neuer Kommunikationslösungen sowie Methoden, um den Entwicklungsprozess durch kontinuierliche Tests zu unterstützen.

Die Notwendigkeit, neue Kommunikationslösungen zu entwickeln, resultiert aus den gestiegenen und veränderten Kommunikationsanforderungen. Aufgrund der wachsenden Funktionalität nimmt zum einen die auszutauschende Datenmenge zu, zum anderen führt der Einsatz in sicherheitskritischen Anwendungen dazu, dass klassische ereignisgetriggerte Ansätze nicht genügen. Daher kommt vermehrt zeitgetriggerte Kommunikation zum Einsatz, welche in der Lage ist, deterministische Garantien bezüglich der Echtzeitfähigkeit zu gewähren. Diese klassischen auf exklusiven Reservierungen basierenden Ansätze sind jedoch gerade bei (seltenen) sporadischen Nachrichten sehr ineffizient in Bezug auf die Nutzung der Bandbreite. Aus diesem Grund untersuchen wir in dieser Arbeit Ansätze, um die *modusbasierte Kommunikation* (*Mode-Based Scheduling with Fast Mode-Signaling*) in aktuelle Kommunikationssysteme (TTCAN, FlexRay, drahtlose Ad-Hoc-Netze) zu integrieren. Bei *Mode-Based Scheduling with Fast Mode-Signaling* handelt es sich um ein Verfahren, welches die Vorteile zeit- und ereignisgetriggerter Kommunikationstechnologien kombiniert und eine effizientere Bandbreitennutzung ermöglicht, dabei jedoch gleichzeitig Echtzeitfähigkeit sowie deterministische Garantien gewährleistet. Um dies zu ermöglichen, erlaubt *Mode-Based Scheduling* einen kontrollierten, slotbasierten Wettbewerb, welcher durch eine schnelle Modussignalisierung (*Fast Mode-Signaling*) aufgelöst wird. Weil es sich hierbei um den Schlüssel für die Umsetzung von *Mode-Based Scheduling* handelt, konzentriert sich diese Arbeit verstärkt auf die zuverlässige, robuste und deterministische Realisierung des *Fast Mode-Signaling* im Kontext der verschiedenen Technologien.

Der zweite Aspekt, mit dem sich diese Arbeit auseinandersetzt, betrifft die Entwicklung komplexer verteilter eingebetteter Echtzeitsysteme und beschäftigt sich damit, wie deren Entwicklungsprozess begleitet

werden kann und welche Ansätze es gibt, das Gesamtverhalten eines solchen Systems zu evaluieren und zu testen. Hierbei konzentrieren wir uns auf den Einsatz von Virtual Prototyping. Virtual Prototyping ermöglicht nicht nur den Test einzelner funktionaler Komponenten, sondern erlaubt es, das Gesamtverhalten des Systems sowie Performance-Aspekte in einer realistisch simulierten Umgebung zu evaluieren. Wir stellen in dieser Arbeit mit FERAL ein im Kontext des Innovationszentrums für *Applied Systems Modeling* mitentwickeltes Framework für die Kopplung spezialisierter Simulatoren vor, welches entworfen wurde, um komplexe verteilte Echtzeitsysteme und deren Umgebung zu simulieren. Die in diesem Kontext entwickelten Simulatoren für Kommunikationsprotokolle erlauben eine realistische Simulation der Interaktionen der Komponenten eines verteilten Echtzeitsystems und die hieraus resultierenden Auswirkungen auf das Gesamtverhalten. Zusätzliche Konzepte und Erweiterungen gestatten es, FERAL in frühen Entwicklungsphasen für die Evaluation von Designalternativen hinsichtlich geeigneter Kommunikationsprotokolle zu verwenden.

Entsprechend dieser beiden ausgewählten Teilaspekte ist auch die Dissertation in zwei Teile gegliedert. Jeder Teil widmet sich einem dieser Aspekte und stellt die im Rahmen dieser Arbeit entwickelten Konzepte und Lösungen vor.

Teil 1 beschäftigt sich mit der Integration von *Mode-Based Scheduling* in verschiedene Kommunikationsprotokolle und konzentriert sich dabei vor allem auf eine zuverlässige, deterministische Realisierung des *Fast Mode-Signaling* unter realen Anwendungsbedingungen. Nachdem wir uns zunächst mit Anwendungsmustern für *Mode-Based Scheduling* auseinandergesetzt haben, widmen wir uns der Erweiterung des TTCAN-Protokolls zur Unterstützung von *Mode-Based Scheduling*. Neben den konzeptionellen Anpassungen von TTCAN werden hierbei auch technische Erweiterungen diskutiert, welche eine deterministische Umsetzung von *Fast Mode-Signaling* auf Basis der CAN-Arbitrierung ermöglichen. Anschließend wird ein alternativer Ansatz für die Realisierung von *Fast Mode-Signaling* mittels Backoffslots entwickelt, der nur sehr geringe technische Voraussetzungen benötigt und sich daher ideal eignet, um *Mode-Based Scheduling* in bereits existierende Protokolle zu integrieren. Dies demonstrieren wir anhand der Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in das FlexRay-Protokoll sowie der Vorstellung von Umsetzungsmöglichkeiten für drahtlose Netzwerke.

Teil 2 stellt zunächst Aufbau und Struktur von FERAL vor, welches in einer Kooperation mit dem Fraunhofer IESE im Rahmen eines Transferprojekts entwickelt wurde. Da es sich bei FERAL um ein Framework für die Kopplung von Simulatoren handelt, haben wir uns zunächst mit einem geeigneten Adaptionsschema für die systematische Integration neuer Simulatoren beschäftigt. Dieses Adaptionsschema wurde dann angewendet, um FERAL durch Simulatoren für Matlab Simulink-Modelle und SDL-Spezifikationen zu erweitern. Hierdurch können sowohl Simulink-Modelle als auch SDL-Spezifikationen zur Beschreibung des Verhaltens der einzelnen funktionalen Komponenten eines verteilten Echtzeitsystems genutzt werden. Um die realistische Simulation des Kommunikationsaspekts mit FERAL zu ermöglichen, wurden spezielle Simulatoren für CAN, FlexRay sowie ein abstraktes Kommunikationsmedium entwickelt. Zusätzliche Konzepte in Form von Bridges und Gateways unterstützen die einfache Evaluation von Designalternativen hinsichtlich einer für das jeweilige Szenario geeigneten Kommunikationstechnologie. Abschließend präsentieren wir mittels Anwendungsszenarien, wie Virtual Prototyping mit FERAL sowohl für die Evaluation von Designalternativen als auch begleitend über verschiedene Entwicklungsphasen hinweg (bis hin zur Integration auf der Zielplattform) für Tests und Evaluationen eingesetzt werden kann.

Teil I

Mode-Based Scheduling with Fast Mode-Signaling

2 Einführung in Mode-Based Scheduling with Fast Mode-Signaling

Kommunikationslösungen für verteilte eingebettete Systeme basieren heute hauptsächlich auf einem zeit- oder ereignisgetriggerten Kommunikationsparadigma [Kop97]. Beide Lösungsansätze verfügen über spezifische Vor- und Nachteile, deren Ausprägung in Abhängigkeit von dem Kommunikationsprofil, dem Nachrichtenaufkommen sowie den Charakeristika der Nachrichten (periodisch oder sporadisch) unterschiedlich stark zur Geltung kommen. Während zeitgetriggerte Kommunikationslösungen deterministische Garantien bezüglich der maximal auftretenden Verzögerung ermöglichen, erlaubt eine ereignisgetriggerte Kommunikation eine bessere Nutzung der verfügbaren Bandbreite bei sporadischen Nachrichten. Die Wahl einer dieser beiden Lösungen stellt somit stets einen Kompromiss zwischen dem garantierbaren Grad der Echtzeitfähigkeit und der Effizienz in Bezug auf die Nutzung der Bandbreite dar. Aus diesem Grund kombinieren Protokolle, wie FlexRay [Fle10b] oder TTCAN [ISO04], zeit- und ereignisgetriggerte Kommunikationsparadigmen. Beide Protokolle erlauben zeitlich disjunkte Abschnitte, in denen jeweils eines der beiden Paradigmen zum Einsatz kommt. Abhängig vom Szenario ermöglicht dies eine bessere Kombination der Vorteile bzw. Abschwächung der Nachteile der Paradigmen.

Ziel dieser Arbeit ist jedoch nicht die Optimierung von Ablaufplänen klassischer Kommunikationsparadigmen; stattdessen verwenden wir in diesem Teil der Arbeit mit *Mode-Based Scheduling with Fast Mode-Signaling* einen neuen in [KG13] vorgestellten Ansatz, welcher die Vorteile zeit- und ereignisgetriggerter Kommunikation kombiniert. Die *modusbasierte Kommunikation* (*Mode-Based Scheduling with Fast Mode-Signaling*) ermöglicht eine Verbesserung der Bandbreitennutzung bei gleichzeitiger Gewährleistung der Echtzeitfähigkeit sowie deterministischer Garantien. Erreicht wird dieses Ziel durch Beschränkung des Wettbewerbs auf eine ausgewählte Menge von Knoten und Nachrichten(-typen) innerhalb eines Zeitslots (*Mode-Based Scheduling*) in Kombination mit einer Technik (*Fast Mode-Signaling*) zur effizienten netzweiten Auflösung von Zugriffskonflikten auf Basis von Prioritäten (*Modus-Präferenzen*). Dieser Teil der Arbeit beschäftigt sich mit der robusten, zuverlässigen und vor allem deterministischen Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* auf Basis existierender Feldbusse (TTCAN und FlexRay) sowie der Entwicklung von Konzepten, welche eine einfache Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in weitere Protokolle und Kommunikationstechnologien (wie drahtlose Ad-Hoc-Netze) ermöglichen.

Der erste Teil dieser Arbeit ist wie folgt strukturiert: **Kapitel 2** widmet sich der grundlegenden Beschreibung und Definition der Konzepte von *Mode-Based Scheduling with Fast Mode-Signaling*. Die nachfolgenden Kapitel beschäftigen sich mit dessen zuverlässiger Umsetzung für unterschiedliche etablierte Kommunikationstechnologien. So stellt **Kapitel 3** eine technische Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* für CAN-Busse vor, welche in der Automobilindustrie sowie im Produktionsumfeld weit verbreitet sind. Nach der Betrachtung eines allgemeinen, technologieunabhängigen Ansatzes zur Implementierung von *Fast Mode-Signaling* in **Kapitel 4**, wenden wir in **Kapitel 5** diese Ergebnisse an, um die Nutzbarkeit von *Mode-Based Scheduling* im Rahmen drahtloser Kommunikationstechnologien zu untersuchen. Abschließend betrachten wir in **Kapitel 6** die Möglichkeiten einer Integration in das zeitgetriggerte FlexRay-Protokoll.

2.1 Ausgangssituation und Problemstellung

Bei einer zeitgetriggerten Kommunikation werden Nachrichten nur zu vorherbestimmten, festgelegten Zeitpunkten übertragen. In der Praxis erreicht man dies durch die Unterteilung des Mediums in Zeitslots. Diese werden in der Designphase (offline) exklusiv für einzelne Knoten oder Nachrichtentypen reserviert, sodass zur Laufzeit keine Kollisionen auftreten. Diese Form des Medienzugriffsverfahrens wird als TDMA (*Time division multiple access* [Tan03]) bezeichnet. Im Gegensatz hierzu wird bei der ereignisgetriggerten Kommunikation eine Übertragung durch das Eintreten von Ereignissen ausgelöst. Ein solches Ereignis könnte beispielsweise die Überschreitung eines zuvor festgelegten Schwellenwertes sein. Treten mehrere Ereignisse zeitgleich auf, werden bei der ereignisgetriggerten Kommunikation Strategien benötigt, um resultierende Zugriffskonflikte aufzulösen. Ein entsprechendes Verfahren, welches dies leistet, ist CSMA/CA und wird z. B. von CAN genutzt.

Durch den festgelegten Ablaufplan mit exklusiven Reservierungen ergibt sich bei der zeitgetriggerten Kommunikation ein deterministisches zeitliches Verhalten, welches Garantien hinsichtlich der maximal auftretenden Verzögerungen ermöglicht. Ist das Zwischenankunftsintervall der zu übertragenden Nachrichten strikt periodisch, so können die Reservierungen so gewählt werden, dass die Bandbreite optimal genutzt und die Verzögerung zwischen dem Erstellen einer Nachricht sowie deren Übertragung minimiert wird (und somit auch der Jitter der Nachrichten). Bei zeitgetriggerten Protokollen müssen für sporadische Nachrichten mit maximalen Antwortzeiten für Slotreservierungen sowohl die Antwortzeiten als auch das minimale Intervall zwischen dem Auftreten der Nachrichten berücksichtigt werden, um deterministische Garantien gewährleisten zu können. Insbesondere bei sehr kleinen maximalen Antwortzeiten in Verbindung mit sehr geringen Auftrittswahrscheinlichkeiten führt dieses Vorgeben dazu, dass viele der exklusiv reservierten Slots ungenutzt bleiben. Ein Beispiel aus der Praxis für eine Klasse von Ereignissen, welche ein ähnliches Anforderungsprofil aufweisen, sind *Angular Events*. Hierbei handelt es sich um periodisch auftretende Ereignisse, deren Periodendauer variiert und von externen physikalischen Vorgängen oder Größen abhängen, deren Regelung jedoch extrem kleine Antwortzeiten erfordert. *Angular Events* finden sich etwa in der Motorregelung, welche den Zündzeitpunkt, abhängig von der Position der Kurbelwelle, bestimmen muss. Dort hängt das Zwischenankunftsintervall der Nachricht des Kurbelwellensensors direkt von der Motordrehzahl ab. Zu große Antwortzeiten führen dazu, dass die Zündzeitpunkte nicht korrekt angesteuert werden und der Motor *klopft* (vgl. [FHR+04, AGAT11]).

Im Gegensatz zur zeitgetriggerten Kommunikation können bei der ereignisgetriggerten Kommunikation Nachrichten sofort übertragen werden, und man muss nicht erst auf den reservierten Slot warten. Des Weiteren wird nur Bandbreite beansprucht, wenn auch wirklich eine Nachricht übertragen werden muss. Bei der zeitgetriggerten Kommunikation hingegen bleibt ein reservierter Slot ungenutzt, falls keine Nachricht zur Übertragung vorliegt. Daher ist der ereignisgetriggerte Ansatz hinsichtlich der Nutzung der Bandbreite bei seltenen sporadischen Nachrichten effizienter. Treten sporadische Nachrichten jedoch nicht vereinzelt, sondern zeitlich gebündelt auf, sind durch die resultierenden Zugriffskonflikte beim ereignisgetriggerten Paradigma nur begrenzte Echtzeitgarantien möglich.

In Bezug auf die Fragestellung, ob das zeitgetriggerte oder ereignisgetriggerte Kommunikationsparadigma besser geeignet ist, um die Anforderungen verteilter Echtzeitsysteme zu erfüllen, gibt es in der Literatur unterschiedliche Meinungen. Die Diskussionen beschränken sich häufig nicht nur auf das Kommunikationsparadigma, sondern auch auf das zugrundeliegende Programmiermodell[1] und betreffen somit letztendlich die Architektur des Gesamtsystems [Kop97, SSP06]. Ein Versuch, die beiden Kommunikationsparadigmen zu vergleichen, ist unter anderem in [Kop91] zu finden, wobei Kopetz eher eine zeitgetriggerte Architektur als Lösung favorisiert und vertritt.

In der Praxis kommen in umfangreichen verteilten Echtzeitsystemen im Allgemeinen sowohl periodische als auch sporadische Nachrichtentypen vor. Somit fällt eine Wahl für eines der beiden Kommunikations-

[1] Zu dem Programmiermodell gehört z. B., ob innerhalb des Systems nur Ereignisse oder periodisch Zustände ausgetauscht werden, und wie die Ausführung der funktionalen Komponenten organisiert ist.

paradigmen schwer und stellt letztendlich stets einen Kompromiss dar. Um die resultierenden Nachteile abzuschwächen, erlauben aktuelle Kommunikationsprotokolle, wie FlexRay oder TTCAN, die Kombination von zeit- und ereignisgetriggerter Kommunikation in strikt voneinander getrennten Bereichen. Doch auch hierdurch wird die grundlegende Problematik nicht gelöst: Innerhalb der Abschnitte mit ereignisgetriggerter Kommunikation konkurrieren alle Nachrichtentypen miteinander, sodass Echtzeitgarantien nur sehr eingeschränkt möglich sind[2]. Dies ist insbesondere dann problematisch, wenn innerhalb des Systems viele sporadische Nachrichten mit sehr kurzer Deadline existieren. Werden hier deterministische Garantien benötigt, so müssen die sporadischen Nachrichten trotzdem zeitgetriggert übertragen werden. Was wiederum bedeutet, dass für die sporadischen Nachrichten exklusive Slotreservierungen benötigt werden – in der Regel mindestens ein Mikroslot pro Makroslot und Nachricht, abhängig von Protokoll, Deadline und minimalem Auftrittsintervall der Nachrichten. Bei seltenen sporadischen Nachrichten führt dies dazu, dass die reservierten Slots häufig ungenutzt bleiben, ohne dass die exklusiv reservierte Bandbreite anderweitig genutzt werden kann. Im Automobilbereich finden sich solche seltenen sporadischen Nachrichten mit hohen Anforderungen z. B. im Bereich sicherheitskritischer X-by-Wire Anwendungen. Brake-by-Wire- oder Steer-by-Wire-Lösungen entfernen mechanische Rückfalllösungen, z. B. für Lenkung oder Bremse, und realisieren deren Ansteuerung allein über Nachrichten. Es ist offensichtlich, dass diese Typen von Nachrichten extrem hohe Anforderungen an die Zuverlässigkeit sowie an die Vorhersagbarkeit der auftretenden Verzögerungen stellen und daher deterministische Garantien zwingend erforderlich sind [WNS+05, FPAF02b].

Mit *Mode-Based Scheduling with Fast Mode-Signaling* wird in [KG13] ein Verfahren vorgestellt, welches es erlaubt, Slotreservierungen für seltene sporadische Nachrichten mit hohen Echtzeitanforderungen für anderweitige Kommunikation zu nutzen, sofern diese ansonsten ungenutzt blieben, ohne dabei deterministische Garantien oder deren Echtzeitanforderungen zu verletzen. *Mode-Based Scheduling with Fast Mode-Signaling* erreicht durch seine dynamische Zuordnung von Zeitslots zu Nachrichten eine deutlich effizientere Nutzung der Bandbreite, als dies bei statischen Slotzuordnungen möglich ist. Hierfür kombiniert *Mode-Based Scheduling with Fast Mode-Signaling* zwei unterschiedliche Konzepte miteinander: *Mode-Based Scheduling* und *Fast Mode-Signaling*.

Die grundlegende Idee von *Mode-Based Scheduling* besteht darin, zur Laufzeit innerhalb eines Slots einen eingeschränkten deterministischen Wettbewerb zwischen ausgewählten Nachrichtentypen zuzulassen. Hierzu werden den Nachrichtentypen pro Zeitslot verschiedene *Transmission Modes* (kurz *Modes*) sowie *Modus-Präferenzen* zugeordnet. Die Modes dienen der Charakterisierung der Nachrichten (z. B. *Emergency*), wohingegen die *Modus-Präferenzen* den Vorrang eines Modes gegenüber einem anderen ausdrücken (Priorisierung).

Fast Mode-Signaling bezeichnet die eingesetzte Methodik, um einen netzwerkweiten deterministischen Konsens darüber herbeizuführen, welche Nachricht – bei mehreren innerhalb eines Slots zu übertragenden Nachrichten – diejenige ist, deren Mode die höchste *Modus-Präferenz* aufweist. Dementsprechend wichtig ist es, dass *Fast Mode-Signaling* sowohl sehr schnell (effizient) als auch zuverlässig arbeitet, um die Bandbreite möglichst vollständig für die eigentlichen Übertragungen nutzen zu können[3]. *Mode-Based Scheduling with Fast Mode-Signaling* kombiniert diese beiden Konzepte, um innerhalb eines Slots jeweils die Nachricht zu übertragen, deren Mode die höchste *Modus-Präferenz* zugeteilt wurde.

Ziel dieses Kapitels ist es, eine Einführung in die Grundlagen von *Mode-Based Scheduling with Fast Mode-Signaling*, einem Verfahren zur Verbesserung der Bandbreitennutzung für zeitgetriggerte Kommunikationsprotokolle, zu geben, das erstmals in [KG13] von Reinhard Gotzhein und Thomas Kuhn vorgestellt wurde. Die Patentschriftt [KG13] konzentriert sich auf die Beschreibung des allgemeinen Verfahrens, dessen grundlegende Konzepte sowie einer konzeptionellen Lösung für die Umsetzung von

[2] Bei CAN beispielsweise sind Garantien auf die Nachrichten mit der höchsten Priorität (d.h. dem kleinsten CAN-Identifier) beschränkt.

[3] Der Austausch zusätzlicher Nachrichten, um eine Entscheidung über die Nutzung des Slots zu erzielen, gehört z. B. nicht zu einer effizienten Umsetzungsmöglichkeit von *Fast Mode-Signaling*.

Fast Mode-Signaling auf Basis von CAN. Der Schwerpunkt dieser Arbeit liegt in der Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in verschiedene Kommunikationsprotokolle. Hierzu entwickeln wir zwei verschiedene Ansätze zur robusten, zuverlässigen und deterministischen Realisierung des *Fast Mode-Signaling*. Die erste Variante basiert auf der in [KG13] nur konzeptionell beschriebenen Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* für TTCAN (anhand der Abbildung des *Fast Mode-Signaling* mittels der CAN-Arbitrierung). In dieser Arbeit definieren wir eine vollwertige Erweiterung von TTCAN und entwickeln Lösungen zur deterministischen Realisierung von *Fast Mode-Signaling* mittels der CAN-Arbitierung unter realen Anwendungsbedingungen, von denen [KG13] abstrahiert (Kapitel 3). Anschließend präsentieren wir einen technologieunabhängigen Ansatz zur Realisierung des *Fast Mode-Signaling* und demonstrieren mit diesem die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in drahtlose Ad-Hoc-Netze auf Basis von IEEE 802.15.4 [IEE03] (Kapitel 5) sowie FlexRay (Kapitel 6).

Zur Vermittlung der Grundlagen von *Mode-Based Scheduling* beschreiben wir zunächst ein ebenfalls in [KG13] vorgestelltes Anwendungsbeispiel aus dem Automobilbereich und illustrieren an diesem die Schwächen klassischer Kommunikationsparadigmen (Kapitel 2.2). Anschließend führen wir in Kapitel 2.3 die Grundlagen von *Mode-Based Scheduling with Fast Mode-Signaling* formal ein. Nach der Vorstellung der Vorarbeiten aus [KG13] diskutieren wir typische Anwendungsmuster für *Mode-Based Scheduling* (Kapitel 2.4), die im Rahmen dieser Arbeit entwickelt wurden. Dann erörtern wir den aktuellen Stand der Technik (Kapitel 2.5) und liefern in Kapitel 2.6 einen Ausblick auf das weitere Vorgehen.

2.2 Anwendungsbeispiel

Zum besseren Verständnis und zur Illustration der zuvor beschriebenen Eigenschaften und Einschränkungen der Kommunikationsparadigmen erfolgt zunächst die Betrachtung eines stark vereinfachten Szenarios aus dem Automobilbereich [KG13]. Für dieses werden Ablaufpläne einer ereignis- und zeitgetriggerten Kommunikation sowie Kombinationen beider präsentiert und mit einem Ablaufplan mit *Mode-Based Scheduling* verglichen. Das vorgestellte Szenario ist konstruiert und wäre in dieser Form nicht zulassungsfähig, da Knoten und Funktionalitäten unterschiedlichster Sicherheitsklassen direkt miteinander vernetzt sind. Das Beispiel wurde bewusst so konstruiert, dass die verschiedenen Nachrichtencharakteristika realer Echtzeitsysteme vertreten und die Kommunikationsanforderungen und -charakteristika aufgrund der Funktionalitäten der Knoten nachvollziehbar sind.

Das Szenario besteht aus sechs miteinander kommunizierender Knoten $v_1, \ldots, v_6$, die über einen gemeinsamen Bus miteinander verbunden sind. Die Tabelle 2.1 fasst die Nachrichtentypen sowie deren Charakterstika zusammen. So sendet Knoten v_1 periodisch Daten aus dem Entertainment-Bereich (z. B. Audio- oder Videostream). Charakterisiert ist diese Übertragung durch ein sehr hohes Datenvolumen, in Verbindung mit einer möglichst niedrigen und konstanten Übertragungsverzögerung. Die Knoten v_2 und v_3 sind für die Übertragung von Steueranweisungen zur Sitz- sowie der Spiegelpositionierung verantwortlich. Hierbei handelt es sich um seltene sporadische Ereignisse, die jedoch einer oberen Schranke in Bezug auf die maximale Übertragungsverzögerung bedürfen. Knoten v_4 und v_5 generieren sicherheitskritische Nachrichten aus dem X-by-Wire Bereich (hier Nachrichten für Steer- und Brake-by-Wire-Systeme [WNS+05]). Hierbei handelt es sich um temporäre, aber bei ihrem Auftreten (beim Lenken bzw. Bremsen), periodische Datenströme, die eine garantierte niedrige und konstante Verzögerung sowie eine hohe Zuverlässigkeit verlangen. Nachrichten zum Auslösen des Airbags werden von Knoten v_6 gesendet. Zwar handelt es sich um ein extrem seltenes Ereignis, dieses erfordert jedoch eine sehr niedrige Übertragungsverzögerung bei gleichzeitig sehr hohen Anforderungen an die Zuverlässigkeit.

Bei einem ereignisgetriggerten Protokoll müssen Konflikte bei gleichzeitigen Medienzugriffen aufgelöst werden. Dies erfolgt bei CAN beispielsweise durch die Vergabe globaler Prioritäten sowie einem Arbitrierungsverfahren, welches im Falle eines gleichzeitigen Zugriffs sicherstellt, dass die Nachricht mit

Knoten	Nachrichtentyp	Nachrichten-charakteristika	Anforderungen
v_1	*Entertainment*	Isochroner Datenstrom	kleine konstante Verzögerung, hohes Datenvolumen
v_2	*Spiegelpositionierung*	Seltene sporadische Ereignisse	beschränkte Übertragungsverzögerung
v_3	*Sitzpositionierung*	Seltene sporadische Ereignisse	beschränkte Übertragungsverzögerung
v_4	*Steer-by-Wire*	Temporäre Ereignisströme	garantierte niedrige, konstante Übertragungsverzögerung, hohe Zuverlässigkeit
v_5	*Brake-by-Wire*	Temporäre Ereignisströme	garantierte niedrige, konstante Übertragungsverzögerung, hohe Zuverlässigkeit
v_6	*Airbagsteuerung*	Sehr seltene sporadische Ereignisse	garantierte sehr niedrige, konstante Übertragungsverzögerung, sehr hohe Zuverlässigkeit

Tabelle 2.1: Nachrichtentypen sowie deren Anforderungen und Charakteristika.

der höchsten Priorität gewinnt (vgl. Kapitel 3.1). Hierzu wird jedem Nachrichtentyp eine eindeutige, nicht veränderbare statische Priorität (zur Designzeit) zugeordnet. Die Tabelle 2.2 zeigt eine solche Zuteilung für die Nachrichtentypen[4] unseres Szenarios. Das Beispiel illustriert ein prinzipbedingtes Problem bei der Vergabe von eindeutigen Prioritäten. So ist es nicht möglich, dass zwei verschiedene Nachrichtentypen die gleiche Priorität besitzen, d.h., bei gleichwichtigen Nachrichtentypen existiert keine Zuordnung der Prioritäten, die verhindert, dass diese Nachrichten sich gegenseitig unbeschränkt verzögern können. Bei den hier gewählten Prioritäten können Brake-by-Wire Nachrichten Steer-by-Wire Nachrichten beliebig lange verzögern. Dies gilt analog für Sitz- bzw. Spiegelpositionierung, jedoch sind diese weniger sicherheitskritisch. In diesem Szenario werden die Nachrichten des Typs *Entertainment* durch alle anderen Nachrichten verzögert, sodass hierdurch die Isochronität nicht garantiert werden kann.

Knoten	Nachrichtentyp	Priorität
v_1	*Entertainment*	10
v_2	*Spiegelpositionierung*	8
v_3	*Sitzpositionierung*	7
v_4	*Steer-by-Wire*	2
v_5	*Brake-by-Wire*	1
v_6	*Airbagsteuerung*	0

Tabelle 2.2: Zuordnung der Prioritäten für eine ereignisgetriggerte Kommunikation.

Bei einer rein zeitgetriggerten Kommunikationstechnologie, wie z. B. FlexRay (vgl. Abschnitt 6.1), die auf exklusiven Slotreservierungen beruht, muss zunächst die Länge des Makroslots festgelegt werden. Der hier verwendete Makroslot ist sehr kurz, sodass lediglich 15 frei nutzbare Slots[5] zur Verfügung stehen.

[4] Hierbei entspricht ein niedrigerer Zahlenwert einer höheren Priorität.

[5] Dieses Beispiel bezieht sich auf keine konkret existierende Kommunikationstechnologie, sondern dient lediglich der Illustration.

Tabelle 2.3 beziffert, wie viele Slots pro Makroslot für die jeweiligen Nachrichtentypen reserviert werden müssten, um die Anforderungen bei der gegebenen Makroslotlänge zu erfüllen. Die Nachrichten des Typs *Entertainment* benötigen, aufgrund des hohen Datenvolumens, 8 Slots pro Makroslot. Für die seltenen Positionierungsnachrichten von Spiegel und Sitz genügt ein Slot pro Makroslot. Um die Anforderungen bezüglich der benötigten maximalen Übertragungsverzögerung für Steer-by-Wire und Brake-by-Wire Nachrichten garantieren zu können, müssten diesen, pro Makroslot und Nachrichtentyp, jeweils 4 Slots zugeteilt werden. Aufgrund der hohen Anforderungen an die Reaktionszeit werden für die Airbagsteuerung zwei Slots pro Makroslot reserviert. Insgesamt werden somit 20 Slots pro Makroslot benötigt, d.h., ein Bus mit einem Kommunikationszyklus bestehend aus 15 Slots ist nicht ausreichend. Eine Lösung besteht darin, die Übertragungsrate zu erhöhen, sodass der Makroslot in 20 Slots unterteilt werden kann oder zwei getrennte Busse zu verwenden. Unabhängig davon bleibt der Nachteil bestehen, dass viele Slots zur Laufzeit nicht verwendet werden, da die sporadischen Nachrichten (X-by-Wire, Airbagsteuerung, Positionierungsnachrichten) bzw. die jeweiligen dazugehörenden Ereignisse nur selten auftreten. Somit wird ein großer Teil der verfügbaren Bandbreite nicht (effizient) genutzt.

Knoten	**Nachrichtentyp**	**Zeitslots pro Makroslot**
v_1	*Entertainment*	8
v_2	*Spiegelpositionierung*	1
v_3	*Sitzpositionierung*	1
v_4	*Steer-by-Wire*	4
v_5	*Brake-by-Wire*	4
v_6	*Airbagsteuerung*	2

Tabelle 2.3: Slotreservierung bei zeitgetriggerter Kommunikation.

Bei einer Kommunikationstechnologie, die zeit- und ereignisgetriggerte Kommunikation unterstützt, wird die zeitgetriggerte Kommunikation über exklusive Slotreservierungen und die ereignisgetriggerte Kommunikation unter Verwendung eindeutiger globaler Prioritäten realisiert[6]. Wir gehen wieder von einem Makroslot bestehend aus 15 Slots aus und weisen periodischen Nachrichtenströmen (*Entertainment*) sowie temporären Nachrichtenströmen (*X-by-Wire*) jeweils exklusive Slotreservierungen zu. Sporadische Nachrichten (*Airbagsteuerung*, *Spiegel-* sowie *Sitzpositionierung*) teilen sich einen gemeinsamen Slot. Zugriffskonflikte werden auf Basis der zugewiesenen Prioritäten aufgelöst. Die Tabelle 2.4 nennt die Anzahl der reservierten Slots pro Makroslot sowie die zugewiesenen Prioritäten. Insgesamt werden für diesen Ablaufplan 17 Slots benötigt. Somit ist dieser geringfügig besser als die rein zeitgetriggerte Lösung, erfüllt jedoch immer noch nicht die Vorgabe von maximal 15 Slots pro Makroslot. Auch die Probleme hinsichtlich der Priorisierung der Nachrichten für Spiegel- und Sitzpositionierung bleiben bestehen.

Im nächsten Schritt wird *Mode-Based Scheduling* eingeführt, ein alternatives Verfahren für die Zuordnung von Zeitslots. Dieses ermöglicht es, die harten Echtzeitanforderungen des Szenarios zu erfüllen und dabei gleichzeitig mit der verfügbaren Bandbreite von 15 Slots auszukommen. Zusätzlich bietet *Mode-Based Scheduling* eine Lösung für den Umgang mit gleichwichtigen Nachrichtentypen an.

[6] Diese Funktionalität wird z. B. von TTCAN (vgl. Abschnitt 3.1.2) angeboten.

Knoten	Nachrichtentyp	Priorität / Slots pro Makroslot (Ereignisgetriggert)	Zeitslots pro Makroslot (Zeitgetriggert)
v_1	*Entertainment*		8
v_2	*Spiegelpositionierung*	8/1	
v_3	*Sitzpositionierung*	7/1	
v_4	*Steer-by-Wire*		4
v_5	*Brake-by-Wire*		4
v_6	*Airbagsteuerung*	0/1	

Tabelle 2.4: Kombination von zeit- und ereignisgetriggerter Kommunikation.

2.3 Formale Definition von Mode-Based Scheduling with Fast Mode-Signaling

Im Folgenden werden die Grundlagen von *Mode-Based Scheduling with Fast Mode-Signaling* formal eingeführt und die jeweiligen Definitionen anhand eines Beispiels aus [KG13] illustriert. Zuerst erfolgt die Definition eines abstrakten Kommunikationsmodells, basierend auf den allgemeinen Konzepten zeitgetriggerter Paradigmen, welches als Ausgangsbasis für die weiteren Definitionen von *Mode-Based Scheduling* dient. Das Kommunikationsmodell ist soweit verallgemeinert, dass es sich auf konkrete Kommunikationstechnologien wie CAN und FlexRay anwenden lässt, wie die Kapitel 3 und 6 demonstrieren. Abschließend wird mit *Fast Mode-Signaling* ein effizientes deterministisches Verfahren vorgestellt, welches im Falle mehrerer sendebereiter Rahmen, die dem gleichen Slot zugeordnet sind, netzwerkweit denjenigen mit der höchsten Präferenz (Priorität) propagiert.

Zusammen bilden *Mode-Based Scheduling* und *Fast Mode-Signaling* das in [KG13, BGK14] entwickelte Verfahren *Mode-Based Scheduling with Fast Mode-Signaling*, dessen primäres Anwendungsziel die Verbesserung der Bandbreitennutzung bei zeitgetriggerter Kommunikation ist, bei gleichzeitiger Gewährung deterministischer Echtzeitgarantien.

2.3.1 Kommunikationsmodell

Das Kommunikationsmodell abstrahiert von technischen und implementierungsspezifischen Details und konzentriert sich auf die grundlegenden Konzepte, wie der hierarchischen Unterteilung der Zeit in Zeitintervalle (Makroslots und Mikroslots). Technische Aspekte wie die in der Realität notwendige regelmäßige (Re-)Synchronisation sind nicht Teil dieses Modells, können aber über reservierte Zeitslots abgebildet werden.

Definition 2.1

Die Zeit ist unterteilt in eine unendliche Sequenz von (fortlaufend nummerierten) Zeitintervallen identischer Länge $\Theta = \{S_1, S_2, \dots\}$, die als Makroslots bezeichnet werden. Jeder Makroslot ist seinerseits unterteilt in eine endliche Menge $S = \{s_1, \dots, s_n\}$ von (fortlaufend nummerierten) Mikroslots (kurz Slots), die eine unterschiedliche aber feste Länge aufweisen dürfen. Jeder Mikroslot kommt genau einmal pro Makroslot vor. Zur Vereinfachung wird das Auftreten des Mikroslots s_j innerhalb des Makroslots $S_i \in \Theta$ kurz mit $s_{i,j}$ bezeichnet.

Die Definition von *Mode-Based Scheduling* beschränkt sich auf Single-Hop-Netzwerke mit einem geteilten Übertragungsmedium. Dies vereinfacht sowohl das Kommunikationsmodell als auch die nachfolgenden Definitionen, da eine Konnektivität zwischen allen Knoten gegeben ist. Die Beschränkung auf Single-Hop-Netzwerke ist in Bezug auf drahtgebundene Bussysteme (CAN, LIN, FlexRay etc.), wie

sie häufig bei Echtzeitsystemen zum Einsatz kommen, nicht relevant, da deren Kommunikationsmodell ebenfalls auf (zumindest logischen) Single-Hop-Topologien beruht. In Kapitel 5 werden wir jedoch auch kurz auf die Besonderheiten von Multi-Hop-Netzwerken eingehen sowie eine konzeptuelle Lösung für Muli-Hop-Funknetzwerke vorstellen.

Definition 2.2

Ein Single-Hop-Netzwerk besteht aus einer endlichen Menge von Knoten $V = \{v_1, \ldots, v_k\}$, die alle paarweise in Kommunikationsreichweite zueinander sind und über ein geteiltes Medium miteinander kommunizieren.

2.3.2 Mode-Based Scheduling

Als Nächstes erfolgt die formale Definition von *Mode-Based Scheduling*. Die Definitionen wird jeweils anhand der in Tabelle 2.1 aufgeführten Nachrichtentypen illustriert. Wie das Beispiel belegt, gelingt durch die Verwendung von *Mode-Based Scheduling* eine Abbildung der Anforderungen der Nachrichten auf einen im Extremfall, nur aus 4 Slots bestehenden Makroslot.

Wir beginnen mit der Einführung eines abstrakten Designkonzepts, den *Transmission Modes* oder kurz *Modes*. Modes dienen der Modellierung und können beispielsweise aus Betriebsmodi des zugrundeliegenden Systems oder den Charakteristika bzw. dem Zweck der Nachrichten selbst abgeleitet werden.

Definition 2.3

Transmission Modes oder auch kurz Modes, werden als nichtleere endliche Menge $M = \{m_1, \ldots, m_r\}$ modelliert.

Die Tabelle 2.5 legt die Modes für dieses Szenario fest und teilt diese den verschiedenen Nachrichtentypen zu. Die Wahl der Modes orientiert sich hier primär nach den Charakteristika und Zielsetzungen der Nachrichten. So wurde den Entertainment Nachrichten der Mode *Stream* zugeordnet; die beiden X-by-Wire Nachrichtentypen teilen sich den gemeinsamen Mode *Safety*. Wie in diesem Beispiel gezeigt, können verschiedenen Nachrichtentypen auch die gleichen Modes zugeordnet werden. In Bezug auf X-by-Wire Nachrichten könnte auch eine feingranulare Abstufung vorgenommen werden und z. B. bestimmte kritische X-by-Wire Nachrichten (ggf. in Abhängigkeit von der erkannten Fahrsituation) mit dem Mode *Emergency* assoziiert werden. Die Festlegung der Modes sowie deren Zuordnung ist Bestandteil der Designphase und während der Laufzeit unveränderlich.

Knoten	Nachrichtentyp	Mode
v_1	*Entertainment*	*Stream*
v_2	*Spiegelpositionierung*	*Regular*
v_3	*Sitzpositionierung*	*Regular*
v_4	*Steer-by-Wire*	*Safety*
v_5	*Brake-by-Wire*	*Safety*
v_6	*Airbagsteuerung*	*Emergency*

Tabelle 2.5: Beispiel für die Definition und Zuordnung der Modes bei dem Automobil-Szenario.

Für jeden Mikroslot und Mode wird festgelegt, welcher Knoten innerhalb dieses Slots Nachrichtentypen des entsprechenden Modes übertragen darf. Hierbei gilt, dass ein Mikroslot $s \in S$ für jeden Mode $m \in M$ **höchstens** einem Knoten zugeordnet werden darf. Daraus folgt, dass bei r definierten Modes $M =$

$\{m_1, \ldots, m_r\}$ ein Mikroslot bis zu r-mal einem beliebigen Knoten[7] zugeteilt werden kann. Eine solche Zuteilung eines Mikroslots $s \in S$ zu einem Knoten $v \in V$ für einen Mode $m \in M$ wird als Slot-Assignment bezeichnet und gilt für alle Instanzen des Slots $s \in S$ in allen Makroslots.

Definition 2.4

Sei V die Menge der Knoten eines Single-Hop-Netzwerks, M die Menge der Modes und S die Menge der Mikroslots. Dann ist $SA : S \times M \to V$ eine partielle Funktion und heißt Slot-Assignment-Funktion oder kurz Slot-Assignment.

Die Festlegung des Slot-Assignment ist – ebenso wie die Definition der Modes – Bestandteil der Designphase und während der Laufzeit unveränderbar. Ist die Slot-Assignment-Funktion für ein Paar (s,m) definiert mit $s \in S$, $m \in M$, so sind alle Instanzen des Mikroslots s eindeutig dem Knoten $SA(s,m)$ für den Mode m zugeordnet. Ist die Slot-Assignment-Funktion für das Paar (s,m) nicht definiert, so darf innerhalb des Slots s der Mode m nicht verwendet, d.h., kein Rahmen mit diesem Mode übertragen werden. Ein Spezialfall ist die Zuordnung eines Slots für nur einen einzigen Mode, dies entspricht einer exklusiven Reservierung. Somit ist dieser Fall ebenfalls mittels *Mode-Based Scheduling with Fast Mode-Signaling* abbildbar.

Die Tabelle 2.6a zeigt die Slot-Assignment-Funktion für ein Szenario mit einem Makroslot bestehend aus 4 Mikroslots. Wie unmittelbar aus der Tabelle ersichtlich, wird die Einschränkung, dass jedes Paar aus Slot und Mode maximal einem Knoten zugeordnet werden darf, eingehalten.

SA	s_1	s_2	s_3	s_4
Emergency	v_6		v_6	
Safety	v_5	v_4	v_4	v_5
Regular		v_2		v_3
Stream	v_1	v_1	v_1	v_1

(a) *Slot-Assignment*-Funktion

MP	s_1	s_2	s_3	s_4
Emergency	0		0	
Safety	1	0	1	0
Regular		5		5
Stream	7	7	7	7

(b) *Modus-Präferenz*-Funktion

Tabelle 2.6: Modusbasierter Ablaufplan für das verkürzte Automobil-Szenario.

Typischerweise wird ein Mikroslot verschiedenen Knoten (für unterschiedliche Modes) zugeordnet (vgl. Tabelle 2.6a). Da pro Slot aber nur eine Übertragung zeitgleich erfolgen kann, wird eine Lösung benötigt, um diesen Konflikt aufzulösen. Das heißt, im Falle mehrerer anstehender Übertragungen in einem Mikroslot wird ein Verfahren benötigt, um deterministisch den Knoten samt Mode zu bestimmen, der seinen Rahmen übertragen darf. Bei *Mode-Based Scheduling* wird dies durch die Einführung von – pro Mikroslot und Mode – eindeutigen *Modus-Präferenzen* gelöst (Definition 2.5). *Modus-Präferenzen* ordnen jedem eingeplanten Mode (Slot-Assignment-Funktion) für den Mikroslot eine eindeutige Priorität zu. Bei *Mode-Based Scheduling* sendet der Knoten seinen Rahmen, dessen Mode die höchste *Modus-Präferenz* für den Mikroslot zugeordnet wurde.

[7] Hierbei darf ein Mikroslot auch mehrmals dem gleichen Knoten für unterschiedliche Modes zugeteilt werden.

Definition 2.5

Sei V die Menge der Knoten eines Single-Hop-Netzwerks, M die Menge der Modes, S die Menge der Mikroslots und SA eine Slot-Assignment-Funktion. Dann ist die Modus-Präferenz-Funktion $mp : S \times M \rightarrow \mathbb{N}_0$ *eine partielle Funktion, die jedem Paar aus Mikroslot und Mode für das SA definiert ist, eine für den Mikroslot eindeutige Priorität zuordnet. Das heißt, für mp muss gelten, dass*

$$\forall s \in S, m_1 \in M, m_2 \in M.(mp(s,m_1) = mp(s,m_2) \Rightarrow (m_1 = m_2))$$

In der hier verwendeten Modellierung entspricht ein niedrigerer Wert einer höheren Priorität.

Tabelle 2.6b zeigt die Zuordnung von *Modus-Präferenzen* zu den verschiedenen Modes in diesem Beispiel. Die *Modus-Präferenzen* wurden so gewählt, dass Nachrichten des Modes *Emergency* (Airbagsteuerung) jeweils die höchste Priorität besitzen und daher nicht durch andere Nachrichten verzögert werden. Nachrichten der Modes *Safety* (X-by-Wire) haben die Priorität 1, sofern diese in dem gleichen Slot wie Nachrichten des Modes *Emergency* eingeplant sind, andernfalls wurde ihnen die Priorität 0 zugeteilt. Die Nachrichten mit Mode *Stream* (Entertainment) haben jeweils die niedrigste Priorität.

Die Unterscheidung von Modes und *Modus-Präferenzen* erscheint zunächst überflüssig, erlaubt aber die saubere Trennung zwischen einem abstrakten Designkonzept (den Transmission Modes) und den anschließenden problembezogenen Detailentscheidungen[8], welche die konkrete Ablaufplanung und die Priorisierung der Nachrichten betreffen (*Modus-Präferenzen*). Diese strikte Trennung gestattet es, verschiedene Nachrichtentypen demselben Mode zuzuordnen (konzeptuelle Ebene) und diesen dennoch unterschiedliche *Modus-Präferenzen* abhängig vom Mikroslot zuzuteilen (vgl. Definition 2.5). Diese Möglichkeit kommt bei den X-by-Wire Nachrichten des Modes *Safety* zum Einsatz. Trotz identischem Mode verfügen die Nachrichten über unterschiedliche Prioritäten in den verschiedenen Mikroslots, z. B. in Slot s_1 und s_2 (Tabelle 2.6a und 2.6b). Durch die Kopplung der *Modus-Präferenzen* sowohl an die Modes als auch Slots lassen sich darüber hinaus auch weiterführende Aspekte, wie etwa Quality of Service-Anforderungen (QoS-Anforderungen), abbilden (vgl. Kapitel 2.4).

Bei Festlegung der *Modus-Präferenzen* muss berücksichtigt werden, dass ein Nachrichtentyp des Modes mit der höchsten vergebenen *Modus-Präferenz* innerhalb eines Slots die Übertragung aller anderen Nachrichten mit niedrigerer *Modus-Präferenz* (innerhalb dieses Slots) blockieren bzw. verzögern kann. In diesem Beispiel können Nachrichten des Modes *Safety* von den Nachrichten des Modes *Emergency* (in den Slots s_1 und s_3) verzögert werden. Dabei ist sowohl bei den Steer-by-Wire als auch bei den Brake-by-Wire Nachrichten sichergestellt, dass pro Makroslot garantiert jeweils eine Nachricht übertragen werden kann (Slot s_2 bzw. s_4, vgl. Tabelle 2.6a und 2.6b). Demgegenüber können die Nachrichten der Modes *Stream* und *Regular* beliebig lange verzögert werden. Da es sich bei den Nachrichten der Modes *Safety* jedoch um temporäre Nachrichtenströme und bei den Nachrichten der Modes *Emergency* und *Regular* (Sitzpositionierung und Spiegelsteuerung) um seltene sporadische Nachrichten handelt, ist eine (dauerhafte) Konkurrenz innerhalb dieser Slots nicht sehr wahrscheinlich. Falls es zu einer solchen Konkurrenz kommt, verzögert sich die Übertragung der Nachrichten mit der niedrigeren *Modus-Präferenz* entsprechend nur kurz[9]. Durch die gewählte Zuteilung der *Modus-Präferenzen* wird den Entertainment Nachrichten des Mode *Stream* jeweils die gesamte verbleibende Bandbreite des Makroslots zugeordnet.

Dadurch, dass allen sicherheitskritischen Nachrichtentypen (Airbagsteuerung, Steer-by-Wire, Brake-by-Wire) mindestens in einem Mikroslot pro Makroslot die höchste *Modus-Präferenz* zugeteilt wurde, sind

[8] Dennoch sind auch diese Entscheidungen noch Teil der Designphase und bilden z. B. die Basis eines Schedulability-Tests. An dieser Stelle werden noch keine Lowlevel-Entscheidungen getroffen, die sich direkt auf eine Kommunikationstechnologie beziehen, wie beispielsweise die Art der Abbildung der *Modus-Präferenzen* auf CAN-Identifier.

[9] Die Entscheidung, ob eine Nachricht, die aufgrund einer zu niedrigen *Modus-Präferenz* nicht übertragen wird, einen erneuten Übertragungsversuch durchläuft, ist eine Designentscheidung.

für diese Nachrichtentypen deterministische Garantien bzgl. der auftretenden Verzögerungen gegeben. Trotzdem bleibt dank der mehrfachen Zuordnung der Mikroslots die Bandbreite nicht ungenutzt, sollten diese Nachrichten nicht übertragen werden. Da der gezeigte Ablaufplan auf eine minimale Anzahl von Mikroslots abgestimmt wurde, fehlen in diesem Beispiel Aspekte, wie die Möglichkeit, Alive-Nachrichten umzusetzen (z. B. für sicherheitskritische Nachrichtentypen) oder die Berücksichtigung von QoS-Anforderungen (wie eine minimale Bandbreite für Entertainment Nachrichten), die sich ebenfalls mittels *Mode-Based Scheduling* abbilden lassen. Diese Feinheiten demonstrieren wir in einem späteren Beispiel (vgl. Kapitel 2.4.2).

Die bislang eingeführten Definitionen beziehen sich auf die Designphase und erlauben die statische Zuordnung von Slots und Modes zu Knoten sowie die Festlegung der *Modus-Präferenzen*. Zur Beschreibung der Dynamik und Konkurrenz von Rahmen zur Laufzeit wird in Analogie zur Slot-Assignment-Funktion nun die Frame-Assignment-Funktion definiert. Mit deren Hilfe kann beschrieben werden, welche Rahmen für den jeweiligen Knoten, Mode und Slot zur Übertragung bereitstehen bzw. eingeplant sind.

Definition 2.6

Sei F die Menge aller zu sendenden Rahmen, Θ die Sequenz der Makroslots, S die Menge der Mikroslots, V die Menge der Knoten eines Single-Hop-Netzwerks. Dann ist die Frame-Assignment-Funktion (kurz Frame-Assignment) $FA : \Theta \times S \times M \times V \rightarrow F$ eine partielle Funktion, die einem Tupel, bestehend aus einem Makroslot, einem Mikroslot sowie einem Mode und Knoten, einen Rahmen zuordnet.

In der Realität wird das *Frame-Assignment* dynamisch zur Laufzeit bestimmt und ist nicht apriori festgelegt, wie etwa das Slot-Assignment. So werden beispielsweise Rahmen zur Spiegelpositionierung nur für die Übertragung eingeplant, wenn der Fahrer den Schalter betätigt. Aufgrund des Umstandes, dass diesem Nachrichtentyp nicht die höchsten *Modus-Präferenzen* zugeteilt wurde, kann es vorkommen, dass Rahmen für eine spätere Übertragung erneut eingeplant werden müssen. Dies wäre z. B. der Fall, wenn zeitgleich in Slot s_2 eine Nachricht des Modes *Safety* übertragen werden muss, da diese eine höhere *Modus-Präferenz* aufweist. Das heißt, der Umstand, dass ein Rahmen für die Übertragung in einem Mikroslot bereitsteht, impliziert nicht, dass dieser auch tatsächlich in dem Slot übertragen wird.

Damit ein Rahmen in einem Mikroslot überhaupt übertragen werden darf, muss ein Slot-Assignment für den jeweiligen Knoten, Slot und Mode existieren. Das heißt, das dynamische Frame-Assignment muss mit dem Slot-Assignment verträglich sein.

Definition 2.7

Sei SA ein Slot-Assignment und FA ein Frame-Assignment. Dann ist FA verträglich mit SA genau dann wenn

$$\forall S \in \Theta, s \in S, m \in M, v \in V, f \in F.(FA(S,s,m,v) = f \Rightarrow SA(s,m) = v)$$

Die Tabelle 2.7 zeigt für unser Beispiel ein mögliches Frame-Assignment *FA* für drei aufeinanderfolgende Makroslots, welches mit dem zuvor eingeführten Slot-Assignment *SA* verträglich ist. Jeder eingeplante Rahmen ist in der Tabelle mit einem Eintrag $f_{i,j}$ dargestellt. Hierbei identifiziert der Index i den Knoten v_i als Sender; der Index j nummeriert alle eingeplanten Rahmen des Knotens fortlaufend. Ein Strich in einer Zelle kennzeichnet einen Slot, für den der jeweilige Knoten keinen Rahmen des entsprechenden Modes eingeplant hat. Wie aufgrund der hohen Bandbreitenanforderungen sowie der Isochronität der Entertainment Nachrichten nicht anders zu erwarten, versucht der Knoten v_1 in jedem Slot seine Rahmen zu übertragen, während bei den anderen Knoten nur sporadisch Nachrichten anfallen.

Mode	Knoten	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,4}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$	$s_{2,4}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$	$s_{3,4}$
Emergency	v_6	–		–		–		–		–		–	
Safety	v_5	–			–	–			$f_{5,1}$	–			–
Safety	v_4		–	–			$f_{4,1}$	–			–	–	
Regular	v_3				–				$f_{3,1}$				$f_{3,1}$
Regular	v_2		–				–				$f_{2,1}$		
Stream	v_1	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$	$f_{1,4}$	$f_{1,5}$	$f_{1,6}$	$f_{1,7}$	$f_{1,8}$	$f_{1,9}$	$f_{1,10}$	$f_{1,11}$	$f_{1,12}$

Tabelle 2.7: *Frame-Assignment*-Funktion für das Automobil-Szenario.

Die grundlegende Aufgabe des *Mode-Based Scheduling* ist es zu gewährleisten,

1. dass innerhalb eines Slots nur Knoten und deren Rahmen um den Medienzugriff konkurrieren, für die eine entsprechende Slotzuordnung in *SA* existiert und
2. dass bei mehreren konkurrierenden Rahmen derjenige Rahmen, dessen Mode die höchste *Modus-Präferenz* für diesen Mikroslot zugeteilt wurde, übertragen wird.

Auf Grundlage dieser Idee sowie den vorherigen Definitionen lässt sich auch die Funktionsweise von *Mode-Based Scheduling* formal beschreiben.

Definition 2.8

Sei $FA : \Theta \times S \times M \times V \to F$ *eine mit SA verträgliche Frame-Assignment-Funktion und* $tx \subseteq \Theta \times S \times V \times F$ *eine Relation, die definiert, ob ein Rahmen innerhalb eines Mikroslots gesendet wird oder nicht. Dann ist Mode-Based Scheduling ein Verfahren, bei dem innerhalb einer Slotinstanz jeweils der eingeplante Rahmen mit der höchsten Modus-Präferenz gesendet wird. Es gilt also:*

$$\forall S \in \Theta, s \in S, m \in M, v \in V. \exists f \in F. (FA(S,s,m,v) = f \wedge$$
$$\forall m' \in M, v' \in V, f' \in F. (m' \neq m \wedge FA(S,s,m',v') = f' \wedge mp(s,m) < mp(s,m')) \Rightarrow (S,s,v,f) \in tx)$$

Die Definition 2.8 erzwingt in Verbindung mit den vorherigen Definitionen, dass pro Mikroslot maximal ein Knoten seinen Rahmen überträgt und dass dieser Rahmen innerhalb des Slots die höchste *Modus-Präferenz* aller sendebereiten Rahmen aufweist. In der Tabelle 2.7 sind die Rahmen, die in dem jeweiligen Slots übertragen werden, grau hinterlegt. Während innerhalb des ersten dargestellten Makroslots kein Wettbewerb um die Mikroslots stattfindet, konkurrieren in den anderen Makroslots teilweise zwei bis drei Rahmen (z. B. in $s_{2,2}$ oder $s_{2,4}$) miteinander. Der Konflikt wird in diesen Fällen auf der Basis der vergebenen *Modus-Präferenzen* aufgelöst. Knoten, die den Wettbewerb um einen Mikroslot verloren haben, können entweder den jeweiligen Rahmen verwerfen – wie z. B. Knoten v_1, der einen Audiostream überträgt und daher den Rahmen $f_{1,6}$ des Slots $s_{2,2}$ verwirft – oder diese für eine spätere Übertragung erneut einplanen. Ein Beispiel hierfür findet sich in Slot $s_{2,4}$, dort wird der Rahmen $f_{3,1}$ zur Steuerung der Sitzpositionierung für die spätere Übertragung in Slot $s_{3,4}$ erneut eingeplant. Dies ist notwendig, da in Slot $s_{2,4}$ ein Rahmen mit dem Mode *Safety* Vorrang hat.

Mode-Based Scheduling selbst ist lediglich ein Planungsverfahren. Hierzu gehört die Zuordnung von Modes zu Nachrichten, die Festlegung von *Modus-Präferenzen* sowie die Zuteilung von Mikroslots zu verschiedenen Knoten in Abhängigkeit von den Modes. Innerhalb eines Slots wird jeweils die sendebereite Nachricht mit der höchsten *Modus-Präferenz* übertragen (unter Berücksichtigung des Slot-Assignment). Um dies zur Laufzeit auch tatsächlich zu realisieren, wird *Fast Mode-Signaling* benötigt.

2.3.3 Fast Mode-Signaling

Fast Mode-Signaling (*Schnelle Modussignalisierung*) ist ein Verfahren, um den Mode mit der höchsten *Modus-Präferenz* schnell, effizient und zuverlässig sowie deterministisch innerhalb des gesamten Netzwerkes zu propagieren. Schnell bedeutet in diesem Kontext, dass die verwendete Technik jeweils zum Beginn eines Slots bzw. mit minimaler Verzögerung den aktuellen Mode mit der höchsten *Modus-Präferenz* bestimmen und an alle Knoten übermitteln muss. Hierbei handelt es sich um eine zentrale Voraussetzung, um *Mode-Based Scheduling* (effizient) implementieren zu können, da in einem Mikroslot jeweils der Rahmen mit der höchsten *Modus-Präferenz* übertragen werden soll.

Bevor wir uns in den nachfolgenden Kapiteln mit der Implementierung von *Fast Mode-Signaling* befassen, geht es hier zunächst um die konzeptionelle Ebene. So lassen sich verschiedene Varianten von *Fast Mode-Signaling* in Abhängigkeit des gewählten Gültigkeitsbereichs der Modes unterscheiden: Hierbei wird zwischen *globalen* und *lokalen Modes* unterscheiden. Ist der ausgewählte Mode global[10] für einen bestimmten Zeitraum gültig, so dürfen in diesem nur Rahmen versendet werden, die diesem Mode zugeordnet sind. Dementsprechend erfolgen Übertragungen nur in den Slots, die über ein Slot-Assignment für diesen Mode verfügen. Der Wechsel eines Modes kann, je nach Umsetzung, zu Beginn eines Makroslots oder innerhalb eines Mikroslots signalisiert werden. In Abhängigkeit von der Signalisierung wechseln alle Knoten zu dem jeweiligen Mode, der dann bis zu einer erneuten Signalisierung gültig bleibt. Vorteil dieses Ansatzes ist, dass innerhalb der Mikroslots die mit dem Mode assoziierten Rahmen sofort und wettbewerbsfrei gesendet werden können. In der Regel wird bei globalen Modes der aktuelle Mode nur selten gewechselt (im Gegensatz zu lokalen Modes (s.u.)), somit wirkt sich ein Overhead bei der Signalisierung weniger stark aus. Von Nachteil ist, dass Slots ungenutzt bleiben, wenn kein Rahmen mit diesem Mode eingeplant wurde oder kein entsprechendes Slot-Assignment für die Kombination von Slot und Mode existiert. Ein Anwendungsszenario für globale Modes wäre beispielsweise das Umschalten von einem Fahr- in ein Updatemodus, welcher ein schnelles Update der Steuergeräte in der Werkstatt ermöglicht, da alle Slots für die Übertragung des Updates genutzt werden können. Das TTP/C-Protokoll unterstützt globale Modes (mit gewissen Einschränkungen), eine detaillierte Beschreibung und Abgrenzung ist in Kapitel 2.5 zu finden.

Die zweite Variante von *Fast Mode-Signaling* beschränkt die Gültigkeit der Modes auf individuelle Knoten (*lokale Modes*). Hierbei wird zunächst zu Beginn jedes Mikroslots von jedem Knoten mit Slot-Assignments lokal die Menge aller für diesen Slot eingeplanten Rahmen ermittelt (Frame-Assignment). Ist diese Menge nicht leer, bestimmt jeder Knoten innerhalb dieser Menge den Rahmen, dessen Mode die höchste *Modus-Präferenz* zugeordnet wurde[11]. Damit der Knoten diesen Rahmen tatsächlich übertragen darf, muss der lokale Mode zuerst den Wettbewerb um den Slot gewinnen. Hierzu signalisieren[12] alle Knoten, den von ihnen ermittelten lokalen Mode. *Fast Mode-Signaling* muss nun gewährleisten, dass derjenige Knoten den Wettbewerb gewinnt, dessen lokaler Mode unter allen signalisierten Modes die höchste *Modus-Präferenz* aufweist. Dieser Knoten darf im Anschluss seinen Rahmen direkt senden, ohne dass es hierbei zu Kollisionen kommt.

Wir vertreten die Auffassung, dass die Verwendung lokaler Modes – wie schon in dem Beispiel in Abschnitt 2.3.2 illustriert – ein höheres Potential in Bezug auf eine bessere Bandbreiteneffizienz bei sporadischen Nachrichten bietet. Dies gilt insbesondere, wenn die Ablaufpläne eine sehr hohe Dynamik aufweisen, die einen Wechsel des Modes in jedem Slot erfordern und nicht einfach durch den Wechsel des Systembetriebszustandes abgebildet werden können. Es ist offensichtlich, dass die konkrete Implementierung von *Fast Mode-Signaling* maßgeblich die Effizienz von *Mode-Based Scheduling with Fast Mode-Signaling* bestimmt. Eine einfache Realisierung von *Fast Mode-Signaling* über den Austausch von

[10] Global steht hier im Sinne von systemweit, d.h. heißt für alle Knoten gültig.

[11] Dieser Mode ist der lokale Mode dieses Knotens in diesem Slot.

[12] Das genaue Verfahren zur Signalisierung gehört zu den implementierungs- und technikspezifischen Aspekten der Realisierung von *Fast Mode-Signaling*.

Nachrichten zur Bestimmung des Modes führt hier nicht zu einer effizienten Lösung. Insbesondere vor dem Hintergrund, dass *Mode-Based Scheduling* den Bandbreitenbedarf reduzieren bzw. die Bandbreite effizienter nutzbar machen soll, ist die Einführung zusätzlicher Nachrichten (die zudem zu Beginn jedes Slots ausgetauscht werden müssten, ohne hierbei Kollisionen zu erzeugen) ohnehin nicht sinnvoll.

Stattdessen werden wir im Rahmen dieser Arbeit zwei unterschiedliche Konzepte zur effizienten Realisierung von *Fast Mode-Signaling* mit lokalen Modes vorstellen. Der erste Ansatz nutzt die Eigenschaften der Kodierung sowie den Arbitrierungsmechanismus von CAN. Diese Variante ist jedoch auf CAN-ähnliche oder darauf aufbauende Protokolle beschränkt, die einen solchen oder ähnlichen Arbitrierungsmechanismus anbieten. Bei dieser Form der Umsetzung fällt für die Realisierung von *Fast Mode-Signaling* kein zusätzlicher Overhead an. Die Details der Umsetzung sowie die Evaluation der entwickelten Prototypen sind in Kapitel 3 beschrieben.

Bei dem zweiten, in Kapitel 4 beschriebenen Konzept werden die *Modus-Präferenzen* durch die Verwendung von Backoffslots (unterschiedliche lange Wartezeiten vor einem Übertragungsstart) abgebildet. Dieser Ansatz ist unabhängig von der konkreten Kommunikationstechnologie und lässt sich sehr gut in bereits existierende Protokolle und Kommunikationstechnologien integrieren. Auf diese Weise wird *Mode-Based Scheduling* auch als Ergänzung und Erweiterung für bereits etablierte Protokolle interessant. Wir zeigen anhand dieser Grundlagen in den Kapiteln 5 und 6 exemplarisch die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* sowohl in drahtlose Netzwerke auf Basis IEEE 802.15.4 [IEE03] als auch in das FlexRay-Protokoll [Fle10b, Fle05a].

Wir unterscheiden in dieser Arbeit in Bezug auf die Realisierung des *Fast Mode-Signaling* zwischen passivem und aktivem *Fast Mode-Signaling*. Beim *aktiven Fast Mode-Signaling* kommunizieren die um einen Mikroslot konkurrierenden Knoten aktiv miteinander, um zu einem netzwerkweiten Konsens über den Mode mit der höchsten Präferenz zu gelangen. Ein Beispiel hierfür ist die CAN-Arbitrierung. Im Gegensatz hierzu erfolgt beim *passiven Fast Mode-Signaling* keine (aktive) Kommunikation der konkurrierenden Knoten. Ein Beispiel hierfür ist die Realisierung des *Fast Mode-Signaling* mittels Backoffslots (Wartezeiten).

2.4 Anwendung von Mode-Based Scheduling

Bevor wir uns jedoch mit den technischen Umsetzungen beschäftigen, entwickeln wir zunächst Muster, wie *Mode-Based Scheduling* bei der Erstellung von Ablaufplänen sinnvoll eingesetzt werden kann, um die Nutzung der verfügbaren Bandbreite zu verbessern sowie weitere nicht-funktionale Merkmale zu optimieren (z. B. Reduktion der mittleren Verzögerung, QoS etc.). Anschließend wenden wir die Muster an, um einen optimierten Ablaufplan für das in Kapitel 2.2 eingeführte Anwendungsbeispiel aus dem Automobilbereich zu entwerfen.

2.4.1 Muster für den Einsatz von Mode-Based Scheduling

Die folgenden Muster beschreiben typische Anwendungsfälle für *Mode-Based Scheduling*. Bei deren Beschreibung orientieren wir uns an der Idee der *Design Patterns* im Bereich der Software-Entwicklung [GHJV09]. Jede Musterbeschreibung besteht aus einem Kurztitel, einer Darstellung des Problems sowie einer Beschreibung der Lösung mit kurzer Erläuterung des Vorgehens, dem Anwendungsgebiet und einem Beispiel. Das Anwendungsgebiet beinhaltet eine Charakterisierung der Nachrichtentypen, für die das Muster primär in Frage kommt, und erlaubt eine schnelle Identifikation von möglichen Kandidaten. Das abschließende Beispiel illustriert die Anwendung des Musters und beschränkt sich auf die unbedingt notwendigen Nachrichtentypen.

Das *Muster 0*, mit dem wir beginnen, nimmt eine gewisse Sonderrolle ein, da es im Grunde lediglich die Idee von *Mode-Based Scheduling* noch einmal postuliert. Es dient primär der Vollständigkeit und zur Verbesserung der Übersicht.

Muster 0 (Seltene sporadische Nachrichten)

Problembeschreibung: Seltene sporadische Nachrichten sollen möglichst effizient, in Bezug auf den anfallenden Bandbreitenbedarf, in einen Ablaufplan integriert werden.

Lösung: Zunächst werden weitere sporadische Nachrichten identifiziert und diesen unterschiedliche Modes, in Abhängigkeit von deren Charakteristika bzw. deren Typ, zugeordnet. Die Nachrichten werden entsprechend den Kommunikationsanforderungen (vergleiche Muster 3 und 4) gruppiert, sodass pro Gruppe jeder Mode maximal einmal vorkommt[13]. Anschließend erfolgt die Zuteilung von einem oder mehreren Slots pro Makroslot zu den so gebildeten Gruppen. Die Festlegung der *Modus-Präferenzen* richtet sich nach den Kommunikationsanforderungen der Nachrichtentypen und deren Wichtigkeit.

Um die Bandbreite optimal zu nutzen, kann den jeweiligen Slots zusätzlich ein niedrig priorisierter periodischer Datenstrom[14] zugewiesen werden, sodass die Bandbreite auch dann genutzt wird, wenn keine sporadischen Nachrichten eingeplant sind. Die *Modus-Präferenz* für den periodischen Datenstrom muss kleiner gewählt werden als die *Modus-Präferenzen* aller anderen mit diesem Slot assoziierten Modes. So wird garantiert, dass die sporadischen Nachrichten trotz des periodischen Datenstrom übertragen werden, sofern eine solche Nachricht eingeplant ist.

Anwendungsgebiet: Seltene sporadische Nachrichten

Beispiel: Das Beispiel in Abschnitt 2.3.2 zeigt, wie ein entsprechendes optimiertes Slot-Assignment in Verbindung mit den zugeordneten *Modus-Präferenzen* (Tabelle 2.6a und 2.6b) aussehen kann.

Muster 1 (Abbildung von QoS-Anforderungen)

Problembeschreibung: Bei der Übertragung periodischer Datenströme (wie z. B. Sprach-, Audio- oder Videodaten) ist es häufig notwendig, bestimmte QoS-Anforderungen zu erfüllen. Diese werden in einer QoS-Spezifikation erfasst, die sowohl eine ausreichende Dienstgüte q_{base} als auch eine bevorzugte Dienstgüte q_{pref} vorgibt. Ein Mapping bildet diese Anforderungen auf die Anzahl von Rahmen ab, welche pro Makroslot übertragen werden müssen, um die QoS-Spezifikation zu erfüllen.

Lösung: Für die Nachrichtentypen des Datenstroms wird zunächst ein eigener Mode definiert. Die minimal benötigte Dienstgüte q_{base} (Bandbreite) wird durch die exklusive Zuordnung von Mikroslots für diesen Mode bereitgestellt. Um die bevorzugte Dienstgüte q_{pref} zu erreichen, können weitere Slot-Assignments für andere Mikroslots ergänzt werden. In diesen Slots wird dem Mode jeweils die niedrigste *Modus-Präferenz* zugeteilt. Allen anderen Modes, die ebenfalls über Slot-Assignments für diese Mikroslots verfügen, werden höhere *Modus-Präferenzen* zugeordnet. Hierbei handelt es sich um Modes, die vorwiegend mit (sehr) seltenen sporadischen Nachrichten assoziiert sind, sodass im Mittel die bevorzugte Dienstgüte q_{pref} erreicht wird.

Anwendungsgebiet: Periodische Datenströme mit QoS-Anforderungen

Beispiel: Wir gehen in diesem Beispiel von einem Audiostrom aus, dem der Mode *Stream* zugeordnet wurde. Die QoS-Anforderungen wurden mit $q_{\text{base}} = 2$ *Mikroslots/Makroslot* bzw. $q_{\text{pref}} = 3$ *Mikroslots/Makroslot* festgelegt. Des Weiteren werden im Mode *Emergency* nur sporadische und sehr selten Nachrichten übermittelt. Dann könnte ein Slot-Assignment, welches die QoS-Anforderungen erfüllt, wie folgt aussehen:

[13] Ein Nachrichtentyp darf hierbei auch mehreren Gruppen zugeordnet werden, sofern dies für die Erfüllung der Kommunikationsanforderungen notwendig ist.

[14] Sofern die Kommunikationsanforderungen des periodischen Datenstroms dies erlauben.

SA	s_1	s_2	s_3	s_4
Emergency	v_1		v_1	
Stream	v_2	v_2	v_2	v_2

(a) *Slot-Assignment*-Funktion

MP	s_1	s_2	s_3	s_4
Emergency	0		0	
Stream	1	0	1	0

(b) *Modus-Präferenz*-Funktion

Tabelle 2.8: Anwendungsbeispiel für Muster 1.

Offensichtlich erfüllt das Slot-Assignment die Anforderung $q_{\text{base}} = 2$ *Mikroslots/Makroslot*; die Prüfung, ob auch $q_{pref} = 3$ *Mikroslots/Makroslot* erfüllt ist, erfordert ein stochastisches Modell für die Nachrichten des Modes *Emergency*, sodass hierüber die erwartete Bandbreite des Modes *Stream* berechnet werden kann.

Muster 2 (Disjunkte Nachrichten und Datenströme)

Problembeschreibung: Mehrere verschiedene Nachrichtentypen werden von Ereignissen ausgelöst, welche sich gegenseitig ausschließen (logisch und/oder physikalisch). Dies kann ebenso für Datenströme gelten, deren Auftreten von bestimmten Vorbedingungen abhängt (beim Auto z. B., ein Werkstattmodus für Updates).

Lösung: Bestimme jeweils Mengen von disjunkten Nachrichtentypen. Hierbei handelt es sich um Nachrichtentypen, für die durch externe oder logische Bedingungen bzw. Ereignisse sichergestellt ist, dass jeweils nur maximal eine Nachricht eines einzigen Nachrichtentyps zur gleichen Zeit erzeugt bzw. für den Versand eingeplant wird. Diesen disjunkten Nachrichtentypen können die gleichen Mikroslots[15] zugeteilt werden. Hierfür müssen den Nachrichtentypen jeweils unterschiedliche Modes zugeordnet werden; die Zuordnung der *Modus-Präferenzen* hat in diesem Fall keine Auswirkungen. Um eine höhere Robustheit zu erreichen, ist es jedoch sinnvoll, kritischeren Nachrichtentypen weiterhin eine höhere *Modus-Präferenz* zuzuordnen.

Anwendungsgebiet: Periodische und sporadische Nachrichten, gekoppelt an sich gegenseitig ausschließende Bedingungen oder Ereignisse

Beispiel: Wir definieren für einen Automobilbus die Betriebsarten Fahrt- und Werkstattmodus. Der Werkstattmodus dient dazu, die Steuergeräte zu aktualisieren, in diesem Modus werden keine Sensor- oder Regelungsnachrichten ausgetauscht. In unserem Beispiel repräsentiert der Mode *Update* den Werkstattmodus und der Mode *Stream* einen isochronen Audiodatenstrom während des Fahrtmodus.

SA	s_1	s_2	s_3	s_4
Update	v_{Prog}	v_{Prog}	v_{Prog}	v_{Prog}
Stream	v_2	v_2	v_2	v_2

(a) *Slot-Assignment*-Funktion

MP	s_1	s_2	s_3	s_4
Update	0	0	0	0
Stream	1	1	1	1

(b) *Modus-Präferenz*-Funktion

Tabelle 2.9: Anwendungsbeispiel für Muster 2.

[15] Die Anzahl der reservierten Slots hängt von dem Bandbreitenbedarf im Falle des Auftretens der Nachrichten, dem minimalen Zwischenankunftsintervall sowie der Deadlines ab.

Muster 3 (Garantierte maximale Verzögerungen, Einhaltung von Deadlines)

Problembeschreibung: Es müssen Deadlines für seltene sporadische Nachrichten deterministisch garantiert werden. Da die Nachrichten jedoch nur selten übermittelt werden, soll trotzdem die Bandbreite möglichst effizient genutzt werden.

Lösung: Um eine maximale Verzögerung bzw. das Einhalten einer Deadline garantieren zu können, muss das Slot-Assignment für diesen Nachrichtentyp so gestaltet werden, dass die Nachricht garantiert übertragen wird, sofern diese eingeplant ist. Dies wird dadurch erreicht, dass dem assoziierten Mode innerhalb der zugewiesenen Mikroslots die höchste – in diesem Slot verwendete – *Modus-Präferenz* zugeordnet wird. Dann entspricht die garantierte maximale Verzögerung dem Maximum der minimalen Abstände zweier aufeinanderfolgender derartiger Slot-Assignments.

Um die Bandbreite effizient zu nutzen, wenn keine Nachricht eingeplant ist, sollte eine Kombination mit Muster 1 und 2 geprüft werden.

Anwendungsgebiet: (Seltene) Sporadische Nachrichten, Angular Events

Beispiel: In dem gezeigten Beispiel beträgt die maximale Verzögerung für die Übertragung von Rahmen des Modes *Emergency* zwei Mikroslots.

SA	s_1	s_2	s_3	s_4
Emergency		v_1		v_1
Stream	v_2	v_2	v_2	v_2

(a) *Slot-Assignment*-Funktion

MP	s_1	s_2	s_3	s_4
Emergency		0		0
Stream	0	1	1	0

(b) *Modus-Präferenz*-Funktion

Tabelle 2.10: Anwendungsbeispiel für Muster 3.

Muster 4 (Zusicherung einer durchschnittlichen Verzögerung)

Problembeschreibung: Bestimmten Arten sporadischer Nachrichten (wie z. B. Angular Events) genügt die Gewährung einer erwarteten (stochastischen) bzw. durchschnittlichen Verzögerung. Andere sporadische Nachrichtentypen benötigen zwar deterministische Garantien bzgl. der maximalen Verzögerung, die Anwendung profitiert jedoch von einer niedrigeren mittleren Verzögerung, sodass dieses Kriterium bei der Erstellung des Ablaufplanes zusätzlich berücksichtigt werden soll.

Lösung: Für die Berücksichtigung einer garantierten maximalen Verzögerung kommt zunächst Muster 3 zur Anwendung. Wird die geforderte erwartete Verzögerung durch diese Maßnahme bereits erfüllt, sind keine weiteren Slotzuordnungen mehr notwendig. Ist die Anforderung noch nicht erfüllt oder besteht keine Notwendigkeit der Garantie einer maximalen Verzögerung, so werden Slot-Assignments derart gewählt, dass sich mehrere (vorzugsweise seltene) sporadische Nachrichten (mit unterschiedlichen Modes) einen Mikroslot teilen. Die Zuteilung der *Modus-Präferenzen* erfolgt entsprechend den Anforderungen der Anwendung, kann aber auch im Hinblick auf die zu erfüllende durchschnittliche Verzögerung optimiert werden. Die Bestimmung der erwarteten Verzögerung basiert auf stochastischen Modellen der Auftrittswahrscheinlichkeiten der verschiedenen beteiligen sporadischen Nachrichten. Zusätzlich können die gewählten Slots noch mit einem periodischen Datenstrom geteilt werden, um die Bandbreite möglichst effizient zu nutzen (vgl. Muster 0 und 1). Dieser muss jedoch dann mit einer niedrigeren *Modus-Präferenz* assoziiert werden.

Anwendungsgebiet: (Seltene) Sporadische Nachrichten, Angular Events

Beispiel: Wir betrachten ein Beispiel mit den beiden Modes *Emergency* und *Safety* und nehmen an, dass entsprechende stochastische Modelle für beide Nachrichtentypen vorliegen. Auf Basis dieser stochastischen Modelle sowie des Slot-Assignments können dann Aussagen bzgl. der erwarteten Verzögerung der Nachrichten des Modes *Safety* getroffen werden.

SA	s_1	s_2	s_3	s_4
Emergency			v_1	
Safety	v_2		v_2	

(a) *Slot-Assignment*-Funktion

MP	s_1	s_2	s_3	s_4
Emergency			0	
Stream	0		1	

(b) *Modus-Präferenz*-Funktion

Tabelle 2.11: Anwendungsbeispiel für Muster 4.

Die fünf zuvor definierten Muster erheben keinen Anspruch auf Vollständigkeit, spiegeln jedoch bereits sehr viele Aspekte und Einsatzmöglichkeiten von *Mode-Based Scheduling* wieder. In der Praxis lassen sich die Muster auch miteinander kombinieren. Zum Beispiel bietet die Kombination der Muster 2 und 3 entsprechendes Potential. Die Muster 3 und 4 sowie Muster 1 sind sich sehr ähnlich in Bezug auf die vorgestellten Lösungen, beziehen sich jedoch auf unterschiedliche Anwendungsfelder und wurden daher getrennt aufgeführt.

2.4.2 Erweitertes Anwendungsbeispiel für Mode-Based Scheduling

Anhand der vorgestellten Muster entwerfen wir nun einen modusbasierten Ablaufplan für das in Kapitel 2.2 vorgestellte Anwendungsbeispiel aus dem Automobilbereich. Um unser Ergebnis mit den Ablaufplänen für zeit- bzw. ereignisgetriggerte Kommunikation vergleichen zu können, bleiben die Nachrichtentypen sowie deren Charakteristika unverändert und können der Tabelle 2.1 entnommen werden. Die Definition der Modes sowie die Zuordnung der Nachrichtentypen bleibt ebenfalls identisch (Tabelle 2.5). In Bezug auf die geforderte Bandbreite orientieren wir uns an den Angaben in Tabelle 2.3. Wobei wir, im Gegensatz zur klassischen zeitgetriggerten Lösung, je nach Nachrichtentyp entscheiden können, ob es sich um eine harte Anforderung handelt (Airbag, X-by-Wire, Spiegel- und Sitzpositionierung) oder ob es in dem jeweiligen Fall ausreichend ist, wenn die dort angegebene Bandbreite im Mittel zur Verfügung steht (Entertainment).

Bei der Benutzung einer reinen zeitgetriggerten Kommunikation konnten wir die Kommunikationsanforderungen mit einem aus 15 frei nutzbaren Mikroslots bestehenden Makroslot nicht erfüllen, da die erforderlichen Freiheitsgrade bei der Ablaufplanung fehlen. Setzen wir jedoch *Mode-Based Scheduling* ein, ist dies kein Problem. Anstatt lediglich die Anforderungen zu erfüllen, demonstrieren wir, wie durch die Anwendung von *Mode-Based Scheduling* sowohl der Bandbreitenbedarf reduziert als auch gleichzeitig das Kommunikationsverhalten optimiert und trotzdem deterministischen Garantien gewährt werden können.

Bei unserer Makroslotkonfiguration dient der Mikroslot s_0 exklusiv für die (Re-)Synchronisation.Das Slot-Assignment sowie die gewählten *Modus-Präferenzen* sind in den Tabellen 2.12 und 2.13 dargestellt.

Das verwendete Slot-Assignment ordnet jedem Mikroslot maximal zwei Modes zu. Um eine effiziente Nutzung der Bandbreite zu erlauben, werden jeweils seltene sporadische Nachrichten mit dem periodischen Datenstrom zur Übertragung von Entertainment Nachrichten (Muster 0) kombiniert. Darüber hinaus wurden allen sicherheitskritischen Nachrichtentypen (Airbagsteuerung, X-by-Wire Nachrichten), innerhalb der ihnen zugeteilten Mikroslots jeweils die höchste *Modus-Präferenz* zugeordnet. Hierdurch wird sichergestellt, dass sicherheitskritische Nachrichten sofort übertragen und nicht von anderen Nachrichten verzögert werden. Somit wird eine obere Schranke für die maximale Verzögerung garantiert (Muster 3). Zusätzlich wurden den sicherheitskritischen Funktionen jeweils ein Slot für deren exklusive Nutzung zuge-

SA	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
Emergency				v_6								v_6				
Safety		v_5	v_4			v_5	v_4			v_5	v_4			v_5	v_4	
Regular					v_2								v_3			
Stream		v_1		v_1	v_1			v_1	v_1				v_1		v_1	v_1

Tabelle 2.12: *Slot-Assignment*-Funktion für das Anwendungsbeispiel.

MP	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	s_{13}	s_{14}	s_{15}
Emergency				0								0				
Safety		3	3			3	3			3	3			3	3	
Regular					6								6			
Stream		10		10	10			10	10				10		10	10

Tabelle 2.13: *Modus-Präferenz*-Funktion für das Anwendungsbeispiel.

ordnet (in Tabelle 2.12 grau hervorgehoben). Dies erlaubt, z. B. die Realisierung von Alive-Nachrichten[16] (Heart-Beat) für den jeweiligen Dienst, ohne andere Kommunikationen zu stören.

Für die Entertainment Nachrichten des Modes *Stream* garantiert das vorgestellte Slot-Assignment eine minimale Bandbreite von 3 Mikroslots pro Makroslot. Die mittlere Bandbreite für die Entertainment Nachrichten ist jedoch deutlich höher und beträgt bis zu 8 Slots pro Makroslot, da sämtliche anderen Nachrichten – mit denen sich v_1 Mikroslots teilt – seltene sporadische Nachrichten sind (Muster 1). Die zu erwartende Bandbreite pro Makroslot des Mode *Stream* lässt sich noch weiter erhöhen, wenn man v_1 noch weitere Slots zuordnen würde, wie z. B. s_2, s_5 und s_{13}. Wird die *Modus-Präferenz* größer als 3 gewählt, so hat dies keinen Einfluss auf die dort bereits eingeplanten Nachrichten des Modes *Safety* (Muster 3).

Das Slot-Assignment in der dargestellten Form bietet ebenfalls noch Raum, um die garantierte maximale Verzögerung oder die erwartete mittlere Verzögerung für Sitz- oder Spiegelpositionierung (Mode *Regular*) zu optimieren (Muster 3, 4). Die garantierte maximale Verzögerung kann durch zusätzliche Reservierungen – etwa in den Slots s_7, s_8, s_{15} – reduziert werden[17]. Dies ging dann jedoch zu Lasten der minimal garantierten Bandbreite des Modes *Stream*. Soll stattdessen lediglich die zu erwartende mittlere Verzögerung reduziert werden, könnte das Slot-Assignment um die Slots s_1, s_2, s_5, s_6, s_{13} für den Mode *Regular* erweitert werden. Bei dieser Variante bliebe die garantierte minimale Bandbreite für Entertainment Nachrichten unverändert.

Wie in diesem einfachen Beispiels ersichtlich, bietet *Mode-Based Scheduling* sehr viele Freiheitsgrade, um die vorhandene Bandbreite effektiver und optimaler im Sinne der Anwendung zu nutzen als klassische zeit- und ereignisgetriggerte Ansätze. Neben der Möglichkeit, für seltene sporadische Nachrichten sowohl gezielt die maximal zulässige Verzögerung (mit deterministischen Garantien) zu reduzieren als auch effektiv die zu erwartenden mittleren Verzögerungen zu optimieren, lassen sich auch bandbreiteneffizient QoS-Anforderungen abbilden (vgl. Mode *Stream*). Anders als bei der rein zeitgetriggerten Lösung genügen nun 15 Mikroslots um die spezifizierten Anforderungen abzudecken. Im direkten Vergleich reduziert die modusbasierte Lösung die benötigte Bandbreite um 25% gegenüber der rein zeitgetriggerten Lösung aus Kapitel 2.2.

[16] In diesen Slots wird wahlweise die Alive-Nachricht oder eine Steueranweisung übertragen, die in diesem Fall die Alive-Nachricht ersetzt.

[17] Bei den beschriebenen Varianten wird dem Mode *Regular* jeweils die *Modus-Präferenz* 6 zugewiesen.

2.5 Stand der Technik und Abgrenzung

Mode-Based Scheduling with Fast Mode-Signaling ist ein Verfahren zur effizienteren Nutzung der Bandbreite bei zeitgetriggerter Kommunikation bei gleichzeitiger Gewährung deterministischer Garantien bzgl. auftretender Verzögerungen mit besonderen Stärken in Bezug auf seltene sporadische Nachrichten. Die Notwendigkeit deterministischer Garantien ergibt sich zwangsläufig, da zeitgetriggerte Kommunikationsprotokolle zunehmend in sicherheitskritischen Anwendungen (z. B. X-by-Wire [NSSLW05, BPS08, Rau08, WNS+05, CMDM07]) eingesetzt werden. Zu den speziell für diesen Anwendungsbereich entwickelten und etablierten Protokollen gehören beispielsweise FlexRay und TTCAN. Eine effizientere Nutzung der Bandbreite reduziert die Kosten für die Vernetzung (z. B. durch die Einsparung von Bussen) und ist erforderlich, da die Kommunikation zwischen den Komponenten stetig zunimmt.

Diese Zielsetzung ist nicht neu; in der Literatur finden sich unterschiedliche Ansätze, um die Nutzung der Bandbreite bei zeitgetriggerter Kommunikation zu verbessern. Diese Lösungen umfassen sowohl speziell entwickelte Protokolle oder Protokollerweiterungen als auch Planungsverfahren. Wir haben einige Ansätze ausgewählt, die sich mit dieser Fragestellung beschäftigen und den Stand der Technik repräsentieren, gegen den sich *Mode-Based Scheduling with Fast Mode-Signaling* behaupten bzw. abgrenzen muss. Um die Übersicht und Lesbarkeit zu verbessern, erfolgte die Unterteilung des Kapitels anhand der in den Arbeiten adressierten Kommunikationstechnologien. In Bezug auf TTCAN und FlexRay beschränken wir uns auf eine kurze Einführung und die Abgrenzung zwischen *Mode-Based Scheduling with Fast Mode-Signaling* und speziellen Planungsverfahren für diese Protokolle. Eine detaillierte Beschreibung der Protokolle sowie eine allgemeine Abgrenzung zu *Mode-Based Scheduling with Fast Mode-Signaling* wird in den Kapiteln 3 bzw. 6 gegeben.

TTCAN

TTCAN [ISO04] ist ein zeitgetriggertes Kommunikationsprotokoll von Bosch, welches auf dem etablierten CAN-Bus aufbaut, der innerhalb der Automobil- und Produktionsautomatisierung eine sehr weite Verbreitung besitzt. Ansätze, die Nutzung der Bandbreite bei TTCAN effizienter zu gestalten, die zumindest teilweise mit *Mode-Based Scheduling* vergleichbar sind, beschäftigen sich mit Algorithmen zur Erstellung optimaler (respektive zulässiger) Ablaufpläne für eine gegebene Menge von Nachrichten mit Echtzeitanforderungen. Die Arbeiten von Naughton [Nau05] und Schmidt et al. [SS07] beschreiben beispielsweise zwei Verfahren für die Erstellung solcher optimaler Ablaufpläne.

Als zeitgetriggertes Protokoll unterteilt TTCAN die Zeit zunächst in Basiszyklen, die ihrerseits in einer Systemmatrix organisiert sind. Die Basiszyklen selbst sind in Time Windows unterteilt, in denen der Zugriff entweder exklusiv (exklusive Reservierung für genau einen Knoten/Nachrichtentyp – Exclusive Time Window) oder prioritätsbasiert (alle Knoten konkurrieren – Arbitrating Time Window) erfolgt. Diese Einschränkung ist ein wesentlicher Unterschied zu *Mode-Based Scheduling* und dient als Grundannahme der Verfahren aus [Nau05, SS07]. Hieraus resultiert eine Beschränkung des betrachteten Suchraums für die von den Verfahren bestimmte optimale Lösung. Dementsprechend sind [Nau05] und [SS07] gegenüber Verbesserungen wie *Mode-Based Scheduling* sie ermöglicht blind, da diese außerhalb des betrachteten Suchraums liegen.

LIN

Ein weiterer, im Automobil-Umfeld verbreiteter zeitgetriggerter Bus ist LIN – Local Interconnect Network [LIN10, ZS08]. LIN wurde als kostengünstige Alternative zu CAN entwickelt und wird zur Realisierung einfacher Sensor-Aktor-Netzwerke genutzt. Innerhalb des Automobils kommt der LIN beispielsweise für die Steuerung von Komfortsystemen, wie Klimaanlage, Tür-, Sitz- oder Schiebedach-Elektronik, zum Einsatz [ZS08].

LIN wurde als masterbasierter Bus entworfen, bei dem der Master die Slaves anhand eines statischen, periodischen Ablaufplans durch die Übertragung einer Nachrichten-Kennung zu Beginn des Zeitslots (LIN Master-Task) pollt. Eine Nachrichtenkennung ist bei den Standardrahmen (Unconditional Frames) jeweils nur einem Knoten zugeordnet, welcher beim Empfang der Kennung seine Daten versendet (LIN Slave-Task). Die Kommunikation bei der Verwendung von Standardrahmen ist somit kollisionsfrei.

Bei den sporadischen Rahmen darf der Master innerhalb eines Slots verschiedene Nachrichtenkennungen senden. Auch hier wird, über die Zuordnung einer Nachrichtenkennung zu einer einzigen Nachricht (und Sender), die Kollisionsfreiheit sichergestellt. Zusätzlich unterstützt LIN ereignisgetriggerte Rahmen. Bei diesen wird eine Nachrichtenkennung zum Polling mehrerer Nachrichten unterschiedlicher Sender genutzt. Auch hier darf nur der Master eine Übertragung innerhalb eines solchen Slots initiieren. Bei ereignisgetriggerten Rahmen kommt es zu Kollisionen, falls mehr als ein Knoten einen auf die Kennung passenden Rahmen eingeplant hat. Im Falle einer Kollision werden die Übertragungen abgebrochen und der Master wechselt auf einen kollisionsfreien Ablaufplan und fragt die Rahmen der einzelnen Slaves über alternative, exklusiv zugewiesene Nachrichtenkennungen ab. Danach erfolgt wieder der Wechsel zu dem vorherigen Ablaufplan. Dementsprechend sind für ereignisgetriggerte Rahmen, im Gegensatz zu *Mode-Based Scheduling*, keine strengen deterministischen Garantien[18] bzgl. der auftretenden Verzögerungen möglich.

Somit unterstützt LIN selbst keine mit *Mode-Based Scheduling with Fast Mode-Signaling* vergleichbare Funktionalität, doch die technischen Voraussetzungen ermöglichen eine effiziente Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling*. Die hierfür notwendigen Modifikationen und Anpassungen, welche für die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* erforderlich sind, haben wir in [6] beschrieben.

FlexRay

FlexRay [Fle10b, Fle05a] ist ein zeitgetriggerter deterministischer und fehlertoleranter Feldbus, der speziell für die Anforderungen der Automobilindustrie entwickelt wurde. FlexRay unterteilt die Zeit in gleich lange Kommunikationszyklen. Diese bestehen aus einem verpflichtenden statischen Segment sowie einem optionalen dynamischen Segment und einem optionalen Symbol Window. Daneben verfügt jeder Kommunikationszyklus über eine Network Idle Time, die der Synchronisation der Knoten dient und in der keine Kommunikation stattfindet.

Das statische Segment selbst ist in eine feste Anzahl statischer Slots unterteilt, die jeweils exklusiv maximal einem Knoten zugeordnet werden. Im Gegensatz hierzu dient das dynamische Segment primär der Übertragung sporadischer Daten und erlaubt eine prioritätsbasierte (dynamische) Zuteilung der verfügbaren Bandbreite. Aufgrund des im dynamischen Segment eingesetzten FTDMA-Verfahrens (vgl. Kapitel 6.1.2) können deterministische Garantien bzgl. der auftretenden Verzögerungen nur für die Nachricht mit der höchsten Priorität gewährt werden. Dies liegt darin begründet, dass sich der Zeitpunkt der Übertragung von Nachrichten mit niedrigeren Prioritäten (relativ zum Start des dynamischen Segments) danach richtet, ob in diesem Zyklus bereits Nachrichten mit höheren Prioritäten gesendet wurden und wie groß diese waren. Dementsprechend wird auch der maximale Jitter mit niedrigerer Nachrichtenpriorität größer. Unter Umständen können (abhängig vom Ablaufplan) niedrig priorisierte Nachrichten gar nicht mehr in dem aktuellen Kommunikationszyklus übertragen werden. Aus diesen Gründen sind für Nachrichten mit niedrigeren Prioritäten nur eingeschränkt Garantien bzgl. der Echtzeitfähigkeit möglich. Somit ergeben sich prinzipiell für Verfahren zur Bestimmung eines zulässigen oder optimalen Ablaufplans die gleichen Einschränkungen, wie auch schon bei TTCAN. Durch den Umstand, dass der Zugriff entweder exklusiv (statisches Segment) oder prioritätsbasiert erfolgt (dynamisches Segment), wird der Suchraum für optimale Lösungen eingeschränkt. In Bezug auf optimale Planungsverfahren für FlexRay seien die

[18] Allenfalls eine sehr schlechte Worst-Case Abschätzung wäre möglich, welche die Kollision inkl. Umschalten des Ablaufplans und Polling aller Slaves berücksichtigt.

Arbeiten von Schmidt et al. [SS09b, SS09a] sowie [LGTM09] genannt, die sich mit der Bestimmung von Ablaufplänen für FlexRay beschäftigen. Die Arbeiten [SS09b] und [LGTM09] berücksichtigen nur das statische Segment, während [SS09a] ausschließlich das dynamische Segment nutzt. Schmidt et al. gehen hierbei von der Grundannahme aus, dass periodische Nachrichten [SS09b] nur im statischen Segment und sporadische Nachrichten ausschließlich im dynamischen Segment übertragen [SS09a] werden (sollen)[19].

Bei der Einplanung periodischer Nachrichten widmet sich [SS09b] sowohl dem Frame-Packing Problem als auch der Fragestellung, wie Reservierungen so gestaltet werden können, dass der Jitter minimiert wird. Das Frame-Packing Problem zielt darauf ab, mehrere Nachrichten eines Knotens so zu einem einzigen Rahmen zusammenzustellen, dass die maximale Rahmenlänge im statischen Segment möglichst effizient ausgeschöpft wird. Beide Probleme werden zunächst getrennt auf Integer Linear Programming (ILP) abgebildet und abschließend ein Ansatz präsentiert, um diese zu kombinieren.

Auch in [LGTM09] wird sowohl die Erstellung von Ablaufplänen als auch das Frame-Packing speziell in Bezug auf die von AUTOSAR [AUT14b] formulierten Einschränkungen für das statische Segment untersucht. Hierbei wird das Frame-Packing Problem auf das zweidimensionale Bin-Packing-Problem [LMV02] reduziert und sowohl eine heuristische Lösung (Greedy-Ansatz) als auch ein ILP zur optimalen Lösung des Problems entworfen. Die Autoren präsentieren eine Verbesserung des ILP unter Ausnutzung der Eigenschaften des Frame-Packing Problems sowie der von AUTOSAR formulierten Einschränkungen. Mittels *Simulated Annealing* [KGV83] wird zudem eine Methodik vorgestellt, um eine gefundene Lösung heuristisch hinsichtlich ihrer späteren Erweiterbarkeit zu optimieren.

Zusätzlich zur Betrachtung des statischen Segments untersuchen Schmidt et al. in [SS09a] auch die Ablaufplanung für das dynamische Segment. Das entwickelte Verfahren basiert wiederum auf einem ILP und nutzt minimale Auftrittsintervalle der sporadischen Nachrichten. Der so erzeugte Ablaufplan, sofern einer gefunden wurde, garantiert die Einhaltung der Deadlines. Optimierungskriterium des ILP ist die Minimierung der Ende-zu-Ende-Verzögerung[20] der Nachrichten. Gleichzeitig soll hierüber auch die Gesamtlänge des benötigten dynamischen Segments minimiert werden. Realisiert wird das Ganze über fest zugewiesene Prioritäten (dynamische Slotnummern) zu den Knoten. Der jeweilige Knoten versendet dann Nachrichten (ggf. unterschiedlichen Typs) innerhalb des ihm zugeordneten Slots, gemäß des berechneten Ablaufplans.

Auch [DMTT05] beschäftigt sich mit der Entwicklung von Ablaufplänen für FlexRay. Hier kommt ein genetischer Algorithmus zum Einsatz, um gleichzeitig die miteinander kommunizierenden Tasks ihren jeweiligen Knoten zuzuordnen als auch einen Ablaufplan für den Austausch der Nachrichten unter Nutzung des statischen Segments von FlexRay abzubilden. Zusätzlich wird ebenfalls das Frame-Packing adressiert. Aber auch in [DMTT05] wird der Suchraum durch die Einschränkung auf exklusive Reservierungen beschnitten.

FTT-CAN und TTP/C

Die Einschränkung des Suchraums resultiert aus den strengen Restriktionen, welche die bisher vorgestellten zeitgetriggerten Protokolle (entweder ausschließlich exklusiver oder prioritätsbasierter Mediumzugriff) erzwingen. Daher betrachten wir abschließend zwei konkrete Kommunikationsprotokolle, die sich diesbezüglich hervortun: FTT-CAN (Flexible time-triggered CAN) und TTP/C. FTT-CAN wurde gewählt, da dieses Protokoll, von den Autoren [FPAF02b, FAF+06], explizit mit der Zielsetzung entworfen wurde, zeit- und ereignisgetriggerte Paradigmen möglichst effizient zu kombinieren. Der entwickelte FTT-Ansatz ist als Konzept so allgemein gehalten, dass dieser auch auf andere Protokolle wie Ethernet portiert (FTT-Ethernet) werden konnte. Aus diesen Gründen wollen wir die Konzepte von FTT (insbesondere FTT-CAN) mit denen von *Mode-Based Scheduling* vergleichen.

[19] Diese Prämisse schränkt den Suchraum für mögliche Lösungen zusätzlich ein.

[20] Die Ende-zu-Ende-Verzögerung bezieht sich auf die Verzögerung beginnend mit Zeitpunkt zu dem eine Nachricht für den Versand vorliegt bis zu deren vollständigen Übertragung.

Zusätzlich betrachten wir mit TTP/C das einzige uns bekannte Protokoll aus dem Bereich der Feldbusse, welches den Begriff der Modes selbst definiert und verwendet. TTP/C unterstützt, wie *Mode-Based Scheduling*, den Wechsel zwischen verschiedenen Modes zur Laufzeit. Daher wollen wir die Unterschiede bzw. Gemeinsamkeiten beider Ansätze betrachten und die Konzepte voneinander abgrenzen.

FTT-CAN

FTT-CAN [FPAF02b, FAF+06] kombiniert zeit- und ereignisgetriggerte Kommunikationsparadigmen und baut hierbei auf dem CAN-Protokoll auf. Die Umsetzung beruht auf dem von den Autoren entwickelten FTT-Paradigma. Dessen Konzepte sind, ebenso wie *Mode-Based Scheduling with Fast Mode-Signaling*, unabhängig von einer konkreten Umsetzung und lassen sich auf Basis unterschiedlichster Protokolle realisieren (z. B. CAN [FPAF02b, FAF+06] oder Ethernet [PGAB05]). Wir werden uns im Folgenden auf die Vorstellung der Konzepte von FTT sowie deren Implementierung für CAN (FTT-CAN) konzentrieren.

Grundgedanke des FTT-Paradigmas ist es, Bandbreite nur zu belegen, sofern tatsächlich eine Übertragung stattfindet, um dadurch eine effiziente Nutzung zu erreichen. Dies ist ein wesentlicher Unterschied zu den klassischen rein zeitgetriggerten Protokollen, die auf exklusiven Reservierungen basieren. Um dies zu erreichen, setzt FTT auf einen zentralen, dynamischen Traffic-Scheduler (TS), der von einem Masterknoten ausgeführt wird und den einzelnen Nachrichtentypen ihre Slots zur Laufzeit zuweist. Dieser ermöglicht eine Adaption der Ablaufplanung zur Laufzeit und unterstützt sowohl das Hinzufügen und Entfernen von Nachrichtentypen zur Laufzeit sowie die Anpassung deren Periodendauer. Aufgabe des TS ist die Sicherstellung der Einhaltung der Nachrichtendeadlines durch die Anpassung des Ablaufplans an veränderte Anforderungen. Die Verwendung des zentralen TS ermöglicht eine Realisierung von FTT auf Basis einer zeitgetriggerten Kommunikation, ohne hierbei die Nachteile statischer exklusiver Reservierungen in Kauf nehmen zu müssen. Zusätzlich unterstützt FTT auch die (optionale) ereignisgetriggerte Kommunikation.

Das von FTT verwendete Kommunikationsmodell unterteilt die Zeit in Elementary Cycles (ECs). Jeder EC beginnt mit einem Asynchronous Window (AW) für die ereignisgetriggerte Kommunikation, an das sich ein Synchronous Window (SW) anschließt. Jeder EC wird durch die Übertragung einer speziellen Triggernachricht (TM) durch den Masterknoten gestartet, die gleichzeitig der Synchronisation der Knoten dient. Die Payload der TM enthält zusätzlich den vom TS festgelegten Ablaufplan für das SW des aktuellen EC. Ein Knoten darf nur Nachrichten innerhalb des SW übertragen, wenn diese durch den Master eingeplant und die Zuordnung in der TM bekannt gegeben wurde. Da die TM jeden EC startet, kann der Ablaufplan für jeden EC individuell festgelegt werden. Innerhalb des AW hingegen dürfen die Knoten beliebige ereignisgetriggerte Nachrichten senden.

FTT-CAN nutzt die ohnehin zur Verfügung stehende prioritätsbasierte CAN-Arbitrierung sowohl für das AW als auch das SW. Daher entspricht das AW einem langen Arbitrating Time Window von TTCAN (vgl. Kapitel 3.1). Auch für das SW wird die CAN-Arbitrierung verwendet, so legt die TM nur fest, welche Nachrichten innerhalb des SW übertragen werden. Die Reihenfolge der Übertragung der Nachrichten wird durch die CAN-Arbitrierung anhand der eindeutigen Identifier der Nachrichten zur Laufzeit festgelegt. Daher wird auch das SW von FTT-CAN durch ein langes Arbitrating Time Window umgesetzt.

FTT ermöglicht aufgrund des TS eine sehr effiziente Einplanung von Nachrichten, da der Ablaufplan für das SW für jeden EC neu festgelegt werden kann. Das verwendete Planungsverfahren kann den Anforderungen entsprechend gewählt und mittels Schedulability-Tests die Einhaltung der Deadlines geprüft werden. Die effiziente Nutzung der Bandbreite erfordert jedoch, dass spätestens zu Beginn des EC (vor dem Versenden der TM) bekannt ist, welche Knoten Nachrichten übertragen müssen. Dies gilt zwar für periodische Nachrichten, jedoch nicht in Bezug auf sporadische Nachrichten, die z. B. aufgrund von externen Ereignissen (Indeterminismus) erzeugt werden.

Dementsprechend bleibt für sporadische Nachrichten bei FTT-CAN wieder nur die Verwendung

des AW[21] mit den entsprechenden Einschränkungen in Bezug auf die Echtzeitfähigkeit. Somit liefert auch FTT, im Gegensatz zu *Mode-Based Scheduling*, keine effiziente Lösung für den Umgang mit seltenen sporadischen Nachrichten, deren Echtzeitfähigkeit garantiert werden muss. Zudem benötigt *Mode-Based Scheduling* keinen zentralen Master, da die Entscheidung darüber, welche Nachricht übertragen wird, dezentral für jeden einzelnen Slot durch *Fast Mode-Signaling* getroffen wird. Das heißt, statt auf zentralisiertem Wissen und Informationen, basiert die Entscheidung auf dem lokalen Wissen der Knoten sowie den *Modus-Präferenzen* und dem Slot-Assignment.

Ein konzeptueller Nachteil von FTT-CAN besteht darin, dass das SW nicht in feste Zeitslots[22] unterteilt ist und daher die Nachrichten innerhalb des SW einem Jitter unterliegen, dessen Ausprägung davon abhängt, welche Nachrichten mit welcher Priorität in dem jeweiligen SW eingeplant sind. In Bezug auf sicherheitskritische Anwendungen stellt der notwendige Masterknoten einen potentiellen Single-Point-of-Failure dar. Dieses Problem wird in [FPAF02a, FAF+06] jedoch durch Replacement- und Replikationsmechanismen adressiert. Dennoch handelt es sich auch hierbei um ein Problem, welches bei *Mode-Based Scheduling with Fast Mode-Signaling* nicht existiert.

TTP/C

TTP/C [TTA04, KG94, TTA05] ist ein zeitgetriggerter Feldbus mit zwei getrennten physikalischen Kanälen, für eine redundante Datenübertragung mit 2 MBit/s, 5 MBit/s oder 25 MBit/s. TTP/C unterteilt die Zeit in Cluster Cycles. Ein Cluster Cycle besteht aus einer vorgegebenen Anzahl von TDMA-Runden und wiederholt sich unendlich oft. Jede TDMA-Runde setzt sich ihrerseits aus einer festen Anzahl von Slots zusammen. Die Länge dieser Slots wird statisch festgelegt, dürfen jedoch innerhalb einer TDMA-Runde unterschiedlich lang gewählt werden. Eine TDMA-Runde legt ebenfalls das temporale Zugriffsmuster der Knoten auf die Slots fest. Hierbei gilt, dass pro TDMA-Runde jedem Knoten maximal ein Slot zugeordnet werden darf und ein Slot auch umgekehrt maximal einem Knoten zugeordnet wird. Eine Übertragung durch einen Knoten ist nur innerhalb des ihm zugeordneten Slots zulässig. Durch die exklusive Zuteilung der Slots ist die Kollisionsfreiheit gewährleistet, sodass TTP/C in der Lage ist, deterministische Garantien zu gewähren. Jede TDMA-Runde des Cluster Zyklus ist identisch aufgebaut und wahrt das definierte temporale Zugriffsmuster (Länge der Slots sowie Zuordnung der Slots zu den Knoten), lediglich die innerhalb der Slots übertragenen Nachrichtentypen sowie deren Rahmenlänge dürfen sich unterscheiden.

Neben der exklusiven Zuordnung eines Slots zu einem einzigen Knoten (in allen TDMA-Runden eines Cluster Cycle) erlaubt TTP/C auch, dass sich mehrere Knoten einen Slot in verschiedenen TDMA-Runden des Cluster Cycle teilen. Die Kollisionsfreiheit wird dadurch sichergestellt, dass ein geteilter Slot innerhalb jeder TDMA-Runde weiterhin maximal einem Knoten[23] (fest) zugeordnet ist. Die geteilten Slots erlauben eine Verbesserung der Bandbreitennutzung für Nachrichtentypen, deren Zwischenankunftsintervall größer als eine TDMA-Runde ist. Auch im Falle sporadischer Nachrichten können diese Slots verwendet werden, sofern deren Deadlines hinreichend groß sind.

TTP/C selbst definiert einen eigenen Mode-Begriff im Sinne *globaler System Operating Modes*. Die globalen Modes von TTP/C unterscheiden sich in Bezug auf Zielsetzung und Flexibilität von den lokalen Modes des *Mode-Based Scheduling*. *Global* bedeutet hier, dass der aktive Mode für alle Knoten des Clusters gilt und diese fortan nur noch Nachrichten senden dürfen, welche diesem Mode zugeordnet sind. Daher wird der jeweils aktive Mode eines Clusters auch als *Cluster Mode* bezeichnet. Wie *Mode-Based Scheduling* erlaubt auch TTP/C die Definition eigener Modes und das Umschalten zwischen diesen zur Laufzeit. Zusätzlich zu den frei definierbaren Modes gibt es immer den reservierten Mode *Startup*, der

[21] Die Einplanung der sporadischen Nachrichten in das SW widerspricht darüber hinaus auch dem Anspruch von FTT, dass innerhalb des SW alle (in der TM) einplanten Nachrichten auch tatsächlich übertragen werden.

[22] FTT-Ethernet nutzt innerhalb des SW feste Übertragungszeitpunkte, die in der TM übermittelt werden [PGAB05].

[23] Solche Knoten werden als *multiplexed nodes* bezeichnet.

während der Initialisierung und dem Start des Clusters aktiv ist. Nur zuvor bestimmte Knoten sind dazu berechtigt, einen Wechsel des *Cluster Mode* anzufordern. Hierzu signalisiert der entsprechende Knoten einen sogenannten Modus Change Request bei der Übertragung eines regulären Rahmens in einem ihm zugeteilten Slot. Das Rahmenformat reserviert hierfür im Header spezielle Bits. Ein empfangener Mode Change Request wird von allen Knoten bis zum Beginn des nächsten Cluster Cycle gespeichert, erst dann wechseln alle Knoten zu dem neuen Mode (*Deferred Mode Change*). Modus Change Requests, die später (aber noch vor dem Ende des laufenden Cluster Cycle) gesendet werden, überschreiben hierbei frühere Requests [TTA04, Kapitel 9.3.1].

Zu beachten ist ebenfalls, dass die TDMA-Runden sich in Bezug auf die Modes nur bzgl. der innerhalb der Slots übertragenen Nachrichtentypen unterscheiden dürfen, nicht jedoch in Bezug auf das definierte temporale Zugriffsmuster [TTA04, Rus01, Kapitel 9.3.1]. Somit wird die Zuordnung der Slots zu den Knoten durch verschiedene Modes bei TTP/C, im Gegensatz zu *Mode-Based Scheduling*, nicht beeinflusst. Das bedeutet, dass bei TTP/C ein reservierter, aber ungenutzter Slot nicht von einem anderen Knoten verwendet werden kann. Erschwerend kommt hinzu, dass der Wechsel von einem Mode auf einen anderen erst zu Beginn des neuen Cluster Cycle erfolgt und dann für den kompletten Cluster Cycle gilt. Zusätzlich dürfen während des gesamten Cluster Cycle nur Nachrichtentypen übertragen werden, welche zu dem *Cluster Mode* gehören. Im Gegensatz hierzu erfolgt bei *Fast Mode-Signaling* die Signalisierung des aktuellen Modes jeweils zu Beginn eines Slots und legt damit den für diesen einen Slot gültigen Mode und somit auch indirekt die Zuordnung des Slots zu einem Knoten fest. Des Weiteren ist der propagierte Mode bei *Mode-Based Scheduling* nur für diesen einen Mikroslot des Makroslots gültig.

2.6 Zusammenfassung und weiteres Vorgehen

Dieses Kapitel führt die allgemeinen Grundlagen von *Mode-Based Scheduling with Fast Mode-Signaling* ein, einem Verfahren zur Verbesserung der Bandbreitennutzung für zeitgetriggerte Kommunikationsprotokolle, dass in [KG13] vorgestellt wurde. Ausgangspunkt und Motivation sind die Probleme klassischer zeitgetriggerter Protokolle mit exklusiven Reservierungen, die Echtzeitanforderungen seltener sporadischer Nachrichten (bei gleichzeitig effizienter Bandbreitennutzung), zu erfüllen. *Mode-Based Scheduling with Fast Mode-Signaling* setzt genau hier an und bietet eine Option, die in diesen Fällen reservierten, aber häufig ungenutzten Slots für die Übertragung anderer Nachrichten nutzbar zu machen.

Nach der formalen Einführung der zentralen Bestandteile von *Mode-Based Scheduling with Fast Mode-Signaling* wurden zunächst mögliche Anwendungsfälle identifiziert, bei denen der Einsatz von *Mode-Based Scheduling* sinnvoll ist, und diese dann in Form allgemeiner Muster, nach dem Vorbild der Design Patterns in der Softwareentwicklung, beschrieben. Anschließend wurden diese auf ein Beispiel aus dem Automobilbereich (Kapitel 2.2) angewendet. Wir konnten bei dem Beispiel die Bandbreitenanforderung dank *Mode-Based Scheduling* um 25% reduzieren und dabei trotzdem weiterhin alle Anforderungen, bezogen auf die Echtzeitfähigkeit der Nachrichten, erfüllen.

In den nächsten Kapiteln konzentrieren wir uns auf die Umsetzungsmöglichkeiten. Hierbei betrachten wir insbesondere, welche Möglichkeiten es gibt, um *Mode-Based Scheduling* in bereits etablierte Kommunikationstechnologien zu integrieren. Während *Mode-Based Scheduling* als Konzept auf beliebige, zeitgetriggerte Protokolle anwendbar ist, stellt die effiziente und deterministische Realisierung von *Fast Mode-Signaling* auf Basis der technischen Eigenarten der Protokolle eine Herausforderung dar.

Im Rahmen dieser Arbeit wurden zwei grundlegend unterschiedliche Ansätze für die Realisierung von *Fast Mode-Signaling* bis zur Praxistauglichkeit entwickelt: Der erste Ansatz nutzt aktives *Fast Mode-Signaling* und verwendet CAN als Basistechnologie. Die effiziente Realisierung basiert auf der CAN-Kodierung und verwendet den CAN-Arbitrierungsmechanismus.

Der zweite Ansatz verwendet passives *Fast Mode-Signaling* und basiert auf Backoffslots. Die dort verwendeten Konzepte sind, anders als bei dem auf CAN aufbauenden Ansatz, weitestgehend unab-

hängig von einer konkreten Technologie (vgl. Kapitel 4) und können daher sehr einfach nachträglich in verschiedenste Protokolle integriert werden. Bei der konkreten Implementierung sind aber auch hier technische und protokollspezifische Eigenheiten, z. B. für die Dimensionierung der Backoffslots, zu berücksichtigen, um eine zuverlässige und deterministische Funktionsweise zu gewährleisten. Anhand zweier Umsetzungen für unseren, auf IEEE 802.15.4 basierenden Protokollstack BiPS (Kapitel 5) sowie das FlexRay-Protokoll (Kapitel 6) demonstrieren wir die Anpassung der allgemeinen Konzepte an konkrete Kommunikationstechnologien.

3 Mode-Based Scheduling with Fast Mode-Signaling für TTCAN

Nach der theoretischen Einführung in die Grundlagen von *Mode-Based Scheduling with Fast Mode-Signaling*, widmen wir uns nun einer möglichen Realisierung auf Basis des Time-Triggered Controller Area Network (TTCAN) [ISO04].

TTCAN basiert auf dem 1983 von Bosch entwickelten CAN-Protokoll. Die erste öffentliche Vorstellung von CAN erfolgte 1986, 1991 publizierte Bosch dann die CAN Specification 2.0 [CiA14a, Rob91]. CAN wurde primär für die Vernetzung von Steuergeräten innerhalb von Automobilen entwickelt, um die bis dahin vorhandenen Kabelbäume zu reduzieren und langfristig abzulösen. Hierdurch konnten sowohl die Kosten als auch das Gewicht erheblich reduziert werden. Gleichzeitig schafft die flexible Vernetzung der Komponenten (Sensoren, Aktoren, Steuergeräte) erst die Voraussetzung für moderne Assistenz- und Sicherheitssysteme. Heute kommen CAN-Busse in unterschiedlichsten Anwendungsbereichen wie der Automatisierung sowie im Umfeld medizinischer Geräte zum Einsatz [CiA14b].

TTCAN wurde 2005 standardisiert und erweitert das CAN-Protokoll um einen zeitgetriggerten Operationsmodus, welcher die Gewährung deterministischer Garantien durch die Einführung von TDMA ermöglicht. TTCAN nutzt zur Übertragung von Rahmen das CAN-Protokoll, d.h. sowohl das Rahmenformat als auch die physikalische Schicht wurden unverändert übernommen. Dieser Umstand erlaubt eine effiziente Realisierung von *Fast Mode-Signaling* durch die Nutzung des CAN-Arbitrierungsmechanismus.

Dieses Kapitel ist wie folgt strukturiert: Kapitel 3.1 beschreibt den aktuellen Stand der Technik in Bezug auf das CAN- und TTCAN-Protokoll, während Kapitel 3.2 sich der Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* für TTCAN widmet. In diesem Rahmen werden die notwendigen Voraussetzungen für eine zuverlässige Umsetzung von *Fast Mode-Signaling* untersucht. Aufbauend auf diesen Ergebnissen wurde ein Prototyp implementiert und evaluiert (Kapitel 3.3). Um die Robustheit von *Fast Mode-Signaling* gegenüber Synchronisationsungenauigkeiten zu verbessern, haben wir eine funktionale Erweiterung für CAN-Controller entwickelt, welche eine deterministische Realisierung von *Fast Mode-Signaling* ermöglicht (Kapitel 3.4). Mit dem von uns auf Basis eines FPGAs (Field Programmable Gate Array) entwickelten TTCAN-Controllers (Kapitel 3.4.3) erfolgte dann die Evaluation dieser Erweiterung (Kapitel 3.4.4). Kapitel 3.5 bewertet die Robustheit von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN*, unter Verwendung der zuvor entwickelten Techniken, während Abschnitt 3.6 sich mit CAN FD (*CAN Flexible Datarate*) beschäftigt – einer sich aktuell in der Entwicklung befindlichen schnelleren Variante von CAN – und wie diese im Kontext von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* zu bewerten ist.

Ein Teil der Ergebnisse dieses Kapitels wurde in [8, 9] publiziert.

3.1 Stand der Technik von CAN und TTCAN

Da das 2004 veröffentlichte TTCAN-Protokoll die physikalische Schicht sowie die MAC-Schicht des CAN-Protokolls unverändert nutzt [HMFH02], konzentrieren wir uns zunächst auf die Grundlagen von CAN und widmen uns danach den Erweiterungen durch TTCAN. Wir beschränken uns bei dieser Einführung auf die für die Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* relevanten Aspekte beider Protokolle. Weitere Informationen können den entsprechenden Standards entnommen werden. Für das CAN Protokoll ist dies der ISO 11898-1 [ISO03a] und ISO 11898-2 Standard [ISO03b],

welcher im Wesentlichen der CAN Specification 2.0B entspricht [Rob91]. TTCAN wurde als Erweiterung von CAN im Rahmen des ISO 11898-4 Standards [ISO04] beschrieben und 2005 veröffentlicht.

3.1.1 Grundlagen von CAN

CAN ist ein nachrichtenbasierter, multimasterfähiger, ereignisgetriggerter Feldbus. Nachrichtenbasiert bedeutet, dass jeder CAN-Nachricht ein eindeutiger Identifier zugeordnet wird und darüber hinaus keine direkte Adressierung des oder der Empfänger erfolgt. Der Identifier repräsentiert gleichzeitig die Priorität der Nachricht (ein niedrigerer Identifierwert steht für eine höhere Priorität). Er besitzt eine feste Länge von 11 bzw. 29 Bits (bei erweitertem Rahmenformat) und schließt sich direkt dem Start-of-Frame-Bit (SOF) an, mit dem ein Rahmen beginnt (Abbildung 3.1). Die Kodierung des Identifier beginnt mit dem Most-Significant-Bit. Das *Arbitrierungsfeld* besteht neben dem Identifier aus dem Remote-Transmission-Bit, welches zur Unterscheidung zwischen Datenrahmen und Datenanforderungsrahmen (vgl. [Rob91]) dient. Das Steuerfeld enthält die Länge der Payload des Datenrahmens oder bei einem Datenanforderungsrahmen, die Länge der Payload des angeforderten Datenrahmens sowie zusätzliche reservierte Steuerbits. Hierauf folgt bei einem Datenrahmen das Datenfeld. Zusätzlich beinhaltet jeder Rahmen eine Prüfsumme (15 Bit Cyclic Redundancy Check) und ein Bestätigungsfeld (Acknowledge-Feld), in dem andere Knoten den fehlerfreien Empfang eines Rahmens quittieren. Beendet wird der Rahmen mit einem 7 Bit langen End-of-Frame Feld (EOF), welches ausschließlich aus rezessiven Bits besteht. Zwei aufeinander folgende Rahmen werden durch einen Interframe-Space (IFS) getrennt, welcher aus mindestens drei rezessiven Bits besteht. Die Abbildung 3.1 zeigt einen CAN-Rahmen mit einem 11 Bit Identifier. Das erweiterte Rahmenformat verwendet 29 Bits zur Codierung des Identifier und nutzt ein reserviertes Bit des Steuerfeldes, um zwischen den Formaten zu unterscheiden.

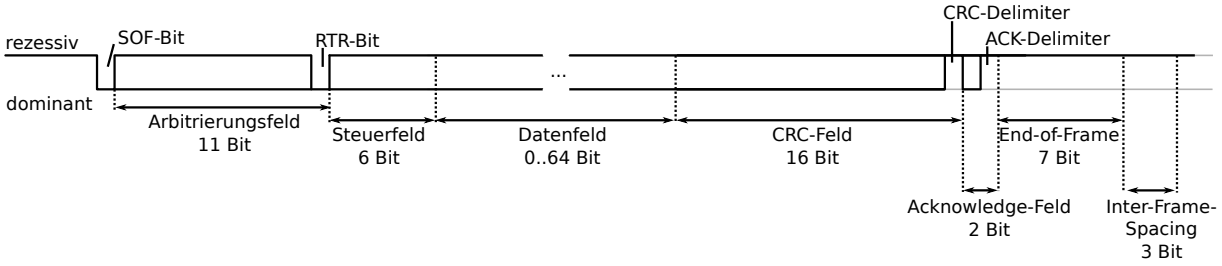


Abbildung 3.1: Format eines CAN-Datenrahmens [Ets01, Abbildung 2.2-1]

CAN verwendet als Zugriffsverfahren CSMA/CR (Carrier Sense Multiple Access with Collision Resolution), um ein kollisionsgeschütztes, prioritätsbasiertes Arbitrierungsverfahren zu implementieren. Hierzu werden die Bits eines CAN-Rahmens mittels dominanter und rezessiver Pegel kodiert. Ein dominanter Pegel repräsentiert ein Bit mit dem Wert 0, ein rezessiver Pegel ein Bit mit dem Wert 1. Durch eine geeignete Abbildung auf physikalischer Ebene wird erreicht, dass ein aufgeschalteter dominanter Pegel einen angelegten rezessiven Pegel auf dem Bus überschreibt.

Vor einer neuen Übertragung prüft ein Knoten zunächst, ob das Medium frei ist (CSMA). Ist dies der Fall, beginnt der Knoten unmittelbar mit der Übertragung des Rahmens, ansonsten wird bis zum Ende der laufenden Übertragung gewartet. Zu Beginn einer Übertragung sendet ein Knoten zunächst das dominante SOF-Bit und beginnt anschließend mit der Übertragung des Identifier. Mittels Bitmonitoring wird geprüft, ob das von dem Knoten gesendete Bit auch tatsächlich am Bus anliegt. Trifft dies nicht zu, bricht er entweder seine Übertragung ab (nur innerhalb des Arbitierungsfeldes) oder sendet einen Fehlerrahmen.

Starten mehrere Knoten gleichzeitig oder annähernd gleichzeitig eine Übertragung überlagern sich die gesendeten Bits der Knoten, beginnend mit dem SOF-Bit. Insbesondere überlagern sich auch die Bits des Arbitrierungsfeldes. Mittels Bitmonitoring prüft jeder Knoten während der laufenden Übertragung, ob das

gesendete Bit mit dem am Bus anliegenden Bit übereinstimmt. Ist dies der Fall, fährt der Knoten mit der Übertragung fort und sendet das nächste Bit. Wurde von dem Knoten innerhalb des *Arbitrierungsfeldes* ein rezessives Bit gesendet, auf dem Bus liegt jedoch ein dominanter Pegel an, so beendet er seine Übertragung, da ein Knoten mit einer höheren Priorität (und somit niedrigerem Identifier) ebenfalls an dieser Arbitrierung teilnimmt[1]. Hierdurch garantiert der CAN-Arbitrierungsmechanismus, dass derjenige Knoten dessen Rahmen die höchste Priorität besitzt, die Arbitrierung gewinnt [Rob91]. Bei der Vergabe der Identifier muss sichergestellt werden, dass ein Datenrahmen mit einem bestimmten Identifier jeweils nur von einem einzigen Knoten gesendet wird. Dies gewährleistet, dass die Arbitrierung von genau einem Knoten gewonnen wird. Eine Ausnahme stellen Datenanforderungsrahmen dar. Diese dürfen von mehreren Knoten (auch gleichzeitig) gesendet werden. Wenn mehrere Knoten gleichzeitig einen Datenanforderungsrahmen mit demselben Identifier versenden, scheidet kein Knoten bei der Arbitrierung aus und alle Knoten senden den Datenanforderungsrahmen gemeinsam. Da die gesendeten Datenanforderungsrahmen identisch sind (ein Anforderungsrahmen enthält keine Nutzdaten), resultieren hieraus keine verfälschenden Bitkollisionen und es ergeben sich keine Einschränkungen bzw. Störungen.

Die Abbildung 3.2 zeigt die Arbitrierung anhand eines Beispiels bei dem zwei Knoten v_1 und v_2 gleichzeitig mit der Übertragung ihres Rahmens beginnen und somit um den Zugriff auf das Medium konkurrieren. Knoten v_1 beabsichtigt die Übertragung eines Datenrahmens mit dem Identifier 1434, während Knoten v_2 einen Datenanforderungsrahmen mit dem Identifier 1456 senden möchte. Die Abbildung zeigt die von den Knoten gesendeten Signalpegel sowie den überlagerten Signalpegel auf dem Bus. Zum Zeitpunkt t_1 vergleicht Knoten v_2 den Wert des von ihm gesendeten Bits mit dem an dem Bus anliegenden Pegel und stellt fest, dass diese nicht übereinstimmen. Da Knoten v_2 ein rezessives Bit gesendet hat, jedoch am Bus ein dominantes Bit anliegt, stellt v_2 seine Übertragung ein. Infolgedessen gewinnt der Knoten v_1 die Arbitrierung und kann somit seine Übertragung fortsetzen.

Übertragungsfehler können entweder durch den Sender aufgrund des Bitmonitoring oder durch die Empfänger detektiert werden (Verletzung des Rahmenformats oder fehlerhafte CRC-Summe). Sobald ein Knoten einen Fehler erkannt hat, beginnt dieser sofort mit der Übertragung eines Fehlerrahmens. Da ein Fehlerrahmen das zulässige Rahmenformat verletzt (genauer die Bitstuffing-Regel (s.u.)) führt dies dazu, dass auch alle anderen Knoten ihrerseits einen Fehlerrahmen senden, sodass diese sich auf dem Bus überlagern. Um zu vermeiden, dass ein fehlerhafter Knoten die Kommunikation dauerhaft stört werden Fehlerzähler eingesetzt. Überschreiten die Wert der Fehlerzähler eines Knotens vorgegebene

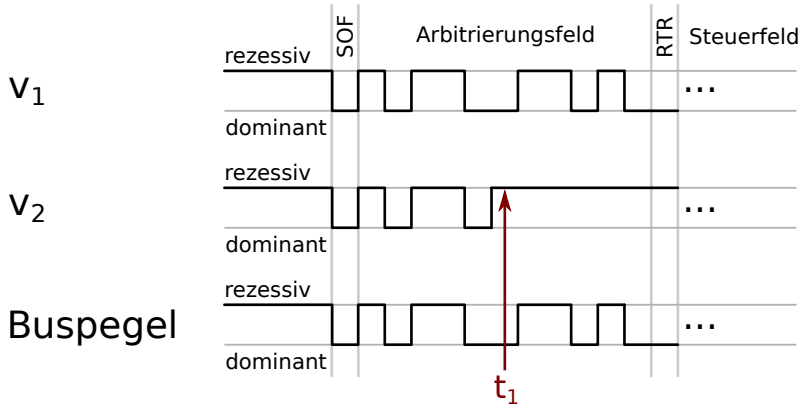


Abbildung 3.2: Beispiel für den Ablauf des CAN-Arbitrierungsmechanismus (vgl. [Ets01]).

[1] Ist ein Übertragungsfehler für den falschen interpretierten Signalpegel verantwortlich, wird der Fehler im weiteren Verlauf erkannt und signalisiert (entweder aufgrund der Verletzung des Bit-Stuffings oder bei der CRC-Prüfung). Sendet der Knoten selbst ein dominantes Bit, empfängt jedoch ein rezessives Bit, erkennt der Knoten sofort, dass es sich um einen Übertragungsfehler handelt. In diesem Fall erfolgt die Fehlersignalisierung sofort.

Schwellenwerte, so wechselt dieser zunächst in den fehlerpassiven Zustand und bei anhaltenden Fehlern in den Zustand bus-off. In diesem Zustand ist er von einer weiteren Teilnahme an der Kommunikation ausgeschlossen (vgl. [Ets01, Rob91]).

Die Kodierung auf dem Physical-Layer verwendet Non-Return-to-Zero (NRZ). Da bei dieser Kodierung der Wechsel des am Bus anliegenden Signalpegels nur von den zu kodierenden Bits abhängt, verwendet CAN Bitstuffing. Beim Bitstuffing wird nach fünf identischen Bits jeweils ein komplementäres Bit in den Datenstrom eingefügt. Hierdurch werden regelmäßige Flanken innerhalb des Bitstroms erzwungen. Diese dienen der Resynchronisation, um eine hinreichend gleichzeitige Abtastung und Interpretation des am Bus anliegenden Signalpegels durch die Knoten zu gewährleisten. Das Bitstuffing erstreckt sich bei Datenrahmen und Datenanforderungsrahmen vom SOF-Bit bis zum CRC-Feld (inklusive), ausgenommen sind spezielle Überlast- und Fehlerrahmen. Die regelmäßige Resynchronisation der Empfänger mit dem Sender ist notwendig, um den Clock Skew (Gangunterschied) zwischen den Taktgebern zu kompensieren.

Bei CAN wird, wie in Abbildung 3.3 dargestellt, eine Bitzeit in vier Segmente unterteilt: Das *Synchronization Segment*, das *Propagation Time Segment* und die Phasenpuffer Segmente *Phase Segment 1* und *Phase Segment 2* [HB99, Rob91]. Die Länge der Segmente kann für jeden Knoten – durch die Festlegung der Anzahl der Zeitquanten (Timequanta), aus denen das Segment besteht – individuell konfiguriert werden. Ein Zeitquantum stellt die kleinste Zeiteinheit eines Knotens im Zusammenhang mit CAN dar und wird aus dem Taktgeber des Knotens abgeleitet. Im Folgenden werden wir die konfigurierte Länge eines solchen Zeitquantums des Knotens v mit d_q^v bezeichnen.

Das Medium wird durch den CAN-Controller jeweils nur zu Beginn eines Zeitquantums abgetastet. Eine Flanke wird dadurch erkannt, dass sich der aktuell abgetastete Pegel von dem beim letzten Samplepoint ermittelten Buspegel unterscheidet. Als fallende Flanke wird ein Wechsel von einem rezessiven zu einem dominanten Pegel bezeichnet. Ein Wechsel von einem rezessiven zu einem dominanten Pegel entspricht einer steigenden Flanke. Diese Form der diskreten Flankenerkennung filtert sehr kurze Störimpulse aus dem Signal, die z. B. aufgrund von elektromagnetische Einstrahlungen auftreten können.

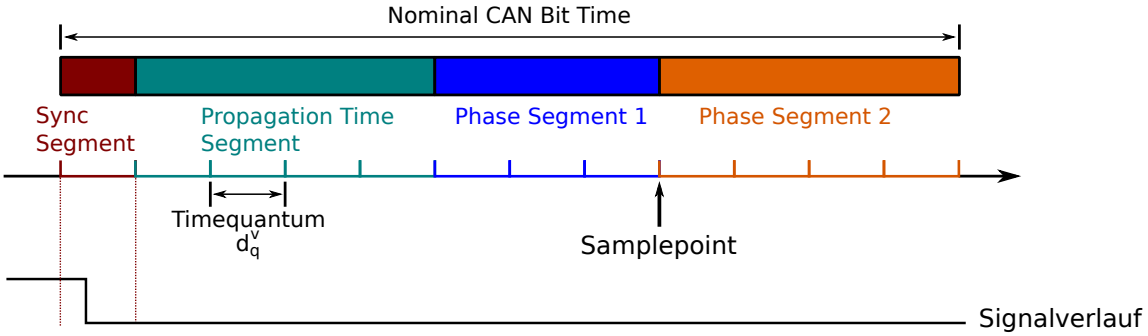


Abbildung 3.3: Die verschiedenen Segmente des CAN Bit-Timing nach [HB99, Figure 1].

Beim Empfang eines Rahmens wird von den Empfangsknoten erwartet, dass die Flankenwechsel der übertragenen Bits jeweils innerhalb des Synchronisationssegments erfolgen, wie in Abbildung 3.3 beispielhaft dargestellt. Um dies zu erreichen, führt jeder Knoten, der ein SOF-Bit auf dem Medium erkennt (d.h. eine fallende Flanke innerhalb einer Zeitspanne, in der der Bus als unbelegt erkannt wurde), einmalig für diese Rahmenübertragung eine *harte Synchronisation* durch.

Definition 3.1 (Harte Synchronisation)

Eine harte Synchronisation startet die interne Bitzeit mit dem Ende des Synchronisationssegments neu, sobald ein Knoten[2] eine fallende Flanke detektiert während das Medium frei ist. Dieser Vorgang erzwingt, dass die die harte Synchronisation auslösende Flanke innerhalb des Synchronisationssegments liegt [HB99]. Pro Rahmen wird nur eine harte Synchronisation durchgeführt.

Der Abtastzeitpunkt (Samplepoint) definiert innerhalb einer Bitzeit den Zeitpunkt zu dem der am Bus anliegende Pegel (dominant oder rezessiv) als Wert des in dieser Bitzeit übertragenen Bits interpretiert wird [HB99, Rob99, Mic12a]. Der Samplepoint liegt zwischen *Phase Segment 1* und *Phase Segment 2* und ist an den Grenzen der Zeitquanten ausgerichtet. Die Interpretation des Buszustandes zum Samplepoint führt dazu, dass ein Knoten trotz der Detektion einer fallenden Flanke, durch ein potentielles SOF-Bit und der hierdurch ausgelösten harten Synchronisation, noch bis zum Erreichen seines Samplepoint eine eigene Übertragung starten kann. Allerdings kann es durchaus implementierungsabhängig sein, zu welchen Zeitpunkten ein CAN-Controller den Wechsel vom Empfänger zum Sender nach einer erfolgten harten Synchronisation noch gestattet, da das Auslösen einer Übertragung durch einen externen Befehl an den CAN-Controller angestoßen wird. Die CAN-Spezifikation macht keine Aussagen über eine solche Schnittstelle und deren Verzögerungen bzw. Latenzen. Wir gehen im Folgenden von der Annahme aus, dass ein Wechsel von der Rolle des Empfängers in die des Senders bis zum Erreichen des Samplepoints funktioniert.

Trotz der zu Beginn der Übertragung durchgeführten harten Synchronisation wird hierdurch keine perfekte Synchronisation der Knoten des CAN-Busses erreicht, da diese die fallende Flanke des SOF-Bits (aufgrund von Signalverzögerungen sowie den nicht aufeinander ausgerichteten Zeitquanten) nicht zeitgleich detektieren. Und somit ist auch deren Bit-Timing, welches durch diese Flanke gestartet wird, zueinander verschoben. Konkurrieren mehrere Knoten um den Zugriff auf den Bus, so muss trotzdem sichergestellt werden, dass sich die gesendeten Bits des Arbitrierungsfeldes bei allen Knoten des Busses korrekt überlagern, bevor diese den Signalpegel interpretieren (d.h. ihren Samplepoint erreichen). Deshalb muss das Propagation Time Segment mindestens doppelt so groß gewählt werden wie die maximale Signalverzögerung innerhalb des gesamten Busses [HB99, Mic12a].

Constraint 3.1 (Länge des Propagation Time Segment)

Sei V die Menge der an einen CAN-Bus angeschlossenen Knoten. Des Weiteren sei $delay_{max} : V \times V \to \mathbb{R}$ eine Funktion (vgl. Definition 3.3), welche für zwei Knoten deren maximale Ende-zu-Ende Signalverzögerung liefert. Sei $v \in V$ ein beliebiger Knoten mit Zeitquanten der Länge d_q^v. Dann muss für die Länge des Propagation Time Segments $d_{PropSeg}^v$ von Knoten $v \in V$ gelten:

$$\forall v \in V. \exists i \in \mathbb{N}.\ d_{PropSeg}^v = i \cdot d_q^v \geq 2 \cdot \max_{x \in V, y \in V} (delay_{max}(x,y))$$

Die maximale Signalverzögerung $delay_{max}(x,y)$ gemäß Definition 3.3 beinhaltet sowohl die reine Übertragungsverzögerung des Signals über das physikalische Medium als auch Verzögerungen für die Signalverarbeitung innerhalb der CAN-Transceiver sowie des CAN-Controllers (s.u.).

Aufgrund des Clock Skew der Taktgeber der einzelnen Knoten zueinander kann es passieren, dass die Flanken mit Voranschreiten der Zeit (trotz der initialen harten Synchronisation) nicht mehr innerhalb des Synchronisationssegments empfangen werden. Damit aus dieser Verschiebung keine Abtastfehler resultieren, müssen diese Abweichungen durch den Empfänger mittels Bit-Resynchronisation(en) kompensiert werden. Zu diesem Zweck können die Knoten ihre beiden Phasensegmente *Phase Segment 1* und *Phase*

[2] Der Knoten ist zum Zeitpunkt des Empfangs der fallenden Flanke ein reiner Empfänger und hat (noch) nicht selbst mit der Übertragung eines dominanten Bits als Sender begonnen – da ein Sender seine Bitzeit beim Beginn der Übertragung seines eigenen SOF-Bits startet [Rob99, Mic12a].

Segment 2 verlängern bzw. verkürzen, um Gangungenauigkeiten auszugleichen. Hierbei legt der Konfigurationsparameter *Segment Jump Width* (SJW) fest, um wie viele Zeitquanten das *Phase Segment 1* maximal verlängert bzw. das *Phase Segment 2* maximal verkürzt werden darf. Eine solche Bit-Resynchronisation wird nur durch eine fallende Flanke innerhalb einer Rahmenübertragung ausgelöst [Rob91, HB99]. Eine Signalflanke heißt synchron, wenn diese innerhalb des Synchronisationssegments auftritt. Ist dies nicht der Fall, bezeichnet man die Distanz zwischen dem Auftreten der Flanke und dem zu dieser Flanke gehörenden Synchronisationssegment als Phasenfehler. Ziel der Bit-Resynchronisation ist es, den Phasenfehler zu korrigieren oder diesen zumindest zu reduzieren. Die Definition 3.2 erläutert den Bit-Resynchronisation-Mechanismus und basiert auf den Beschreibungen aus [HB99, Rob91], unter Berücksichtigung der Dokumentation des VHDL Reference CAN User's Manual [Rob99][3].

Definition 3.2 (Bit-Resynchronisation, *Late Edge*, *Early Edge*)

Die Bit-Resynchronisation führt während einer laufenden Übertragung zur Verlängerung des Phase Segment 1 oder zur Verkürzung des Phase Segment 2 mit dem Ziel, den Phasenfehler einer nicht synchronen fallenden Signalflanke durch die Verschiebung des zu dieser Flanke gehörenden, nachfolgenden Samplepoints zu kompensieren. Im Folgenden bezeichne d_{SJW} die konfigurierte Segment Jump Width des Knotens $v \in V$. Der ermittelte Phasenfehler der Signalflanke wird mit $d^v_{phaseError}$ bezeichnet und wie d^v_{SJW} in Zeitquanten angegeben.

- *Wenn die fallende Signalflanke nach dem Synchronisationssegment, aber vor dem Samplepoint detektiert wird, so wird diese Flanke als Late Edge bezeichnet. Ist der Knoten v nicht selbst ein Sender, so verlängert v sein Phase Segment 1 um* $\min(d^v_{phaseError}, d^v_{SJW})$ *Zeitquanten [Rob99].*
- *Wird eine fallende Signalflanke zwischen Samplepoint und dem Synchronisationssegment detektiert, so wird diese als Early Edge bezeichnet. In diesem Fall verkürzt v sein Phase Segment 2 um* $\min(d^v_{phaseError}, d^v_{SJW})$ *Zeitquanten unabhängig davon, ob der Knoten v gleichzeitig Sender ist oder nicht.*

Zusätzlich gilt, dass zwischen zwei Samplepoints jeweils nur eine Bit-Resynchronisation erfolgen darf. Während der Übertragung des SOF-Bits erfolgt keine Bit-Resynchronisation, da diese bereits mit einer harten Synchronisation beginnt.

Durch die Bit-Resynchronisation wird erreicht, dass ein Phasenfehler, welcher kleiner als die konfigurierte SJW ist, vollständig kompensiert wird. In diesem Fall entsprechen die Resultate der Bit-Resynchronisation denen einer harten Synchronisation. Die Einschränkung, dass zwischen zwei aufeinanderfolgenden Samplepoints nur eine Bit-Resynchronisation erfolgen darf, vermeidet wiederholte Bit-Resynchronisationen innerhalb einer Bitzeit aufgrund einer Folge kurzer Störimpulse. Die Beschränkung der Korrektur von *Late Edges* auf reine Empfangsknoten ist notwendig, da ein Sender seine eigene gesendete fallende Flanke durch Verzögerungen des CAN-Transceivers, als *Late Edges* detektieren würde (*loop-delay*, s.u.). Eine Bit-Resynchronisation würde in diesem Fall dazu führen, dass der Sender seinen Pegel länger als eine Bitzeit an den Bus anlegt.

Häufiger als der Clock Skew, führt jedoch die Konkurrenz mehrerer Knoten um den Medienzugriff zu Bit-Resynchronisationen während der Arbitrierung [HB99]. Dies liegt darin begründet, dass sich zunächst jeder Knoten hart auf den Sender synchronisiert dessen fallende Flanke des SOF-Bits diesen zuerst erreicht. Hierbei kann es sich um den Knoten handeln, der seine Übertragung zuerst startet, aufgrund von unterschiedlichen Signalverzögerungen und bei nahezu gleichzeitigem Übertragungsstart mehrerer Knoten, muss dies jedoch nicht immer der Fall sein. Insbesondere können sich verschiedene Knoten auch auf unterschiedliche Sender synchronisieren. Scheidet ein solcher Sender bei der Arbitrierung aus müssen sich die Knoten, welche sich auf diesen Sender synchronisiert haben, auf einen anderen Sender abstimmen.

[3] Hierbei handelt es sich um das Nutzerhandbuch einer Referenzimplementierung von Bosch, die Einblick in Detaillösungen und Implementierungsdetails sowie Testfälle bietet.

Dies gilt ebenso für das Bestätigungsfeld. Auch hier senden verschiedene Knoten (das ACK-Bit), die aufgrund von Signalverzögerungen und Clock Skew nicht perfekt miteinander synchronisiert sind.

Auf physikalischer Schicht erfolgt die Anbindung eines Knotens (z. B. realisiert durch einen Mikrocontroller) an den CAN-Bus durch einen CAN-Controller, welcher die Protokolllogik implementiert. Der CAN-Controller ist mit einem CAN-Transceiver (Bustreiber) verbunden, welcher ein differentielles elektrisches Signal für die Übertragung über die Busleitung erzeugt bzw. das differentielle Signal wieder in ein digitales Signal für den CAN-Controller umwandelt. Hierbei unterscheidet der CAN-Standard zwischen Low-Speed-CAN (ISO 11898-3) mit Datenraten von ≤ 125 kBit/s und High-Speed-CAN mit einer Datenrate ab 250 kBit/s bis zu 1 MBit/s (ISO 11898-2) [ZS08]. Beide Standards verwenden unterschiedliche Signalniveaus für die Codierung dominanter und rezessiver Bits und sind daher elektrisch nicht kompatibel zueinander und benötigen unterschiedliche CAN-Transceiver.

Die Abbildung 3.4 zeigt unterschiedliche Ausprägungen einer Anbindung eines Knotens an einen CAN-Bus. Knoten v_1 verwendet einen in den Mikrocontroller integrierten CAN-Controller mit einem externen CAN-Transceiver. Bei Knoten v_2 hingegen ist der CAN-Controller nicht direkt in den Mikrocontroller integriert, während bei Knoten $v3$ neben dem CAN-Controller auch der CAN-Transceiver [NXP13] integriert wurde. Diese Variante ist in der Praxis jedoch nicht sehr gebräuchlich. Um Reflektionen zu vermeiden, wird der CAN-Bus (wie in der Abbildung gezeigt) mittels zweier 120 Ω Widerstände terminiert. In rot bzw. blau sind in der Abbildung zusätzlich die Signalpfade für eine Übertragung eines Rahmens von Knoten v_1 zu Knoten v_4 eingezeichnet. Der in rot markierte Pfad wird von Knoten v_1 für das Bit-Monitoring benötigt, um den gesendeten Pegel mit dem tatsächlich am Bus anliegenden Pegel vergleichen zu können. Die sich auf diesem Pfad ergebende Verzögerung, die hauptsächlich durch den CAN-Transceiver geprägt ist, wird als *loop-delay* bezeichnet. Natürlich empfangen auch Knoten v_2 und v_3 den von v_1 gesendeten Rahmen. Aus Gründen der Übersichtlichkeit sind die entsprechenden Signalpfade jedoch nicht in der Grafik eingezeichnet.

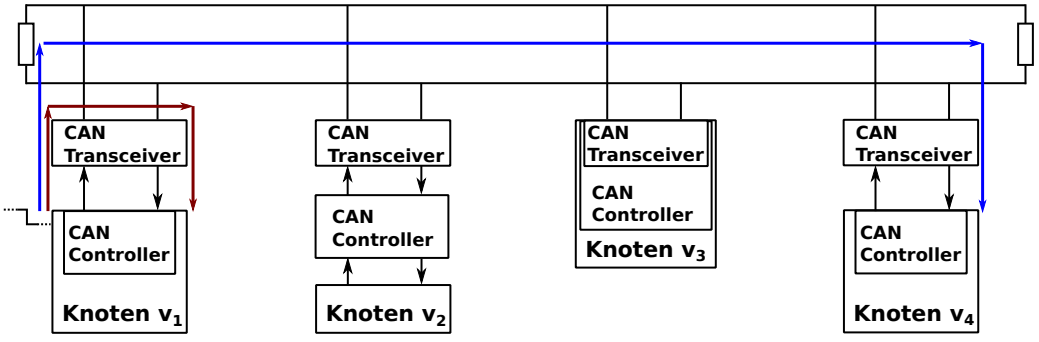


Abbildung 3.4: Anbindung von CAN Knoten an einen CAN-Bus.

Basierend auf den beschriebenen technischen Grundlagen der Anbindung und den Informationen aus dem CAN-Standard sowie Datenblättern von CAN-Transceiver und CAN-Controllern definieren wir uns ein Modell zur Beschreibung der auftretenden (maximalen) Signalverzögerungen. Die Signalverzögerung für die Übertragung eines Bits von einem Sende- zu einem Empfangsknoten setzt sich aus der Signallaufzeit auf dem Bus (Propagation-Delay), der Verarbeitungszeit der CAN-Controller sowie der CAN-Transceiver [Tex08a, Tex08b] zusammen. Bei den CAN-Transceivern spielen zusätzlich noch Temperatureinflüsse eine Rolle sowie der Umstand, ob von einem rezessiven zu einem dominanten Pegel gewechselt wird oder umgekehrt [Har12, Tex02, Nat96]. Der Fall, dass sich der Signalpegel nicht ändert ist an dieser Stelle uninteressant, da es aus Sicht des Empfängers hierbei zu keiner (sichtbaren) Signalverzögerung kommt. Das Propagation-Delay hängt dagegen direkt von der physikalischen Länge der Leitung zwischen Sende- und Empfangsknoten ab.

Definition 3.3 (Maximale und minimale Signalverzögerung)

Sei V die Menge der an einen CAN-Bus angeschlossenen Knoten. $E = \{\nearrow, \searrow\}$ bezeichnet den jeweiligen durch den Sender angelegten Flankenwechsel; $\nearrow$ repräsentiert eine steigende Flanke, $\searrow$ eine fallende Flanke.
Wir definieren die maximale Signalverzögerung zwischen zwei Knoten als Funktion $delay_{max} : V \times V \times E \to \mathbb{R}$. Die maximale Signalverzögerung für die Übertragung einer Flanke $e \in E$ zwischen Knoten $a \in V$ (Sender) und $b \in V$ (Empfänger) setzt sich zusammen aus

$$delay_{max}(a,b,e) \equiv d^{a,max}_{txProcDelay,e} + d^{a,max}_{txTransDelay,e} + d^{a,b,max}_{PropagationDelay,e} + d^{b,max}_{rxTransDelay,e} + d^{b,max}_{rxProcDelay,e}$$

Hierbei bezeichnet

$d^{x,max}_{txProcDelay,\nearrow}$, $d^{x,max}_{txProcDelay,\searrow}$ *die maximale interne Verarbeitungsverzögerung des CAN-Controllers des Knotens $x \in V$ bis zum Anliegen des neuen Signalpegels an dessen Ausgangspin (`TXD`-Pin).*

$d^{x,max}_{txTransDelay,\nearrow}$, $d^{x,max}_{txTransDelay,\searrow}$ *die maximale Signalverzögerung durch den CAN-Transceiver des Knotens $x \in V$ von der Pegeländerung e an dessen Eingangspin bis zur Ausgabe des entsprechenden differentiellen Signals auf dem Bus.*

$d^{x,y,max}_{PropagationDelay,\nearrow}$, $d^{x,y,max}_{PropagationDelay,\searrow}$ *Signallaufzeit (Propagation-Delay) aufgrund der physikalischen Länge des verwendeten elektrischen Leiters zwischen den Knoten $x \in V$ und $y \in V$.*

$d^{x,max}_{rxTransDelay,\nearrow}$, $d^{x,max}_{rxTransDelay,\searrow}$ *die maximale Signalverzögerung durch den CAN-Transceiver des Knotens $x \in V$, von der Erfassung eines differentiellen Signalpegels bis zu dessen digitaler Wandlung und Ausgabe an dem mit dem CAN-Controller verbundenen Ausgangspin des CAN-Transceivers.*

$d^{x,max}_{rxProcDelay,\nearrow}$, $d^{x,max}_{rxProcDelay,\searrow}$ *die maximale interne Verarbeitungszeit des CAN-Controllers des Knotens $x \in V$ zwischen dem Anliegen eines neuen Signalpegels an dessen Eingangspin (`RXD`-Pin) bis zu dessen Verarbeitung.*

Zur Vereinfachung definieren wir zusätzlich die Funktion $delay_{max} : V \times V \to \mathbb{R}$ als

$$delay_{max}(a,b) \equiv \max(delay_{max}(a,b,\nearrow), delay_{max}(a,b,\searrow))$$

Analog hierzu definieren wir die minimale Signalverzögerung zwischen zwei Knoten als Funktion $delay_{min} : V \times V \times E \to \mathbb{R}$, sowie

$$delay_{min}(a,b) \equiv \min(delay_{min}(a,b,\nearrow), delay_{min}(a,b,\searrow))$$

Definition 3.3 erfasst auch die lokale Übertragungsverzögerung (*loop-delay*). Hierbei handelt es sich um die Zeitspanne von dem Senden einer Flanke durch den CAN-Controller des Knotens, bis zur Registrierung einer Änderung des Bus-Pegels durch dessen eigenen CAN-Controller[4]. Der entsprechende Signalpfad ist in Abbildung 3.4 in rot für den Knoten v_1 eingezeichnet. In diesem Fall entfällt das Propagation-Delay, d.h. $d^{v_1,v_1,j}_{PropagationDelay,e} = 0$ für $e \in E$ und $j \in \{min, max\}$. Es bleiben die Verzögerung durch den CAN-Transceiver sowie die Verarbeitungszeiten des CAN-Controllers. Das maximale *loop-delay* des Knotens $v_1 \in V$ unter Verwendung von Definition 3.3 lässt sich mit dem Ausdruck $delay_{max}(v_1, v_1)$ beschreiben. Der größte Anteil des *loop-delay* entfällt auf den CAN-Transceiver (vgl. Tabelle A.1, Anhang A.1).

[4]Unter der Annahme, dass der Knoten als einziger Sender aktiv ist.

Für die Praxis ist es schwierig die einzelnen Kenngrößen von Definition 3.3 exakt zu ermitteln, da diese zumeist nur unvollständig in den Datenblättern der jeweiligen ICs angegeben sind. Da der CAN-Controller intern mit digitalen Signalen arbeitet, nehmen wir im Folgenden an, dass die internen Verarbeitungsverzögerungen für steigende und fallende Flanken identisch sind, d.h. $d^{x,j}_{rxProcDelay,\nearrow} = d^{x,j}_{rxProcDelay,\searrow}$ und $d^{x,j}_{txProcDelay,\nearrow} = d^{x,j}_{txProcDelay,\searrow}$ mit $x,y \in V$, $j \in \{min, max\}$. Ebenfalls identisch ist die Übertragungsverzögerung über das physikalische Medium, d.h. $d^{x,y,j}_{PropagationDelay,\nearrow} = d^{x,y,j}_{PropagationDelay,\searrow}$. Da der CAN-Standard die Verwendung von Kabeln mit einer maximalen Signallaufzeit von $5\frac{ns}{m}$ vorschreibt, lässt sich das maximale Propagation-Delay aus dem Abstand der Knoten bestimmen [ISO03a, Ets01]. Auch die bei den CAN-Transceivern auftretenden Signalverzögerungen sind nicht konstant bzw. für alle Busse identisch, sondern unter anderem von der (elektrischen) Kapazität des verwendeten Übertragungsmediums sowie von Temperatureinflüssen abhängig. In den Datenblättern vieler CAN-Transceiver finden sich jedoch häufig Angaben hinsichtlich der typischen und maximalen Signalverzögerungen, die in Berechnungen verwendet werden können. Tabelle A.1 in Anhang A.1 zeigt exemplarisch in welcher Größenordnung sich die aufgeführten Verzögerungen in der Praxis bewegen.

Für eine kompaktere Schreibweise definieren wir zusätzlich Ausdrücke für minimale und maximale Signalverzögerung innerhalb des gesamten TTCAN-Busses mit den Knoten V sowie innerhalb von Teilmengen dieser Knoten.

Definition 3.4

Seien $V' \subseteq V$ und $V'' \subseteq V$ zwei Teilmengen der an einen CAN-Bus angeschlossenen Knoten V. Dann definieren wir die minimale und maximale Signalverzögerung $delay^{V' \to V''}_{min}$ und $delay^{V' \to V''}_{max}$ bei der Übertragung der Knoten aus V' zu den Knoten aus V'' wie folgt:

$$delay^{V' \to V''}_{min} \equiv \min_{v_1 \in V', v_2 \in V''} (delay_{min}(v_1, v_2))$$

$$delay^{V' \to V''}_{max} \equiv \max_{v_1 \in V', v_2 \in V''} (delay_{max}(v_1, v_2))$$

Zur Vereinfachung schreiben wir,

$$delay^{V'}_{min} \equiv delay^{V' \to V'}_{min}$$

$$delay^{V'}_{max} \equiv delay^{V' \to V'}_{max}$$

Für Berechnungen wird die minimale Signalverzögerung in der Regel durch $delay^{V}_{min} = 0$ abgeschätzt, während man für die maximale Signalverzögerung entsprechende Maximalwerte, für die in Definition 3.3 angegebenen Größen, annimmt. Das maximale Propagation-Delay ergibt sich aus der maximalen Ausbreitungsverzögerung und der maximal zulässigen Leitungslänge. Für die Verzögerungen durch die CAN-Transceiver lassen sich Maximalwerte aus den entsprechenden Datenblättern ablesen (z.B. [Tex02, Nat96]). Auch für die Verarbeitungsverzögerung innerhalb des CAN-Controllers gibt es Erfahrungswerte aus der Literatur [Tex08b]. Die Tabelle A.1 in Anhang A.1 stellt die aus verschiedenen Quellen zusammengetragenen Werte einander gegenüber. Der so ermittelte Wert für die maximale Signalverzögerung $delay^{V}_{max}$ kann dann auch als Grundlage für die Wahl der Länge des Propagation Time Segments für das CAN-Bit-Timing (Constraint 3.1) dienen.

3.1.2 Grundlagen von TTCAN

TTCAN [ISO04] erweitert das CAN-Protokoll um eine zeitgetriggerte Betriebsart, welche die Gewährung deterministischer Garantien bezüglich auftretender Übertragungsverzögerungen ermöglicht. Diese Eigenschaft weist das CAN-Protokoll selbst nicht auf, da Übertragungen ereignisbasiert erfolgen und diese bei gleichzeitigem Medienzugriff strikt nach Priorität abgearbeitet werden. Dementsprechend können

Rahmen mit einer hohen Priorität, Rahmen mit niedrigerer Priorität beliebig lange verzögern. Da CAN für den Medienzugriff CSMA/CR verwendet, wird eine laufende Übertragung nicht unterbrochen. Somit kann die aktive Übertragung unter Umständen eine anstehende Übertragung eines Rahmens mit höherer Priorität verzögern[5]. Daraus folgt, dass selbst für die Nachricht mit der höchsten Priorität ein Jitter nicht ausgeschlossen werden kann [FMD+00]. Insgesamt wächst der maximale Jitter (Worst Case) mit abnehmender Priorität einer Nachricht [TB94] aufgrund der Funktionsweise der Arbitrierung.

Um diese Probleme zu beheben, setzt TTCAN auf TDMA, d.h. Rahmen werden nur zu vorbestimmten Zeitpunkten in festen Zeitslots gesendet. TTCAN baut hierbei auf dem CAN-Protokoll auf, d.h. sowohl das kollisionsgeschützte Medienzugriffsverfahren als auch die definierten Rahmenformate werden unverändert weiter genutzt. Für die Realisierung eines TDMA-basierten Zugriffsschemas unterteilt TTCAN die Zeit in Abschnitte fester Dauer. Ein solcher Zeitabschnitt wird Matrixzyklus (Matrix Cycle) genannt. Der Matrixzyklus wird kontinuierlich wiederholt und besteht selbst aus einer Abfolge von Basiszyklen (Basic Cycles). Die Basiszyklen sind ihrerseits in Zeitslots, sogenannte Time Windows, unterteilt. Die Struktur und der Aufbau eines Matrixzyklus wird mit Hilfe der Systemmatrix beschrieben. Hierbei bilden die Basiszyklen die Zeilen der Systemmatrix. Die Spalten der Systemmatrix, die sogenannten Transmission Columns, werden von den Time Windows gebildet (vgl. Abbildung 3.5). Die Länge der Time Windows kann frei konfiguriert werden, ist aber während des Ablaufs unveränderlich. Alle Time Windows einer Transmission Column müssen jedoch gleich lang sein. Des Weiteren fordert der TTCAN-Standard, dass die Anzahl der Basiszyklen innerhalb der Systemmatrix eine Zweierpotenz sein muss.

Voraussetzung für die Nutzung eines TDMA-basierten Medienzugriffsverfahrens ist eine gemeinsame Zeitbasis der einzelnen Knoten. Hierfür ist eine regelmäßige (Re-)Synchronisation[6] der lokalen Uhren der Knoten des CAN-Busses erforderlich. Aus diesem Grund ist das erste Time Window jedes Basiszyklus exklusiv für die Übertragung der Referencemessage reserviert. Bei der Referencemessage handelt es sich um einen gewöhnlichen CAN-Datenrahmen, welcher von einem als Time Master bezeichneten Knoten versendet wird. Der Empfangszeitpunkt des SOF-Bits der Referencemessage wird von den anderen Knoten als Referenzzeitpunkt für die Synchronisation verwendet. Darüber hinaus enthält der Datenrahmen als Nutzlast unter anderem die Zeilennummer des aktuellen Basiszyklus sowie optional einen globalen Zeitstempel für die Durchführung einer Zeitsynchronisation.

TTCAN unterstützt drei verschiedene Arten von Time Windows: *Exclusive Time Windows*, *Arbitrating Time Windows* und *Free Time Windows*.

Exclusive Time Windows sind genau einem Nachrichtentyp zugeordnet. Daher erfolgt die Übertragung innerhalb dieses Zeitfensters ohne Konkurrenz und gestattet somit deterministische Garantien bezüglich der Übertragungsverzögerung.

Arbitrating Time Windows werden von beliebigen Nachrichtentypen geteilt. Zugriffskonflikte werden durch den CAN-Arbitrierungsmechanismus gelöst. Innerhalb eines Basiszyklus aufeinanderfolgende Arbitrating Time Windows können miteinander verschmolzen werden.

Free Time Windows sind reserviert für spätere Erweiterungen des Ablaufplans.

Abbildung 3.5 zeigt eine Systemmatrix bestehend aus vier Basiszyklen mit jeweils 10 Time Windows. Die verschiedenen Time Windows haben unterschiedliche Längen, wobei alle Time Windows einer Transmission Column eine identische Länge aufweisen. Innerhalb einer Spalte variieren die verwendeten Typen von Time Windows. So verwendet z. B. Basiszyklus 3 ein Arbitrating Time Window, welches sich über 3 Transmission Columns erstreckt, während es sich in den anderen Basiszyklen um separate Time Windows handelt. Für die Exclusive Time Windows enthält die Systemmatrix auch die Zuordnung zu den

[5] In der Literatur ist diese Problematik unter der Bezeichnung Priotitätsinversion bzw. Priority Inversion bekannt [Tan07].

[6] Die bei TTCAN durchgeführte (Re-)Synchronisation der lokalen Uhren zu Beginn jedes Basiszyklus darf nicht mit der Bit-Resynchronisation von CAN verwechselt werden, welche sich auf die Bit-Übertragung bezieht. Diese beiden Konzepte sind unabhängig voneinander und kommen bei TTCAN beide zum Einsatz.

jeweiligen Nachrichtentypen. So ist die gesamte zweite Transmission Column für die Nachrichten mit dem Identifier 45 reserviert, wohingegen den Exclusive Time Windows in der dritten Spalte unterschiedliche Nachrichtentypen zugeordnet wurden. Auch kann ein Nachrichtentyp innerhalb eines Basiszyklus beliebig vielen Time Windows zugewiesen werden (vgl. Identifier 8 im dritten Basiszyklus).

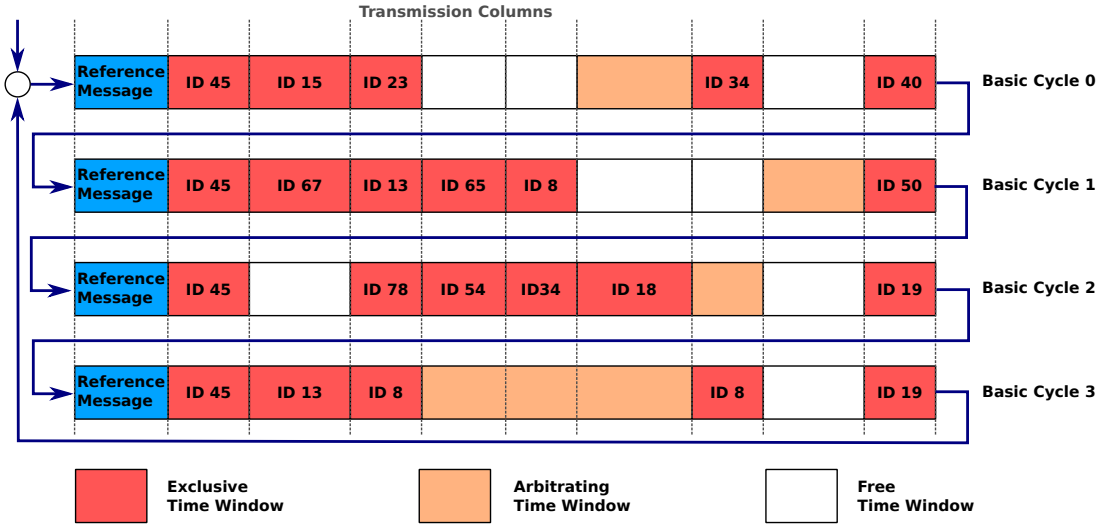


Abbildung 3.5: Systemmatrix mit vier Basiszyklen.

Der in blau dargestellte Pfeil zeigt den zeitlichen Ablauf: Nach dem Starten des Time Masters sendet dieser die Referencemessage des Basiszyklus mit der Nummer 0. Nach dem Ende dieses Basiszyklus überträgt der Time Master die Referencemessage für den Basiszyklus 1 und fährt entsprechend fort. Wurde das Ende der Systemmatrix erreicht, wiederholt sich dieser Ablauf mit dem ersten Basiszyklus. An dieser Stelle soll zur Vollständigkeit noch erwähnt werden, dass ein Time Master das Senden der Referencemessages nicht periodisch durchführen muss, sondern auch bezogen auf ein externes Ereignis. Über das `Next_is_Gap`-Bit der Referencemessage können die Knoten darüber informiert werden, dass die nächste Referencemessage nicht direkt im Anschluss gesendet wird, sondern erst nach einer variablen Zeitspanne oder auf ein bestimmtes Ereignis hin (vgl. [ISO04, 5.2.3]). Dieser Mechanismus kann z. B. für die Synchronisation mehrerer TTCAN-Netzwerke [HMFH02] genutzt werden.

Da jeder Knoten Beginn und Ende der Time Windows auf der Grundlage seiner lokalen Uhr festlegt, ist eine möglichst exakte Synchronisation notwendig, damit Übertragungen bei allen Knoten innerhalb der Grenzen der reservierten Time Windows beginnen und abgeschlossen werden. Je größer die maximale Synchronisationsungenauigkeit ist, desto größer müssen die Time Windows gewählt werden, um dies zu gewährleisten. Um die maximale Abweichung der Uhren zu begrenzen, erfolgt zu Beginn jedes Basiszyklus eine (Re-)Synchronisation. Doch trotz der regelmäßigen (Re-)Synchronisationen, lässt sich keine perfekte Synchronisation erreichen. Dies liegt zum einen an dem Clock Skew der lokalen Taktgeber, die während des Basiszyklus zu einer wachsenden Abweichung der lokalen Uhren führen kann, zum anderen an der Signalverzögerung (vgl. Definition 3.3). Diese ist dafür verantwortlich, dass die fallende Flanke des SOF-Bits der Referencemessages von den Knoten zu unterschiedlichen Zeitpunkten empfangen[7] wird. Direkt in Korrelation mit der erreichbaren Synchronisationsgenauigkeit steht neben der Länge der Time Windows auch der Zeitpunkt zu dem ein Knoten innerhalb des Time Windows frühestens seine Übertragung starten darf, damit sichergestellt ist, dass alle anderen Knoten den Übertragungsbeginn dem korrekten Time Window zuordnen. Kapitel 3.2.2, Constraint 3.3 beschreibt diese Zusammenhänge. Die diesbezüglichen Überlegungen gelten sowohl für Exclusive als auch und Arbitrating Time Windows.

[7] Die Art der Flankenerkennung bei CAN führt zu weiteren Abweichungen (vgl. Kapitel 3.1.1).

TTCAN unterscheidet zwei verschiedene Level (*TTCAN-Level-1* und *TTCAN-Level-2*) bei der Implementierung des TTCAN-Standards [HMFH02]. Beiden ist gemein, dass die Knoten anhand ihrer lokalen Uhren entscheiden, wann (bezogen auf dem Empfang der Referencemessage) Time Windows starten bzw. enden und wann eine Übertragung zu starten ist. Die lokale Zeit eines TTCAN-Controllers wird vom Systemtakt durch einen Frequenzteiler (TUR – Time Unit Ratio) abgeleitet. Der hieraus resultierende Takt wird als NTU (Network Time Unit) bezeichnet. *TTCAN-Level-1* verwendet eine einfache Ticksynchronisation, welche auf dem Empfangszeitpunkt des SOF-Bits der Referencemessage beruht. Die verwendete Auflösung für die NTU beträgt eine Bitzeit. Der Frequenzteiler TUR wird bei der Konfiguration fest und konstant gewählt, um die gewünschte NTU aus dem Systemtakt zu generieren. Die lokale Uhr wird durch einen einfachen 16-Bit Zähler realisiert, der einmal pro NTU inkrementiert wird. Die Nutzlast der Referencemessage enthält die Nummer des Basiszyklus innerhalb des laufenden Matrixzyklus und Informationen darüber, ob zwischen dem gerade gestarteten Basiszyklus und dem nachfolgenden eine Pause eingeplant ist. Durch die Nutzung der Ticksynchronisation ist es möglich, TTCAN-Level-1 in Software, unter Verwendung handelsüberlicher CAN-Controller[8], zu implementieren (vgl. Kapitel 3.3).

TTCAN-Level-2 basiert auf den gleichen Konzepten, erweitert TTCAN-Level-1 jedoch um eine Zeitsynchronisation und Mechanismen für eine höhere Synchronisationsgenauigkeit. Als Referenzzeitpunkt für die Synchronisation dient wieder der Empfangszeitpunkt der fallenden Flanke des SOF-Bits der Referencemessage. Um die Ungenauigkeit zu reduzieren, wird dieser direkt innerhalb des TTCAN-Controllers erfasst. Zusätzlich ist die für die lokale Uhr verwendete Auflösung im Vergleich zu TTCAN-Level-1 höher. Der verwendete Zähler umfasst mindestens 19 statt 16 Bit und zählt auch Bruchteile der konfigurierten NTU. Neben der reinen Korrektur des Clockoffsets enthält TTCAN-Level-2 auch Mechanismen zur Kompensation der Clock Skew. Hierzu misst jeder Knoten die Dauer eines Basiszyklus und bestimmt aufgrund der bekannten Größe des Basiszyklus einen Korrekturfaktor für den TUR, um hierüber den Clock Skew zwischen dem Knoten und dem Time Master auszugleichen. Für die Implementierung der Zeitsynchronisation wird in der Referencemessage zusätzlich die lokale Zeit des Time Masters übertragen.

Da der Umstand, dass TTCAN-Level-2 eine Zeitsynchronisation durchführt, für die eigentliche Funktionsweise keine Relevanz aufweist, werden wir im Folgenden vereinfacht von einer Ticksynchronisation sprechen, in dem Bewusstsein, dass sich je nach Implementierungslevel unterschiedliche Mechanismen dahinter verbergen können. Nur an den Stellen, wo es erforderlich ist, werden wir auf die Unterschiede in Bezug auf die verschiedenen Implementierungslevel von TTCAN eingehen.

3.2 Realisierung von Mode-Based Scheduling with Fast Mode-Signaling mit TTCAN

Im vorherigen Kapitel haben wir uns mit den Grundlagen von CAN und TTCAN beschäftigt und den aktuellen Stand der Technik beschrieben. CAN ist ein nachrichtenbasierter, ereignisgetriggerter Feldbus und unterstützt ausschließlich die ereignisgetriggerte Kommunikation. Konflikte beim gleichzeitigen Zugriff auf das Medium werden prioritätsbasiert durch die CAN-Arbitrierung aufgelöst. Hierdurch wird zwar eine geringe mittlere Latenz in Bezug auf Nachrichten mit hoher Priorität sowie eine effiziente Nutzung der verfügbaren Bandbreite erreicht (insbesondere bei sporadischen Nachrichten), allerdings sind nur eingeschränkt Echtzeitgarantien möglich, da Nachrichten mit höherer Priorität solche mit niedrigerer Priorität beliebig lange verzögern können.

TTCAN erweitert CAN um eine zeitgetriggerte Kommunikation, unterstützt jedoch nur Time Windows mit exklusiven Zuordnungen sowie Arbitrating Time Windows, in denen alle Knoten ereignisbasiert kommunizieren. Aufgrund der eingeschränkten Echtzeitgarantien in den Arbitrating Time Windows sind für sporadische Nachrichten mit hohen Echtzeitanforderungen exklusive Reservierungen notwendig. Dies

[8] Der verwendete CAN-Controller muss lediglich die Deaktivierung der automatischen erneuten Übertragung erlauben.

führt bei seltenen sporadischen Nachrichten jedoch dazu, dass die reservierten Slots häufig ungenutzt bleiben und die verfügbare Bandbreite somit schlecht genutzt wird.

Mode-Based Scheduling with Fast Mode-Signaling kombiniert die Vorteile ereignis- und zeitgetriggerter Kommunikation und verbessert die Bandbreitennutzung bei gleichzeitiger Gewährleistung der Echtzeitfähigkeit sowie deterministischer Garantien hinsichtlich auftretender Verzögerungen. Hierfür erlaubt *Mode-Based Scheduling* einen eingeschränkten kontrollierten Wettbewerb innerhalb der Zeitslots, welcher durch *Fast Mode-Signaling* aufgelöst wird. In [KG13] wird bereits vorgeschlagen, *Mode-Based Scheduling with Fast Mode-Signaling* auf Basis von TTCAN zu realisieren und *Fast Mode-Signaling* über die CAN-Arbitrierung abzubilden, geht aber nicht näher auf die Umsetzung und die hierfür notwendigen Erweiterungen des TTCAN-Protokolls ein. Wir betrachten an dieser Stelle im Detail, wie das TTCAN-Protokoll angepasst und erweitert werden muss, um die Konzepte von *Mode-Based Scheduling* auf TTCAN zu übertragen, und wie *Fast Mode-Signaling* auf dieser technischen Basis umgesetzt werden kann (Kapitel 3.2.1). Insbesondere widmen wir uns hierbei den Schwierigkeiten, die sich bei dieser Art der Realisierung von *Fast Mode-Signaling* bei einer realen Implementierung ergeben und entwickeln Lösungen, um *Fast Mode-Signaling* zuverlässig, robust und deterministisch durch die CAN-Arbitrierung umzusetzen. Hierfür analysieren wir zunächst die auftretenden Probleme und definieren geeignete Constraints für die Konfiguration des TTCAN Basiszyklus sowie den von uns eingeführten *Mode-Based Time Windows*– in Anlehnung an die Protokollspezifikation von FlexRay–, bei deren Einhaltung eine zuverlässige Funktionalität des *Fast Mode-Signaling* sichergestellt ist (Kapitel 3.2.2). Diese Erweiterungen von TTCAN, sowohl auf konzeptioneller als auch technischer Ebene, bezeichnen wir als *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN*.

Anschließend evaluieren wir *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* (Kapitel 3.3) und präsentieren mit *Bring-your-own Tick (BOT)* eine funktionale Erweiterung für TTCAN, welche die Robustheit des *Fast Mode-Signaling* verbessert und eine deterministische Realisierung in der Praxis ermöglicht (Kapitel 3.4).

3.2.1 Instanziierung des abstrakten modusbasierten Kommunikationsmodells für TTCAN

Zunächst übertragen wir die Konzepte von *Mode-Based Scheduling with Fast Mode-Signaling* auf TTCAN und dessen Matrixzyklus. Hierfür genügt es, das in Kapitel 2.3.1 eingeführte abstrakte Kommunikationsmodell für *Mode-Based Scheduling* leicht anzupassen, so dass es ebenfalls für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* verwendet werden kann. Definition 3.5 beschreibt formal das Mediumslotting von TTCAN, unter Berücksichtigung der TTCAN-Matrixzyklen und entspricht konzeptionell Definition 2.1 des abstrakten Kommunikationsmodells. Die Anpassungen sind lediglich formaler Natur oder betreffen die Verwendung von TTCAN-spezifischen Fachbegriffen sowie die Berücksichtigung der Eigenschaften der TTCAN Systemmatrix. Die Definition 3.5 fasst diese zusammen und erlaubt die eingeführten Definitionen 2.3 bis 2.8 (Kapitel 2.3.2) auch in diesem Kontext anzuwenden.

Definition 3.5

TTCAN unterteilt die Zeit in eine unendliche, fortlaufend nummerierte Sequenz $\Theta = \{Z_1, Z_2, \dots\}$ *von Matrixzyklen. Jeder Matrixzyklus* $Z = \{S_1, \dots, S_{2^k}\}$ *besteht aus einer endlichen Menge von* 2^k *Basiszyklen* $k \in \mathbb{N}_0$*. Jeder Basiszyklus* S_i *ist unterteilt in eine endliche Menge von (fortlaufend nummerierten) Time Windows, die eine unterschiedliche Länge aufweisen können, d.h.* $S_i = \{s_0^i, \dots, s_n^i\}$*. Jedes Time Window kommt nur einmal innerhalb eines Matrixzyklus (und somit auch innerhalb eines Basiszyklus) vor.*
Das erste Time Window s_0 *jedes Basiszyklus* $S_i \in Z$ *ist exklusiv für die Übertragung der Reference-message reserviert und es gelte ferner, dass Time Windows einer Transmission Column die gleiche Länge besitzen müssen, d.h.*

$$\forall S_i, S_j \in Z. \forall s_x^i \in S_i, \forall s_y^j \in S_j.\ (x = y) \Rightarrow (length(s_x^i) = length(s_y^j)).$$

Die Funktion $length : Z \to \mathbb{R}$ *ordnet einem Time Window eine Länge zu.*
Des Weiteren definieren wir S als die Abfolge aller Time Windows eines Matrixzyklus, d.h.

$$S \equiv \{s_0^0, \dots s_n^0, s_0^1 \dots, s_n^1, \dots\dots, s_0^{2^k-1}, \dots, s_n^{2^k-1}\}$$

Verwenden wir die Definition 3.5 anstelle von Definiton 2.1 als Ausgangsbasis für die *Modus-Präferenz*-Funktion *mp* und die *Slot-Assignment*-Funktion *SA* (Definitionen 2.4 und 2.5, Kapitel 2.3.2), so können diese ohne Einschränkungen auf *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* übertragen werden. Deshalb verzichten wir auf die erneute Darstellung der formalen Definitionen. Wir gehen im Folgenden jeweils davon aus, dass $M = \{m_1, \dots, m_r\}$ die Menge der Modes und $V = \{v_1, \dots, v_s\}$ die Knoten des TTCAN-Busses bezeichnen.

Die Realisierung von *Mode-Based Scheduling* erfordert die Bereitstellung von Zeitslots, in denen ein eingeschränkter Wettbewerb um das Medium zwischen bestimmten, durch die *Slot-Assignment*-Funktion festgelegten Knoten erfolgen kann. Allen anderen Knoten ohne Slotzuordnung ist es nicht erlaubt, in diesem Slot, um das Medium zu konkurrieren. Daher ergänzen wir die drei von TTCAN unterstützten Time Window Typen um einen weiteren Typ, das *Mode-Based Time Window*. Dieses dient speziell der Umsetzung der modusbasierten Kommunikation.

Da TTCAN auf CAN aufbaut und dieses unverändert für den Medienzugriff verwendet, ist die CAN-Arbitrierung in allen Time Windows weiterhin aktiv. Dies nutzen wir aus, um auf der Basis der CAN-Arbitrierung *Fast Mode-Signaling* zu realisieren. Hierzu müssen die vergebenen Modes in geeigneter Weise auf CAN-Identifier abgebildet werden, sodass die CAN-Prioritäten *verträglich* mit den definierten *Modus-Präferenzen* sind. Definition 3.6 formuliert die Anforderungen an eine *verträgliche* Abbildung von Modes auf CAN-Identifier.

Definition 3.6

Sei $CAN_{ID} \subseteq \mathbb{N}_0$ *die Menge der zulässigen CAN-Identifier für die gewählte Bus-Konfiguration, S die Menge der Time Windows des konfigurierten Matrixzyklus und* $mp : S \times M \to \mathbb{N}_0$ *eine Modus-Präferenz-Funktion für die Menge der Modes* $M = \{m_1, \dots, m_r\}$*. Dann heißt eine Funktion* $c\_map_{mp} : S \times M \to CAN_{ID}$ *verträglich mit mp, genau dann, wenn*

$$\forall s \in S, m_1 \in M_s, m_2 \in M_s.(mp(s,m_1) < mp(s,m_2) \Rightarrow c\_map_{mp}(s,m_1) < c\_map_{mp}(s,m_2))$$

mit

$$M_s \equiv \{m \in M | SA(s,m) \text{ ist definiert}\}.$$

Eine gemäß Definition 3.6 mit der *Modus-Präferenz*-Funktion verträgliche Abbildung zwischen Modes und CAN-Identifier sorgt dafür, dass die durch die *Modus-Präferenzen* eingeführte Präferenzordnung korrekt auf die vergebenen CAN-Identifier und deren durch die Arbitrierung realisierte Präferenzordnung übertragen wird.

Die zentrale Voraussetzung und Herausforderung für die korrekte und zuverlässige Implementierung von *Fast Mode-Signaling* ist, dass innerhalb eines *Mode-Based Time Window* alle Knoten mit einer Slotzuordnung *hinreichend zeitgleich* mit der Übertragung ihres Rahmens beginnen, sodass sich die SOF-Bits der gesendeten Rahmen überlagern. Nur unter dieser Voraussetzung ist gewährleistet, dass das CAN-Arbitrierungsverfahren alle Sender korrekt berücksichtigt und am Ende der Knoten mit der höchsten Priorität – und dementsprechend auch der höchsten *Modus-Präferenz* – die Arbitrierung gewinnt und seinen Rahmen ohne Verzögerung senden kann. Wir werden uns in Kapitel 3.2.2 im Detail damit beschäftigen, wie das *Mode-Based Time Window* aufgebaut sein muss, um einen *hinreichend zeitgleichen* Übertragungsstart zu realisieren. Dabei untersuchen wir auch, was *hinreichend zeitgleich* in diesem Kontext genau bedeutet und wie groß die Toleranz zwischen den Übertragungsstarts maximal sein darf, damit das *Fast Mode-Signaling* durch die CAN-Arbitrierung korrekt umgesetzt wird (vgl. Kapitel 3.2.2.3).

Doch bevor wir uns im Detail mit den Einzelheiten der entworfenen Erweiterungen und deren technischen Hintergründen auseinandersetzen, betrachten wir zunächst in welchen Punkten sich *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* von TTCAN unterscheidet. Zunächst ist festzustellen, dass die vorgestellte Form der Realisierung von *Fast Mode-Signaling* sehr effizient ist, da keine Anpassung der CAN-Arbitrierung notwendig ist. Deshalb resultiert aus deren Umsetzung kein Overhead, weil der CAN-Arbitrierungsmechanismus bei TTCAN als fester und essentieller Bestandteil des CAN-Protokolls ohnehin in jedem Time Window zum Einsatz kommt. Obwohl diese Art der Nutzung von TTCAN sehr intuitiv erscheint, wird weder *Mode-Based Scheduling* – also die Beschränkung der Konkurrenz auf eine Gruppe von Nachrichtentypen innerhalb eines Time Window – noch die entsprechende Nutzung der Arbitrierung in diesem Sinne in den TTCAN bezogenen Veröffentlichungen vorweggenommen (vgl. [ISO04, FMD+00, HMFH02, KG13]). Wird einem *Mode-Based Time Window* nur ein einziger Nachrichtentyp bzw. Mode zugeordnet, resultiert daraus für die jeweilige Slotinstanz eine exklusive Nutzung mit entsprechenden deterministischen Garantien. Das Verhalten entspricht dann einem *Exclusive Time Window*. Dies gilt ebenfalls für eine Zuordnung mehrerer Modes zu einem *Mode-Based Time Window*, wenn sich die Nachrichtentypen gegenseitig ausschließen (vgl. Muster 2).

Ein weiterer Unterschied sowohl zu CAN als auch zu TTCAN besteht darin, dass die für die Kodierung der Modes verwendeten CAN-Identifier in verschiedenen Zeitslots für unterschiedliche Nachrichtentypen (gesendet von unterschiedlichen Knoten) eingesetzt werden können. Dies ist gemäß CAN-Standard nicht zulässig, verursacht jedoch beim *Mode-Based Scheduling* keine Probleme, da jeder Mode nur maximal einem Knoten pro Zeitslot zugeteilt werden darf und eine *Modus-Präferenz* pro Mikroslot höchstens einem Mode zugeordnet ist (vgl. Definition 2.5 und 3.6).

Ein Vorteil, den *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* ebenso wie TTCAN miteinander teilen, ist der Umstand, dass das eingesetzte Protokoll konform zum CAN-Standard ist und somit handelsübliche Standard CAN-Controller die gesendeten Rahmen empfangen und interpretieren können. Daher lassen sich reine Empfangsknoten für TTCAN, aber auch für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* mit einem Standard CAN-Controller und einem Softwarestack für die Verarbeitung der empfangenen Nachrichten implementieren. Wie wir in Kapitel 3.3 zeigen werden, lässt sich auf Basis eines Standard CAN-Controllers zusammen mit einem von uns entwickelten Softwarestack auch ein Knoten mit vollständiger TTCAN-Funktionalität (TTCAN-Level-1) realisieren, der als Sender und Empfänger agieren kann und dabei ebenfalls *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* unterstützt. Wie unsere Tests gezeigt haben (Kapitel 3.3.3) arbeitet der Softwarestack (ohne jegliche Hardwareunterstützung) jedoch nur bis zu einer bestimmten maximalen Übertragungsrate zuverlässig.

3.2.2 Voraussetzungen für die zuverlässige Umsetzung von Mode-Based Scheduling with Fast Mode-Signaling für TTCAN

Mode-Based Scheduling with Fast Mode-Signaling ist ein Verfahren, welches eine effiziente Bandbreitennutzung mit Echtzeitfähigkeit und deterministischen Garantien verbindet. Für die Realisierung von *Mode-Based Scheduling* stellt das *Fast Mode-Signaling* die eigentliche Schlüsselfunktionalität dar. Das bedeutet, ohne eine zuverlässige, robuste und deterministische Umsetzung von *Fast Mode-Signaling* kann *Mode-Based Scheduling* keine deterministischen Garantien bezüglich der Echtzeitfähigkeit gewährleisten, da nicht sichergestellt ist, ob sich in einem Mikroslot immer (deterministisch) der Mode mit der höchsten *Modus-Präferenz* durchsetzt.

Bei der hier vorgestellten Realisierung von *Fast Mode-Signaling* mit der CAN-Arbitrierung ist dies nur sichergestellt, wenn alle zueinander in Konkurrenz stehenden Knoten eines *Mode-Based Time Window* ihre Übertragung *hinreichend zeitgleich* starten, sodass sich deren SOF-Bits auf dem Medium überlagern. Nur in diesem Fall gewinnt der Rahmen mit der höchsten *Modus-Präferenz* (dem nummerisch niedrigsten CAN-Identifier) die CAN-Arbitrierung. Weicht der Übertragungsstart der Knoten zu stark voneinander ab, könnte sich auch ein Knoten mit einer niedrigeren *Modus-Präferenz* durchsetzen, wenn dieser seine Übertragung ausreichend lange vor den anderen Knoten startet. Ursache hierfür kann ein zu großer Tickoffset zwischen den Knoten sein, der dafür sorgt, dass die Übertragungsstarts zwar nach der lokalen Uhr der Knoten zu einem identischen Zeitpunkt erfolgen, die hieraus resultierenden absoluten Zeitpunkte aber zu weit auseinander liegen.

Wir werden in diesem Kapitel auf Grundlage der Signalverzögerungen sowie der Eigenschaften und gewählten Konfiguration von CAN (harte Synchronisation, Bit-Timing) eine obere Schranke $d^{MAX}_{maxOffset,MB}$ für den maximal zulässigen Tickoffset herleiten, bei deren Einhaltung die korrekte Funktionalität des *Fast Mode-Signaling* gewährleistet ist. Liegt die tatsächliche Abweichung der Uhren unterhalb dieser oberen Schranke $d^{MAX}_{maxOffset,MB}$, so starten die Knoten ihre Übertragung *hinreichend zeitgleich*, d.h.,es ist gewährleistet, dass sich SOF-Bits aller konkurrierenden Knoten (ausreichend) überlagern und die CAN-Arbitrierung von dem Knoten mit der höchsten *Modus-Präferenz* gewonnen wird.

Zuerst untersuchen wir von welchen technischen Größen die maximale Synchronisationsungenauigkeit[9] (Tickoffset) $d_{maxOffset}$, die bei TTCAN auftreten kann, abhängt (Kapitel 3.2.2.1). Hierbei spielt auch die Länge des Basiszyklus eine Rolle (Resynchronisationintervall), welche selbst durch die Länge der enthaltenen Time Windows festgelegt wird. Jedes Time Window muss hinreichend lang gewählt werden, sodass eine Verletzung der Integrität der angrenzenden Time Windows auch bei Auftreten der maximalen Synchronisationsungenauigkeit $d_{maxOffset}$ ausgeschlossen ist. Ansonsten sind keine deterministischen Garantien möglich. Dies gilt sowohl für TTCAN als auch für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN*. Über die Dimensionierung der Time Windows enthält der TTCAN-Standard keine Aussagen, sondern verlässt sich diesbezüglich auf die Anwender.

Wir wollen an dieser Stelle ins Detail gehen und sowohl den internen Aufbau der *Mode-Based Time Windows* genau definieren als auch Constraints herleiten, die bei der Dimensionierung verwendet werden können und bei deren Einhaltung die Integrität angrenzender Time Windows sichergestellt ist. Diese Constraints sowie der beschriebene Aufbau der *Mode-Based Time Windows* können ohne große Anpassungen auch für die Dimensionierung der *Exclusive Time Windows* und *Arbitrating Time Windows* herangezogen werden (Kapitel 3.2.2.2). Im letzten Schritt bestimmen wir dann die obere Schranke $d^{MAX}_{maxOffset,MB}$ für den maximal zulässigen Tickoffset, welcher die korrekte Funktionsweise von *Fast Mode-Signaling* garantiert (Kapitel 3.2.2.3). Anhand der maximalen Synchronisationsungenauigkeit $d_{maxOffset}$ eines gegebenen TTCAN-Basiszyklus lässt sich dann prüfen, ob dieser den zuverlässigen Einsatz von *Mode-Based Scheduling with Fast Mode-Signaling* erlaubt.

Natürlich kann man umgekehrt auch Konfigurationen für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* erstellen, indem man zunächst für das gewählte Bit-Timing die obere Schranke

[9] Wir verwenden hier die Begriffe *maximale Synchronisationsungenauigkeit* und *maximaler Tickoffset* synonym.

$d^{MAX}_{maxOffset,MB}$ für den maximal zulässigen Tickoffset bestimmt. Anhand dieses Wertes kann über die maximale Synchronisationsungenauigkeit $d_{maxOffset}$, die TTCAN zusichert, dann die maximale Länge des Basiszyklus ermittelt werden. Im nächsten Schritt wird der Basiszyklus unter Berücksichtigung der Constraints für die Dimensionierung der Time Windows unterteilt und der Matrixzyklus erstellt. Auf diese Weise ergibt sich eine Konfiguration für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN*, für die die korrekte Funktionalität von *Fast Mode-Signaling* sowie die Integrität der einzelnen Time Windows sichergestellt ist.

3.2.2.1 Synchronisationsgenauigkeit von TTCAN

Da wir primär die Voraussetzungen für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* untersuchen wollen, beschränken wir die formulierten Constraints und Definitionen auf diejenigen Knoten, die innerhalb eines *Mode-Based Time Window* eine Übertragung starten dürfen[10]. Diese werden durch die Slot-Assignment-Funktion festgelegt. Für alle anderen Knoten muss lediglich gewährleistet sein, dass die Grenzen ihrer Time Windows, die wiederum von deren knotenlokalen Uhren abhängen, nicht verletzt werden. Wir unterscheiden zwischen dem maximalen Tickoffset $d_{maxOffset}$, aller an den TTCAN-Bus angebundenen Knoten und dem maximalen Tickoffset $d_{maxOffset,MB}$ der Knoten $V_{MB} \subseteq V$, welche über ein Slot-Assignment für mindestens ein *Mode-Based Time Window* verfügen. Im Weiteren bezeichnen wir die Knoten mit einer Slotzuordnung für mindestens ein *Mode-Based Time Window* als modusbasierte Knoten[11].

Um eine bestehende TTCAN-Konfiguration hinsichtlich ihrer Tauglichkeit für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* zu testen oder um eine neue Konfiguration zu erstellen, benötigen wir eine Definition mit deren Hilfe der maximale Tickoffset $d_{maxOffset}$ und $d_{maxOffset,MB}$ ermittelt werden kann. Hierbei handelt es sich um den Tickoffset unmittelbar vor der erneuten (Re-)Synchronisation, welcher von der Länge der Basiszyklen (Resynchronisationsintervall) sowie der Gangungenauigkeit (Clock Skew[12]) der lokalen Uhren der Knoten abhängt (vgl. hierzu auch [GK11]). Da die Definitionen von $d_{maxOffset}$ und $d_{maxOffset,MB}$ nahezu identisch sind, beschränken wir uns hier auf die Darstellung der Definition von $d_{maxOffset,MB}$ und verweisen ansonsten auf den Anhang A.2.

Definition 3.7 (Maximaler Tickoffset $d_{maxOffset,MB}$)

Sei d_{Cycle} die Länge des Basiszyklus und $V_{MB} \subseteq V$ die Menge der modusbasierten Knoten des TTCAN-Busses. Des Weiteren gebe $d_{maxBaseOffset,MB}$ die maximale paarweise Ungenauigkeit der modusbasierten Knoten unmittelbar nach der Synchronisation durch den Empfang der Referencemessage an (Baseoffset). Darüber hinaus sei $r_{maxClockSkew}$ der maximale Clock Skew[13] der Knoten aus V_{MB}. Dann ist der maximale Tickoffset am Ende des Basiszyklus

$$d_{maxOffset,MB} \equiv d_{maxBaseOffset,MB} + 2 \cdot r_{maxClockSkew} \cdot d_{Cycle}.$$

Für die Durchführung der Synchronisation nutzen die Knoten den Empfangszeitpunkt des SOF-Bits der Referencemessage als Referenzzeitpunkt. Dieser Empfangszeitpunkt variiert bei den verschiedenen Knoten unter anderem aufgrund der unterschiedlichen Signalverzögerungen zwischen den Empfängern und dem Time Master. Dementsprechend definieren wir den maximalen Baseoffset $d_{maxBaseOffset,MB}$ (unmittelbar) nach der Synchronisation als die maximale Abweichung der Empfangszeitpunkte des SOF-Bits der Referencemessage auf Basis der maximalen Signalverzögerung (Definition 3.3) zwischen den einzelnen Knoten.

[10] Eine entsprechende Formulierung der Constraints für Exklusive und Arbitrating Time Windows folgt analog, indem jeweils die Knoten verwendet werden, welche in den jeweiligen Time Windows senden dürfen.

[11] $V_{MB} \equiv \{v \in V | \exists s \in S, m \in M.SA(s,m) = v\}$

[12] Wie gehen im Folgenden davon aus, dass der Clock Skew relativ in *ppm – parts per million* – angegeben wird.

[13] Der maximale Clock Skew $r_{maxClockSkew}$ wird relativ in *ppm – parts per million* – angegeben.

Definition 3.8 (Maximaler Baseoffset $d_{maxBaseOffset,MB}$ nach der Synchronisation)

Sei $V_{MB} \subseteq V$ die Menge der modusbasierten Knoten des TTCAN-Busses und $v_T \in V$ der aktive Time Master des TTCAN-Busses. Dann ist der maximale Baseoffset $d_{maxBaseOffset,MB}$ die maximale paarweise Ungenauigkeit der modusbasierten Knoten V_{MB} unmittelbar nach dem Empfang der Referencemessage.
Der maximale Baseoffset $d_{maxBaseOffset,MB}$ kann abgeschätzt werden durch,

$$\begin{aligned} d_{maxBaseOffset,MB} &\leq \max_{v_1 \in V_{MB}, v_2 \in V_{MB}} |delay_{max}(v_T, v_1) - delay_{min}(v_T, v_2) + d_L^{v_1,v_2}| \\ &\leq delay_{max}^{V_{MB} \cup \{v_T\}} - delay_{min}^{V_{MB} \cup \{v_T\}} + \max_{v_1 \in V_{MB}, v_2 \in V_{MB}} d_L^{v_1,v_2} \end{aligned}$$

mit

$$d_L^{x,y} = \max(d_L^x, d_L^y)$$

$$d_L^x = \begin{cases} d_q^x, & \text{für TTCAN-Level-1} \\ d_{TimeStampPrec}^x & \text{für TTCAN-Level-2} \end{cases}$$

wobei d_q^v die konfigurierte Länge des Zeitquantums des Knotens $v \in V$ bezeichnet. Ist $v \in V$ ein Knoten mit TTCAN-Level-2 Unterstützung, so bezeichnet $d_{TimeStampPrec}^v$ die maximale Ungenauigkeit bei der Ermittlung des Zeitstempels der fallenden Flanke des SOF-Bits der Referencemessage durch den TTCAN-Controller (in Hardware).

Während TTCAN-Level-2 die für die Synchronisation relevante fallende Flanke des SOF-Bits der Referencemessage direkt in der Hardware erfasst, bietet TTCAN-Level-1 eine solche Hardwareunterstützung nicht bzw. nur sehr eingeschränkt an [HMFH02]. Die bei TTCAN-Level-2 anfallende maximale Ungenauigkeit $d_{TimeStampPrec}^v$ bei der Ermittlung des Zeitstempels des SOF-Bits der Referencemessage ist abhängig von dem TTCAN-Controller, dessen Konfiguration der NTU, der Auflösung der lokalen Uhr sowie der Abtastrate. Nähere Informationen über die genaue Umsetzung und den hieraus resultierenden Verzögerungen müssen dem Datenblatt des jeweiligen TTCAN-Controllers[14] entnommen werden.

Bei TTCAN-Level-1 gehen wir von einer Realisierung der Synchronisation anhand eines Interrupts aus, welcher beim Empfang eines Rahmens ausgelöst wird. Schließen wir Verzögerungen für das Auslösen des Interrupts[15] aus und setzen eine identische Realisierung bei allen Knoten voraus, erreichen wir auf diese Weise eine Auflösung von einem Zeitquantum, da CAN den Buspegel jeweils nur zu Beginn eines Zeitquantums abtastet. Da die Zeitquanten verschiedener Empfänger nicht synchron zueinander ablaufen, ergibt sich auf diese Weise im schlimmsten Fall ein Phasenunterschied von einem Zeitquantum [HB99]. Dieser Aspekt wird durch die Definition von $d_L^{x,y}$ berücksichtigt. Das auch bei TTCAN-Level-1 geforderte automatische Erfassen des Empfangszeitpunktes des SOF-Bits durch den CAN-Controller liefert nur eine Auflösung von einer Bitzeit [HMFH02, STM11]. Dieser Ansatz ist zu ungenau für die Realisierung von *Fast Mode-Signaling* (wie in der weiteren Darstellung noch ersichtlich wird) und wurde daher nicht in der Definition berücksichtigt.

Definition 3.8 erlaubt ebenfalls, dass der Time Master ein modusbasierter Knoten ist. Hierbei nehmen wir an, dass der Time Master als Referenzzeitpunkt für die Synchronisation den Empfangszeitpunkt der

[14] Aktuell sind nach Kenntnis des Autors noch keine TTCAN-Controller mit TTCAN-Level-2 Unterstützung verfügbar, daher fehlen Daten zur typischen Größenordnung von $d_{TimeStampPrec}^v$. Da TTCAN-Level-2 in Bezug auf die Synchronisation jedoch eine höhere Genauigkeit als TTCAN-Level-1 anstrebt und eine Abtastung mindestens zu Beginn jedes Zeitquantums erfolgt, dürfte die maximale Verzögerung kleiner als ein Zeitquantum ausfallen.

[15] Hierbei nehmen wir an, dass die Verzögerung für das Auslösen des Interrupts bei allen Knoten identisch ist (sofern vorhanden). Bei der Realisierung in Software gehen wir des Weiteren davon aus, dass die Unterschiede hinsichtlich der Verarbeitungszeit bei einer identischen Realisierung durch den Einsatz von Techniken, wie Early-Timestamping, vernachlässigbar ist.

fallenden Flanke des von ihm selbst gesendeten SOF-Bits nutzt. Dieser Zeitpunkt ist nicht identisch mit dem Sendezeitpunkt, da hierbei die Signalverzögerung des CAN-Transceivers des Time Masters mit berücksichtigt wird (*loop-delay*, vgl. Kapitel 3.1.1). Diese Signalverzögerung wird über den Ausdruck $delay_{max}(v_T, v_T)$ in der Definition 3.8 erfasst. Ebenso spiegelt der Ausdruck $d_L^{v_T,v_T}$ den Umstand wider, dass der verzögerte Empfang im Allgemeinen nicht auf die Grenzen der lokalen Zeitquanten ausgerichtet ist (TTCAN-Level-1). Dies gilt entsprechend für TTCAN-Level-2 in Bezug auf die maximale Ungenauigkeit bei der Ermittlung des Zeitstempels, sofern auch hier als Referenzzeitpunkt die eigene fallende Flanke für die (Re-)Synchronisation genutzt wird.

Darüber hinaus liefert Definition 3.8 auch eine für die Praxis nutzbare Abschätzung von $d_{maxBaseOffset,MB}$. Der resultierende Ausdruck kann durch Annahme $delay_{min}^{V_{MB}\cup\{v_T\}} = 0$ weiter vereinfacht werden, hierdurch wird die Abschätzung jedoch ungenauer. Analog kann auch $delay_{max}^{V_{MB}\cup\{v_T\}}$, wie in Abschnitt 3.1.1 beschrieben, anhand von (technischen) Maximalwerten abgeschätzt werden, um eine Unabhängigkeit von der konkreten Topologie zu erreichen.

3.2.2.2 Aufbau und Dimensionierung der Mode-Based Time Windows

Als Nächstes definieren wir den Aufbau der *Mode-Based Time Windows* und leiten Constraints her, die es erlauben, ein *Mode-Based Time Window* so zu dimensionieren, dass auch beim Auftreten der maximalen Synchronisationsungenauigkeit $d_{maxBaseOffset,MB}$ gemäß Definition 3.7 die Integrität der angrenzenden Time Windows gewährleistet ist. Die vorgestellten Constraints können sinngemäß auch für die Dimensionierung der Exclusive und Arbitrating Time Windows von TTCAN verwendet werden.

Hierzu führen wir zunächst den Transmission Startpunkt (TSP) ein.

Definition 3.9 (Transmission Startpunkt – TSP)

Jedes Mode-Based Time Window verfügt über einen konfigurierbaren (aber festen) Transmission Startpunkt (TSP). Modusbasierte Knoten mit einem Slot- und Frame-Assignment für das Mode-Based Time Window starten ihre Rahmenübertragung zum TSP (basierend auf deren lokaler Uhr). Der TSP wird über den Parameter d_{TSP} konfiguriert, welcher die Zeitspanne zwischen dem Beginn eines Mode-Based Time Window und dessen TSP festlegt. Hierbei handelt es sich um einen globalen Konfigurationsparameter, der für alle Mode-Based Time Windows und Knoten gilt. Die Abbildung 3.6 illustriert den Aufbau eines Mode-Based Time Window sowie den Start einer Übertragung zum TSP.

Der TTCAN-Standard definiert anstatt eines festen Zeitpunktes für den Übertragungsstart, ein konfigurierbares Zeitfenster (*Tx_Enable* [ISO04, 7.2.2]), innerhalb dessen die Übertragung in einem *Time Window* beginnen darf, verzichtet allerdings auf Vorgaben wie dieses zu konfigurieren ist. Für unsere Realisierung von *Fast Mode-Signaling* ist jedoch ohnehin ein fester Startzeitpunkt (TSP) notwendig, damit alle Knoten *hinreichend zeitgleich* ihre Übertragung starten, sodass sich die SOF-Bits aller konkurrierenden Rahmen auf dem Medium überlagern.

Um sicherzustellen, dass alle Knoten V des TTCAN-Busses den Übertragungsbeginn eines Rahmens (auf Basis ihrer lokalen Uhr) dem selben Time Window zuordnen, muss der Parameter d_{TSP} größer als der maximale Tickoffset $d_{maxBaseOffset}$ (Definition A.2) gewählt werden, der durch die regelmäßige (Re-)Synchronisation (zu Beginn jedes Basiszyklus) von TTCAN zugesichert wird. Die richtige Zuordnung eines Rahmens zu dem jeweiligen Zeitfenster ist essentiell für die korrekte Interpretation und Weiterverarbeitung des empfangenen Rahmens und stellt ebenfalls sicher, dass Übertragungen in angrenzenden Time Windows nicht gestört werden.

Constraint 3.2

Ist $d_{maxOffset}$ der maximale Tickoffset der Knoten V des TTCAN-Busses, welcher durch die regelmäßige (Re-)Synchronisation garantiert wird (vgl. Definition A.1), dann muss bei der Konfiguration d_{TSP} so gewählt werden, dass $d_{TSP} \geq d_{TSP}^{MIN}$ ist, mit $d_{TSP}^{MIN} \equiv d_{maxOffset}$.

Die Abbildung 3.6 zeigt die Struktur eines *Mode-Based Time Window* und illustriert die Übertragung eines Rahmens von dem Sender (S) zu dem Empfänger (E) unter der Annahme, dass $d_{TSP} = d_{maxOffset}$ gewählt wurde. Dargestellt ist eine Übertragung mit dem maximalen Tickoffset $d_{maxOffset}$ zwischen Sender und Empfänger. Des Weiteren bezeichnet $delay(S,E)$ die Signalverzögerung zwischen Sender S und Empfänger E für die dargestellte Übertragung.

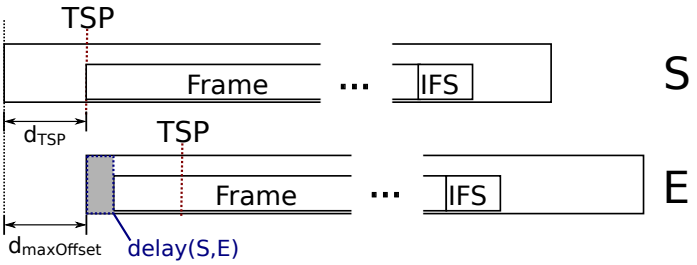


Abbildung 3.6: Rahmenübertragung mit maximalem Tickoffset zwischen Sender (S) und Empfänger (E) (Uhr des Senders eilt voraus).

Um sicherzustellen, dass keine Übertragungen in benachbarten Zeitslots beeinträchtigt werden, muss das *Mode-Based Time Window* so groß gewählt werden, dass auch beim Auftreten des maximalen Tickoffsets $d_{maxOffset}$ und maximaler Signalverzögerung ein Rahmen vollständig innerhalb eines Zeitfensters gesendet und von allen Knoten empfangen werden kann. Hierbei ist insbesondere der Fall zu berücksichtigen, dass die Uhr des Empfängers der des Senders mit dem maximalen Tickoffset $d_{maxOffset}$ vorauseilt. Dies ist in Abbildung 3.7 dargestellt und soll das Verständnis für die nachfolgende Herleitung der minimalen Länge eines *Mode-Based Time Window* erleichtern.

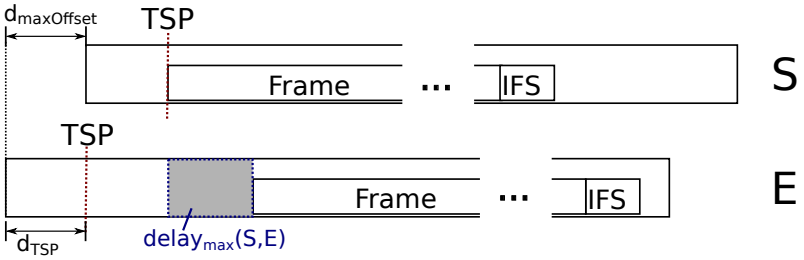


Abbildung 3.7: Rahmenübertragung mit maximalem Tickoffset zwischen Sender (S) und Empfänger (E) (Uhr des Empfängers eilt voraus).

Die minimale Dauer $d_{mtw,s}$ eines *Mode-Based Time Window* $s \in S$ ist abhängig von der Wahl des TSP, der maximalen Signalverzögerung innerhalb des Busses sowie der Länge des zu übertragenden Rahmens.

Constraint 3.3 (Minimale Länge eines *Mode-Based Time Window*)

Gegeben sei ein CAN-Bus mit $V = \{v_1, \ldots, v_s\}$ Knoten. Des Weiteren sei $h_{maxFrame}$ die maximale Länge eines CAN-Rahmens in Bits auf dem Medium, inklusive Interframe Spacing und Bit-Stuffing, d_{Bit} die konfigurierte Bitzeit des CAN-Busses sowie d_{TSP} der konfigurierte TSP.
Ist $s \in S$ ein Mode-Based Time Window, so muss dessen Länge $d_{mtw,s}$ so gewählt werden, dass

$$d_{mtw,s} \geq d_{TSP} + d_{maxOffset} + delay_{max}^{V_s \to V} + h_{maxFrame} \cdot d_{Bit}$$

mit

$$V_s \equiv \{v \in V | \exists m \in M.SA(s,m) = v\}$$

Constraint 3.3 – wie auch Constraint 3.2 – stellt explizit sicher, dass auch für nicht modusbasierte Knoten die Übertragung eines modusbasierten Rahmens innerhalb der Grenzen des jeweiligen *Mode-Based Time Window* erfolgt und die Integrität benachbarter Time Windows gewährleistet ist.

Gelingt es die minimale Signalverzögerung $delay_{min}^{V_s \to V}$ innerhalb des TTCAN-Busses exakt zu bestimmen, so kann diese zur Reduktion der Länge des *Mode-Based Time Window* $s \in S$ genutzt werden. Hierzu starten die Knoten ihre Übertragungen nicht zum TSP, sondern $delay_{min}^{V_s \to V}$ vor dem TSP. Auf diese Weise kann die minimale Länge eines *Mode-Based Time Window* s, im Vergleich zu Constraint 3.3, noch einmal um $delay_{min}^{V_s \to V}$ verkürzt werden. Da die resultierende maximale Einsparung hierdurch jedoch (deutlich) kleiner als eine Bitzeit ist – das Propagation Time Segment muss gemäß Constraint 3.1 doppelt so groß wie die maximale Signalverzögerung sein –, erscheint dieser Optimierungsansatz aufgrund der geringen Ersparnis im Verhältnis zum Aufwand wenig praktikabel.

Wie aus den Abbildungen 3.6 und 3.7 ersichtlich, legt der TSP den Abstand zwischen der Grenze eines Time Window und dem Beginn der Übertragung fest. Durch eine Vergrößerung des Parameters $d_{TSP} > d_{maxOffset}$ lässt sich die Robustheit erhöhen, falls der von TTCAN eigentlich zugesicherte maximale Tickoffset $d_{maxOffset}$ (Definition A.1) aufgrund von Störungen verletzt wird. Dann ist es allerdings sinnvoll, einen solchen zusätzlichen Sicherheitspuffer auch am Ende der Übertragung zu fordern. Dieser kommt zum Tragen, wenn die Uhr des Senders der des Empfängers (unter Verletzung der Zusicherung von TTCAN hinsichtlich des maximalen Tickoffsets, Definition A.1) um mehr als $d_{maxOffset}$ vorauseilt (vgl. Abbildung 3.8). Ein mögliche Umsetzung zeigt Constraint 3.4. Dieses beschreibt eine verschärfte und gleichzeitig praktikabel nutzbare Version des Ausdrucks aus Constraint 3.3. So wird die maximale Signalverzögerung $delay_{max}^{V_s \to V}$ bei der Übertragung eines Rahmens von einem modusbasierten Knoten zu den Empfängern aus Constraint 3.3 mit der maximalen Signalverzögerung $delay_{max}^{V}$ zwischen allen Knoten abgeschätzt. Dies hat darüber hinaus den Vorteil, dass die Menge der Knoten, die modusbasiert in einem Slot kommunizieren (als Sender), beliebig um Knoten aus V erweitert werden kann, ohne dass hierfür die Konfiguration des *Mode-Based Time Window* angepasst werden muss. Wird die maximale Signalverzögerung, wie in Abschnitt 3.1.1 vorgeschlagen, auf Basis der technisch maximal zulässigen Verzögerungen gebildet, vereinfacht dies die Ermittlung und macht die Konfiguration zudem unabhängig von einer konkreten Topologie.

Constraint 3.4 (Länge eines *Mode-Based Time Window*)

Gegeben sei ein CAN-Bus mit $V = \{v_1, \ldots, v_s\}$ Knoten. Des Weiteren sei $h_{maxFrame}$ die maximale Länge eines CAN-Rahmens in Bits auf dem Medium, inklusive Interframe Spacing und Bit-Stuffing sowie d_{Bit} die konfigurierte Bitzeit des CAN-Busses. Zusätzlich sei $delay_{max}^{V}$ die maximale Signalverzögerung zwischen allen Knoten innerhalb des Busses sowie d_{TSP} der konfigurierte TSP gemäß Constraint 3.2.

Ist $s \in S$ ein Mode-Based Time Window, so muss dessen Länge $d_{mtw,s}$ so gewählt werden, dass gilt

$$d_{mtw,s} \geq d_{mtw}^{MIN}$$

mit

$$d_{mtw}^{MIN} \equiv 2 \cdot d_{TSP} + delay_{max}^{V} + h_{maxFrame} \cdot d_{Bit}.$$

Die Berücksichtigung von Constraint 3.4 liegt im Ermessen des Entwicklers und kann in sicherheitskritischen Anwendungen sinnvoll sein, in der besonders viel Wert auf Robustheit gelegt wird. Der Umstand, dass der TSP beliebig groß gewählt werden kann, ermöglicht es, den Sicherheitspuffer für die jeweilige Anwendung geeignet zu dimensionieren.

Abbildung 3.8 zeigt ein entsprechendes Beispiel, bei dem $d_{TSP} > d_{maxOffset}$ gewählt wurde. Die Länge des *Mode-Based Time Window* wurde gemäß Constraint 3.4 auf den minimal zulässigen Wert gesetzt; dargestellt ist eine Übertragung mit der maximal zulässigen Länge $h_{maxFrame}$. Das Beispiel zeigt zwei Knoten, deren Uhren $d_{maxOffset}$ voneinander abweichen und die jeweils einmal als Sender und einmal als Empfänger eines Rahmens agieren. In der Abbildung ebenfalls dargestellt ist die Größe des Sicherheitspuffers. Insbesondere der erste Fall illustriert anschaulich, inwiefern der resultierende Sicherheitspuffer in Abhängigkeit von der Wahl des Parameters d_{TSP} auch in der Lage ist, Tickoffsets größer als $d_{maxOffset}$ zu kompensieren.

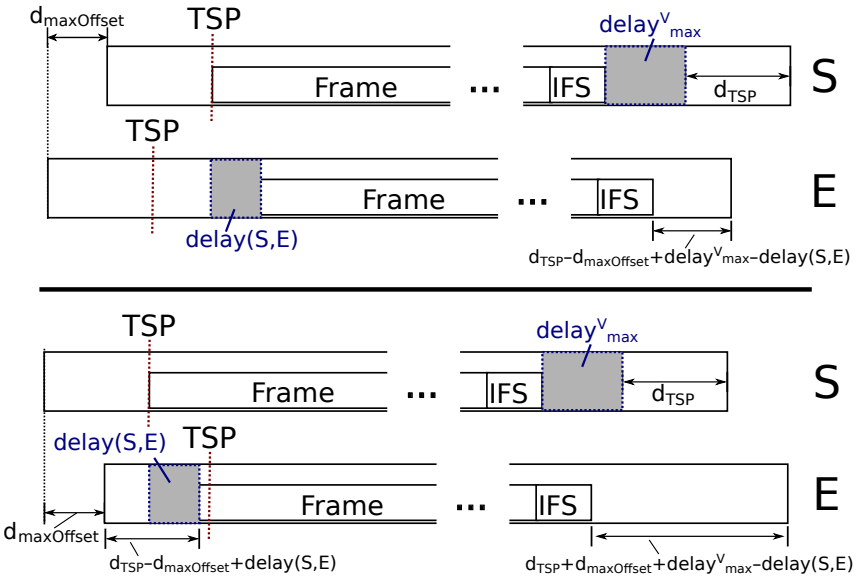


Abbildung 3.8: Rahmenübertragung mit maximalem Tickoffset zwischen Sender (S) und Empfänger (E) und vergrößertem d_{TSP}.

Die Berechnung der minimalen Länge eines *Mode-Based Time Window* gemäß Constraint 3.3 bzw. Constraint 3.4 erfordert die Berücksichtigung der maximalen Übertragungsdauer eines CAN-Rahmens, um die Integrität angrenzender Slots sicherzustellen. Damit dies auch im Falle eines Übertragungsfehlers gewährleistet ist, muss bei der Ermittlung der maximalen Framelänge auch die maximale Länge einer Fehlersignalisierung (bei einem zum spätestmöglichen Zeitpunkt auftretenden Fehler) berücksichtigt werden. Dies ist notwendig, da CAN einen automatischen Mechanismus zur Fehlersignalisierung nutzt,

welcher gemäß CAN-Standard verpflichtend ist. Die Länge eines Rahmens auf dem Medium ist zudem noch abhängig vom Bitstuffing.

Definition 3.10 (Maximale Länge einer Übertragung)

Im Folgenden bezeichne $h_{maxFrame}$ die maximale Anzahl von Bits für die Übertragung eines CAN-Rahmens auf dem Medium unter Worst-Case Bedingungen. Diese beinhalten die Verwendung der maximal zulässigen Nutzlast von 8 Byte, die maximale Anzahl von Stuffing-Bits sowie das spätestmögliche Auftreten eines Übertragungsfehlers mit anschließender Fehlersignalisierung inkl. IFS. Für das Standardrahmenformat gilt $h_{maxFrame} = 158$ Bits und für das erweiterte Rahmenformat $h_{maxFrame} = 183$ Bits.

Für die Ermittlung der maximalen Anzahl von Stuffing-Bits findet man in der Literatur [NSL08, Kapitel 13.2.6] geeignete Formeln. Wendet man diese an, ergibt sich als maximale Länge eines CAN-Datenrahmens mit 8 Byte Nutzlast (inkl. IFS) 135 Bits (bzw. 160 Bits beim erweiterten Rahmenformat).

In Bezug auf die Fehlersignalisierung wird angenommen, dass alle Knoten korrekt arbeiten und eine Störung auf dem Übertragungskanal für eine Verfälschung mit anschließender Fehlersignalisierung verantwortlich ist. Bezogen auf einen spätestmöglich auftretenden Fehler können zwei Fälle unterschieden werden. Im ersten Fall tritt der Fehler noch innerhalb des Rahmens auf und verfälscht das vorletzte Bit[16] des EOF-Feldes, indem ein dominantes Bit anstelle eines rezessiven Bits empfangen wird. Hierauf reagieren die Knoten mit der Übertragung eines Fehlerrahmens, da das Rahmenformat verletzt wird.

Allerdings kann eine Verfälschung innerhalb des IFS zu einer längeren Belegung des Busses führen, deshalb verwenden wir diesen Fall für die Bestimmung von $h_{maxFrame}$. In diesem zweiten Fall empfängt mindestens ein Knoten x ein dominantes Bit innerhalb des IFS. Er interpretiert dies als erstes Bit eines Überlastrahmens (Overload-Frame). Daraufhin beginnt der Knoten im nächsten Bit mit dem Senden eines eigenen Überlastrahmens.

Wir konstruieren den Worst-Case, indem wir annehmen, dass das von x empfangene dominante Bit nur von ihm (aufgrund einer lokalen Störung) empfangen wird. Alle anderen Knoten, die den IFS korrekt empfangen haben, interpretieren die ersten fünf dominanten Bits des Überlastrahmens von Knoten x als die Übertragung eines regulären Rahmens. Da der Überlastrahmen jedoch mit sechs dominanten Bits beginnt, detektieren die Knoten bei der Übertragung des sechsten Bits eine Verletzung der Bit-Stuffing-Regel[17], die für reguläre Rahmen gilt. Daraufhin beginnen die Knoten mit ihrer Fehlersignalisierung.

In dem geschilderten Szenario verlängert sich die Übertragung des ursprünglichen Datenrahmens um 6 Bits bis zum Erkennen des Bitstuffing-Fehlers. Hinzu kommen weitere $6+8+3 = 17$ Bits für die Übertragung des Fehlerrahmens[18]. Dementsprechend ergeben sich die Werte aus Definition 3.10. Diese Abschätzungen sind sehr konservativ. Bei nicht sicherheitskritischen Anwendungen kann auch durchaus mit $h_{maxFrame} = 135$ Bits bzw. $h_{maxFrame} = 160$ Bits gerechnet werden. Im Fehlerfall ist dann jedoch eine Störung des benachbarten Time Window nicht auszuschließen.

3.2.2.3 Obere Schranke für den maximal zulässigen Tickoffset für Fast Mode-Signaling

Die Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* basiert auf der Nutzung des CAN-Arbitrierungsmechanismus für das *Fast Mode-Signaling*. Hierfür müssen in einem *Mode-Based Time Window* alle sendebereiten Knoten mit einem Slot-Assignment für dieses Time Window

[16] Ein dominanter Pegel im letzten Bit des EOF-Feldes wird gemäß [Rob91] ignoriert.

[17] Wir gehen von einer korrekten Übertragung der dominanten Bits des Overload-Frames des Knotens x aus. Würden alle Knoten eines der Bits des Overload-Frames als rezessives Bit interpretieren, käme es zu einer weiteren Verzögerung. Diese Konstellation ist jedoch extrem unwahrscheinlich.

[18] Der Fehlerrahmen (aktives Fehlerflag) besteht aus 6 dominanten Bits (bei aktivem Fehlerflag), 8 rezessiven Bits und einem anschließenden IFS von 3 Bits.

hinreichend zeitgleich ihre Übertragung starten, damit die SOF-Bits kollidieren und sich auf dem Bus überlagern[19]. Durch den CAN-Arbitrierungsmechanismus wird dann der resultierende Zugriffskonflikt aufgelöst und es gewinnt der Knoten mit der höchsten Priorität bzw. höchsten *Modus-Präferenz* aufgrund der verwendeten verträglichen Abbildung der Modes auf CAN-Identifier (Definition 3.6). Dies ist aber nur gewährleistet, wenn die TSPs der Knoten nicht zu stark voneinander abweichen. Ist die Abweichung zu groß kann es sein, dass nicht der Knoten dessen Mode die höchste Präferenz besitzt den Wettbewerb um das *Mode-Based Time Window* gewinnt, weil dieser beim Erreichen seines TSP bereits ein belegtes Medium vorfindet, da ein anderer Knoten seine Übertragung früher gestartet hat. Das heißt, die gewählte Realisierung von *Fast Mode-Signaling* beschränkt die maximal zulässige Abweichung der TSPs der modusbasierten Knoten zueinander. Da die Knoten den TSP anhand ihrer lokalen Uhren bestimmen, führt dies zu einer Beschränkung des maximal zulässigen Tickoffset der modusbasierten Knoten zueinander.

Zur Bestimmung einer oberen Schranke $d^{MAX}_{maxOffset,MB}$ für den maximal zulässigen Tickoffset, welche eine korrekte Funktionsweise von *Fast Mode-Signaling* in allen *Mode-Based Time Windows* garantiert, beschränken wir unsere Betrachtungen zunächst auf ein einzelnes *Mode-Based Time Window* $s \in S$. Für dieses ermitteln wir eine obere Schranke $d^{MAX,s}_{maxOffset,MB}$ für den maximal zulässigen Tickoffset in Abhängigkeit von dem Bit-Timing[20] der Knoten sowie den Slot-Assignments für dieses *Mode-Based Time Window*. Dann schließen wir hieraus auf $d^{MAX}_{maxOffset,MB}$.

Definition 3.11 (Obere Schranke für den maximal zulässigen Tickoffset für *Fast* Mode-Signaling)

Sei $s \in S$ ein Mode-Based Time Window innerhalb des Matrixzklus Z. Ist $v \in V$ ein Knoten, dann bezeichnet $d^{v}_{PropSeg}$ und $d^{v}_{PhaseSeg1}$ die konfigurierte Dauer der entsprechenden Segmente des Bit-Timings dieses Knotens. Damit sichergestellt ist, dass sich die SOF-Bits aller modusbasierten Knoten (mit einem Slot-Assignment für s) überlagern, darf deren Tickoffset zueinander die obere Schranke $d^{MAX,s}_{maxOffset,MB}$ nicht überschreiten. Diese obere Schranke $d^{MAX,s}_{maxOffset,MB}$ wird festgelegt durch,

$$d^{MAX,s}_{maxOffset,MB} \equiv \min_{v_1 \in V_s, v_2 \in V_s} \{delay_{\min}(v_1, v_2) + d^{v_2}_{PropSeg} + d^{v_2}_{PhaseSeg1} | v_1 \neq v_2\} \tag{3.1}$$

mit

$$V_s \equiv \{v \in V | \exists m \in M.\ SA(s,m) = v\}$$

Dann gilt für die obere Schranke $d^{MAX}_{maxOffset,MB}$ des maximal zulässigen Tickoffset aller Mode-Based Time Windows $S_{MBTW} \subseteq S$ eines Matrixzyklus,

$$d^{MAX}_{maxOffset,MB} \equiv \min_{s \in S_{MBTW}} d^{MAX,s}_{maxOffset,MB} \tag{3.2}$$

Das bedeutet die korrekte Funktionsweise des *Fast Mode-Signaling* ist gewährleistet, sofern die TTCAN-Konfiguration so gestaltet wurde, dass der maximale Tickoffset der modusbasierten Knoten vor der (Re-)Synchronisation (Definition 3.7) kleiner als $d^{MAX}_{maxOffset,MB}$ ist. Diese Forderung wird von Constraint 3.5 formuliert.

[19] Somit handelt es sich bei dieser Art der Realisierung um eine Form des aktiven *Fast Mode-Signaling*.

[20] Hierbei berücksichtigen wir ebenfalls, dass die Knoten auch unterschiedliche Konfigurationen hinsichtlich des Bit-Timing aufweisen können.

Constraint 3.5

Sei $d_{maxOffset,MB}$ der maximale Tickoffset der modusbasierten Knoten $V_{MB} \subseteq V$ eines TTCAN-Busses gemäß Definition 3.7. Des Weiteren sei $d^{MAX}_{maxOffset,MB}$ die obere Schranke für den maximal zulässigen Tickoffset für die beschriebene Umsetzung des Fast Mode-Signaling (Definition 3.11). Dann muss die Konfiguration des TTCAN-Busses so gewählt werden, dass

$$d_{maxOffset,MB} < d^{MAX}_{maxOffset,MB}.$$

Bei der Herleitung der oberen Schranke $d^{MAX,s}_{maxOffset}$ des maximal zulässigen Tickoffsets für ein *Mode-Based Time Window s* (Definition 3.11) spielt das Bit-Timing sowie die harte Synchronisation eine entscheidende Rolle. So führt die Erkennung der fallenden Flanke eines SOF-Bits auf einem unbelegten Bus zunächst dazu, dass der entsprechende Knoten eine harte Synchronisation ausführt und sein Bit-Timing mit dem Beginn des Propagation Time Segments startet[21] (vgl. Kapitel 3.1.1). Die harte Synchronisation dient dazu, dass alle Knoten ihr Bit-Timing auf den frühesten Sender synchronisieren[22]. Auch wenn ein Knoten bereits die fallende Flanke eines SOF-Bits empfangen hat, erfolgt die Interpretation des Buspegels erst zum Samplepoint. Daher darf ein Knoten trotz harter Synchronisation auf eine empfangene fallende Flanke noch bis zum Erreichen seines Samplepoint mit der eigenen Übertragung beginnen, da er das Medium bis zum Erreichen des Samplepoint noch nicht als belegt ansieht.

Die Abbildung 3.9 illustriert dies anhand zweier Knoten v_1 und v_2, welche um ein *Mode-Based Time Window* konkurrieren. In dem Beispiel erreicht zunächst Knoten v_1 seinen TSP (gemäß dessen lokaler Uhr) vor v_2 und beginnt mit dem Senden des SOF-Bits, da das Medium zu diesem Zeitpunkt noch frei ist. Die fallende Flanke wird von Knoten v_2 mit einer Verzögerung von $delay(v_1, v_2)$ Zeiteinheiten empfangen. Da eine Abtastung des Mediums nur zu Beginn eines Zeitquantums erfolgt, wird die fallende Flanke erst zum Zeitpunkt t_1 detektiert, woraufhin v_2 eine harte Synchronisation durchführt (vgl. Definition 3.1). Da v_2 den anliegenden Buspegel jedoch erst beim Erreichen des Samplepoints interpretiert, gilt das Medium bis zu diesem Zeitpunkt noch als unbelegt. Das heißt, solange der TSP (gemäß der lokalen Uhr von v_2) vor dem Samplepoint liegt, beginnt v_2 beim Erreichen des TSP sofort mit dem Senden des SOF-Bits seines eigenen Rahmens. Für diesen Fall sind die Zustandsautomaten für den Sende- und Empfangsvorgang innerhalb eines Knotens gekoppelt. Startet ein Knoten, der sich bereits hart auf einen anderen Knoten synchronisiert hat, eine Übertragung, so verwendet dieser das bereits durch die harte Synchronisation gestartete Bit-Timing. Dies stellt sicher, dass das gesendete SOF-Bit nicht zu lange am Bus anliegt und ggf. die Übertragung nachfolgender Bits stört. In Bezug auf das Beispiel auf Abbildung 3.9 sendet v_2 nur ein verkürztes SOF-Bit und beginnt schon zum Zeitpunkt t_2 mit der Übertragung des ersten Bits seines Identifier.

Der Ausdruck (3.1) aus Definition 3.11 bestimmt eine obere Schranke für den maximalen Tickoffset für das *Mode-Based Time Window s*, indem er das beschriebene Szenario aus Abbildung 3.9 nutzt, um für alle möglichen Kombinationen von Knoten mit Slot-Assignments für dieses *Mode-Based Time Window* die kleinste obere Schranke zu bestimmen[23].

[21] Dies gilt natürlich nur, sofern der Knoten vor dem Empfang des SOF-Bits nicht bereits selbst eine Übertragung und damit bereits sein Bit-Timing gestartet hat.

[22] Da die Signalverzögerung zwischen den Knoten im Allgemeinen unterschiedlich ist, können sich verschiedene Knoten auf unterschiedliche Sender synchronisieren. Auch in diesem Fall ist aufgrund der Anforderungen an das Propagation Time Segment (Constraint 3.1) eine ausreichende Überlagerung der (nachfolgenden) Bits gewährleistet.

[23] Hierfür müssen wir von der minimalen Signalverzögerung $delay_{\min}(v_1, v_2)$ für die Übertragung der fallenden Flanke ausgehen und dass diese direkt nach dem Empfang detektiert wird.

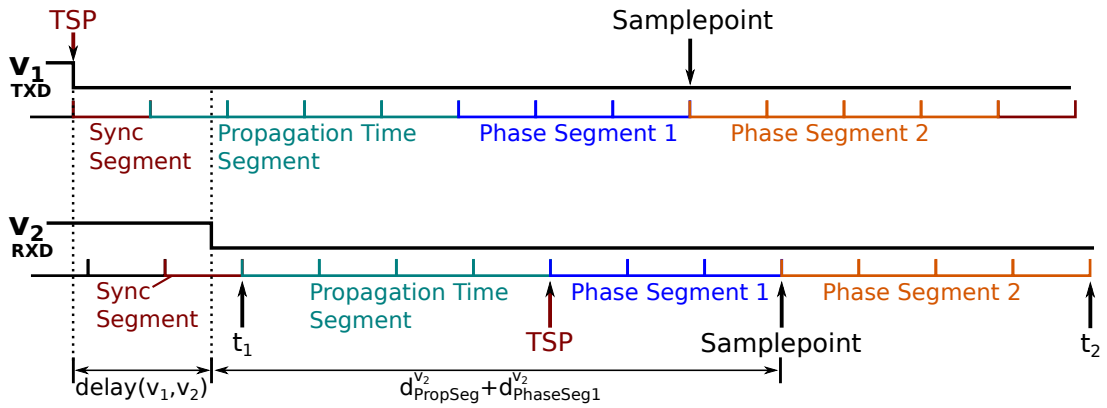


Abbildung 3.9: Knoten v_1 und v_2 übertragen einen konkurrierenden Rahmen, wobei v_2 eine harte Synchronisation auf das von Knoten v_1 gesendete SOF-Bit durchführt.

Die Berechnung der oberen Schranke des maximal zulässigen Tickoffsets für modusbasierte Knoten, anhand von Ausdruck (3.1) aus Definition 3.11, erfordert die Berücksichtigung der minimalen Signalverzögerung und ist daher von der Topologie des Netzes abhängig und aufwändig zu bestimmen. Um dies zu umgehen, setzen wir in Ausdruck (3.1) $delay_{\min}(v_1, v_2) = 0$ und verwenden dies, um Constraint 3.5 zu vereinfachen.

Constraint 3.6

Sei $d_{maxOffset,MB}$ der maximale Tickoffset der modusbasierten Knoten $V_{MB} \subseteq V$ eines TTCAN-Busses gemäß Definition 3.7. Zusätzlich bezeichne $d^v_{PropSeg}$ und $d^v_{PhaseSeg1}$ die konfigurierte Dauer der entsprechenden Segmente des Bit-Timings des Knotens $v \in V$. Damit die korrekte Funktionsweise des Fast Mode-Signaling sichergestellt ist, muss gelten, dass

$$d_{maxOffset,MB} < d^{MAX'}_{maxOffset,MB}$$

mit

$$d^{MAX'}_{maxOffset,MB} \equiv \min_{v \in V_{MB}} (d^v_{PropSeg} + d^v_{PhaseSeg1})$$

Auch wenn das Constraints 3.6 nur Anforderungen an die modusbasierten Knoten hinsichtlich des maximal zulässigen Tickoffsets formuliert, ist es sinnvoll sicherzustellen, dass alle Knoten des TTCAN-Busses diese Anforderungen erfüllen. So sind spätere Änderungen oder Erweiterungen des Ablaufplans ohne Einschränkungen und Modifikation der TTCAN-Konfiguration möglich.

3.3 Implementierung und Evaluation

Nachdem wir uns den Grundlagen für die Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* auf Basis des TTCAN-Protokolls gewidmet haben, wollen wir diese nun im Rahmen eines Prototypen evaluieren. Hierbei dienen die im letzten Kapitel hergeleiteten Constraints dazu, die Konfigurationsparameter eines für *Mode-Based Scheduling with Fast Mode-Signaling* tauglichen Basiszyklus zu bestimmen (Länge des Basiszyklus und der *Mode-Based Time Windows* sowie die Wahl des TSP – Constraints 3.2, 3.3 sowie 3.6).

Für die Evaluation haben wir einen Protokollstack in Software implementiert, welcher *Fast Mode-Signal-*

ing in Verbindung mit *Mode-Based Scheduling* auf Basis des TTCAN-Protokolls realisiert. Technisch basiert unser Prototyp auf einem STM32F407VGT6 ARM Cortex-M4 (STM32F4) Mikrocontroller von STMicroelectronics [STM11]. Dieser Mikrocontroller verfügt über 2 CAN-Controller mit eingeschränkter TTCAN-Unterstützung. Als Ausgangsbasis dient uns ein Evaluation-Board von STMicroelectronics [STM12], welches wir um einen CAN-Transceiver des Typs MCP2551 von Microchip [Mic12b], erweitern. Für die Realisierung musste ein Teil der TTCAN-Funktionalität in Software umgesetzt werden, da zum Zeitpunkt der Evaluation – nach Kenntnisstand des Autors – noch keine CAN-Contoller mit Unterstützung für TTCAN-Level-2 bzw. weiterführender TTCAN-Level-1 Unterstützung auf dem freien Markt verfügbar waren. Der fertige Prototyp wurde anschließend mit Hilfe von Messreihen evaluiert.

3.3.1 Implementierung

Der auf dem Evaluation-Board (Abbildung 3.10) eingesetzte ARM Cortex-M4 Mikrocontroller verfügt über einen integrierten 1-MByte Flashspeicher sowie 192 kB SRAM und wird in unserem Beispiel mit 168 MHz getaktet. Neben mehreren SPI-, I^2C- und UART-Schnittstellen verfügt der Mikrocontroller auch über zwei integrierte CAN-Controller, welche die Anforderungen für TTCAN-Level-1 erfüllen.

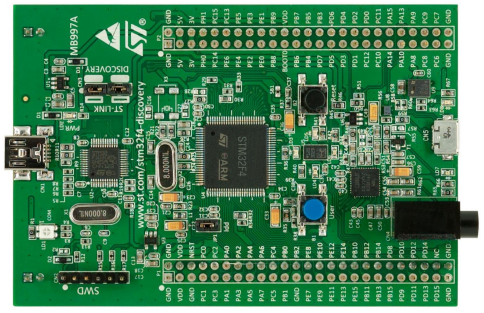


Abbildung 3.10: Das verwendete STM32F4 Evaluation-Board mit ARM Cortex-M4 von STMicroelectronics.

Die integrierten CAN-Controller unterstützen Bitraten bis zu 1 MBit/s und verfügen über einen speziellen zeitgetriggerten Kommunikationsmodus für TTCAN. In diesem Modus ist die automatische, erneute Übertragung von CAN-Rahmen deaktiviert, die ansonsten dafür verantwortlich ist, einen Rahmen bei einem Übertragungsfehler – nach dem Senden des Fehlerrahmens – erneut zu übertragen. Es erfolgt auch kein automatischer erneuter Übertragungsversuch aufgrund einer verlorenen Arbitrierung. Die Verwendung dieses Modus ist essentiell, damit der Protokollstack steuern kann wie oft ein Knoten innerhalb eines Time Window einen Übertragungsversuch durchführt. Nur so kann man sicherstellen, dass die Grenzen der Time Windows nicht verletzt werden.

Der zeitgetriggerte Kommunikationsmodus aktiviert ebenfalls einen 16 Bit-Timer (im Folgenden als CAN-Timer bezeichnet) mit einer Auflösung von einer Bitzeit. Beim Senden eines Rahmens wird zum Zeitpunkt der Übertragung des SOF-Bits der Wert des CAN-Timers in ein Register gesichert. Dies gilt analog für den Empfang eines Rahmens (hier wird der Empfangszeitpunkt des SOF-Bits gespeichert). Leider gibt es keine Möglichkeit den aktuellen Wert des CAN-Timers auszulesen oder diesen als Interruptquelle zu nutzen. Auch das Auslösen eines Übertragungsstarts auf Basis des CAN-Timers wird nicht unterstützt. Die Auflösung des CAN-Timers von einer Bitzeit genügt zwar für die Realisierung von TTCAN-Level-1, nicht aber für die (robuste) Umsetzung von *Fast Mode-Signaling* (vgl. Kapitel 3.2).

Umsetzung der Synchronisation

Um eine höhere Auflösung als eine Bitzeit bei der Synchronisation zu erreichen, konfigurieren wir den Controller so, dass dieser einen Interrupt beim Empfang bzw. beim Versand eines CAN-Rahmens auslöst und implementieren auf dieser Basis die für TTCAN benötigte Ticksynchronisation. Als lokale Uhr des Knotens dient ein 32-Bit Hardware-Timer des STM32F4. Beim Empfang eines Rahmens wird innerhalb der Interruptroutine von den Knoten geprüft, ob es sich bei dem empfangenen Rahmen um eine Referencemessage handelt; ist dies der Fall, setzen die Knoten diesen Timer auf Null zurück. Zusätzlich wird der Timer so konfiguriert, dass dieser einen Interrupt auslöst, sobald ein Time Window beginnt für das der Knoten eine Slotzuordnung besitzt. Der Startzeitpunkt der Time Windows innerhalb des Basiszyklus wird in unserem Protokollstack relativ zu dem Empfang der Referencemessage definiert.

Der Time Master als Sender der Referencemessage empfängt diese selbst nicht und daher wird natürlich auch kein entsprechender Empfangsinterrupt ausgelöst. Dies erschwert die Synchronisation des Time Masters mit den anderen Knoten des TTCAN Busses. Daher verwenden wir für den Time Master einen Interrupt, welcher nach dem Versand einer Nachricht ausgelöst wird. Der hierüber ermittelte Zeitpunkt für den Versand der Referencemessage dient dann als Referenztick für den Time Master.

Die Genauigkeit unserer Synchronisation haben wir mittels eines Testszenarios, bestehend aus einem CAN-Bus mit vier CAN-Knoten $V = \{v_m, v_1, v_2, v_3\}$ und einer Übertragungsrate von 500 kBit/s – und dementsprechend einer Bitzeit von 2 µs – evaluiert. Der Knoten v_m übernimmt die Rolle des Time Masters, der die Knoten v_1, v_2 und v_3 synchronisiert (diese bezeichnen wir im Folgenden auch als Slaves).

Um eine Tendenz bezüglich der Synchronisationsgenauigkeit für unsere Implementierung zu erhalten, haben wir für insgesamt 49 durchgeführte Synchronisationen den maximalen Baseoffset[24] für jede einzelne Synchronisation (einmal auf alle Knoten und einmal auf die Slaves bezogen) ermittelt. Hierzu wurde der Protokollstack so erweitert, dass die Knoten bei der Durchführung der Synchronisation (Zurücksetzen ihres Timers) jeweils einen Pin auf High setzen. Die Zeitpunkte zu denen dies erfolgt, wurden mittels eines Logic-Analyser mit einer Abtastrate von 200 MHz für alle vier Knoten aufgezeichnet. Aus diesen Messwerten lässt sich für jede durchgeführte Synchronisation der aufgetretene maximale Baseoffset bezogen auf eine beliebige Menge von Knoten $V' \subseteq V$ bestimmen.

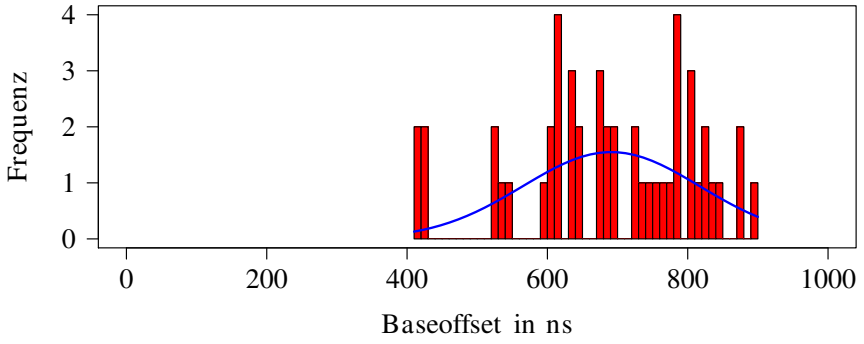
Die beiden Histogramme 3.11 zeigen die Häufigkeitsverteilung des maximalen Baseoffset jeweils bezogen auf unterschiedliche Mengen von Knoten. Während das Histogramm 3.11a die Verteilung der maximalen Baseoffsets der Messreihe bezogen auf alle vier Knoten zeigt, beschränkt sich das Histogramm 3.11b auf die Darstellung der Verteilung der maximalen Baseoffsets der Messreihe bezogen auf die Slaves[25]. Bei Berücksichtigung des Time Masters (Abbildung 3.11a) liegen die gemessenen maximalen Baseoffsets in einem Intervall zwischen 410 ns und 900 ns. Betrachtet man nur die Slaves, so beschränken sich die beobachteten maximalen Baseoffsets auf ein Intervall zwischen 50 ns und 400 ns. Wie das Ergebnis zeigt führt die Synchronisation des Time Masters – sodass dieser als Sender an der (modusbasierten) Kommunikation teilnehmen kann – letztendlich zu einer Reduktion der Synchronisationsgenauigkeit aller Knoten. Der Einfluss des Propagation-Delay auf die beobachteten Verzögerungen ist bei dem Versuchsaufbau zu vernachlässigen, da die Gesamtlänge des Busses nur ca. 50 cm betrug.

Um zu überprüfen, ob für den höheren maximalen Baseoffset bei Berücksichtigung des Time Masters konstante Anteile verantwortlich sind, welche sich kompensieren lassen, untersuchen wir die Messergebnisse noch einmal im Hinblick auf diesen Aspekt. Hierbei stellte sich heraus, dass der Time Master das Übertragungsende stets als erster der vier Knoten detektiert. Das Histogramm 3.12 zeigt die Verteilung der Baseoffsets des Time Masters zu jedem seiner Slaves über alle Synchronisationen hinweg, d.h. pro durchgeführter Synchronisation enthält das Histogramm drei Messwerte (die Baseoffsets des

[24] Der maximale paarweise Offset zwischen den betrachteten Knoten bei einer (Re-)Synchronisation unmittelbar nach dem Empfang der Referencemessage.

[25] Das heißt, die Abweichung zwischen dem Time Master und seinen Slaves wird bei der Ermittlung der maximalen Baseoffsets der einzelnen Synchronisationen nicht berücksichtigt.

(a) Verteilung der maximalen Baseoffsets bezogen auf eine Synchronisation und alle Knoten (Time Master und Slaves) des CAN-Busses.



(b) Verteilung der maximalen Baseoffsets bezogen auf eine Synchronisation und alle Slaves (ohne den Time Master).

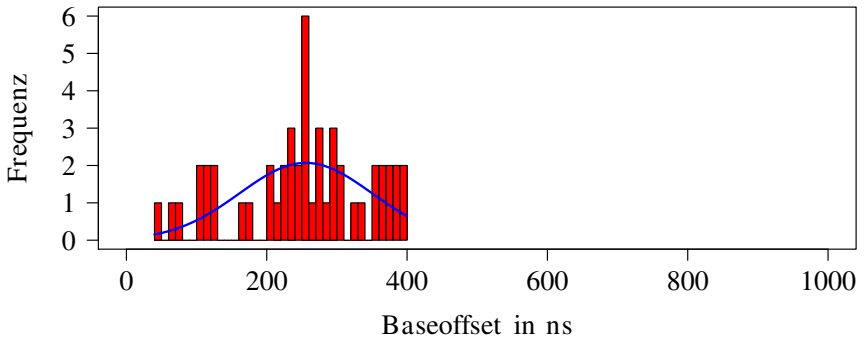


Abbildung 3.11: Häufigkeitsverteilung der maximalen Baseoffsets bezogen auf die durchgeführten Synchronisationen.

Time Masters zu Knoten $v1$, $v2$ und v_3). Der Baseoffset zwischen dem Time Master und seinen Slaves schwankt in einem Intervall zwischen 40 ns und 900 ns. Dieser große Schwankungsbereich deutet darauf hin, dass hierfür kaum konstante Anteile verantwortlich sind und somit auch keine einfache Korrektur des Baseoffsets zwischen dem Time Master und dessen Slaves möglich ist. Eine potentielle Ursache für einen Teil der beobachteten Schwankungen können die durch die CAN-Transceiver verursachten Signalverzögerungen sein. Das Datenblatt gibt für die verwendeten CAN-Transceiver eine Signalverzögerung zwischen 130 ns und 250 ns an [Mic12b] – in der Praxis dürfte die Abweichung zwischen den einzelnen Transceivern, jedoch aufgrund gleicher Umgebungsbedingungen deutlich geringer ausfallen und weniger stark schwanken. Somit bleibt als wahrscheinlichster Einflussfaktor eine unterschiedliche Realisierung der eingesetzten Interrupts für die Signalisierung des Übertragungsendes bei Sender (Time Master) und Empfängern (Slaves) durch den CAN-Controller. Dies ist insofern relevant, da diese Interrupts als Grundlage für die Synchronisation dienen und es sich hierbei um den einzigen relevanten Unterschied in der Implementierung der Synchronisation zwischen Time Master und Slaves handelt. Leider liefert das Datenblatt des STM32F107 [STM11] hierzu keine näheren Informationen, zu welchem Zeitpunkt

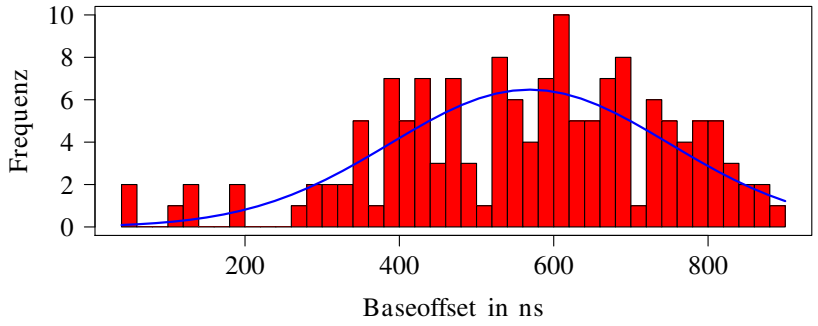


Abbildung 3.12: Verteilung der Baseoffsets des Time Masters zu dessen Slaves über alle Synchronisationen der Messreihe.

innerhalb der Übertragung die jeweiligen Interrupts exakt ausgelöst werden und inwiefern zusätzliche Randbedingungen und Verzögerungen hierbei zu berücksichtigen sind.

Da letztendlich nicht eindeutig geklärt werden konnte wie diese Verzögerung zu Stande kommt und die große Schwankungsbreite eine Kompensation unmöglich macht, haben wir uns bei der Realisierung unseres Prototypen dazu entschieden die Aufgabe des Time Masters einem dedizierten Knoten zuzuordnen, welcher keine modusbasierte Kommunikation durchführt (d.h. keine Slotzuordnung für einen *Mode-Based Time Window* besitzt). Dadurch gefährdet die ungenaue Synchronisation des Time Masters mit den anderen Knoten des TTCAN-Busses nicht die korrekte Funktionsweise des *Fast Mode-Signaling*.

Ablaufsteuerung innerhalb der Mode-Based Time Windows

Der Ablauf bei der Übertragung eines Rahmens innerhalb eines *Mode-Based Time Window* wurde bereits in Kapitel 3.2 thematisiert; alle modusbasierten Knoten mit einer Slotzuordnung, die in dem jeweiligen Slot einen Rahmen senden möchten, starten ihre Übertragung zum TSP. Basierend auf der Abbildung der Modes (*Modus-Präferenz*) auf die CAN-Identifier wird das *Fast Mode-Signaling* durch den CAN-Arbitrierungsmechanismus umgesetzt. Damit die korrekte Funktionalität des *Fast Mode-Signaling* gewährleistet ist, müssen die in Kapitel 3.2.2 beschriebenen Constraints hinsichtlich des maximal zulässigen Tickoffsets erfüllt werden. Hierin besteht die eigentliche Herausforderung bei der Entwicklung des Protokollstacks, eine möglichst genaue Synchronisation zu implementieren und Übertragungen möglichst exakt zum TSP zu starten.

Bevor ein CAN-Rahmen übertragen wird, muss dieser zunächst in den Speicher des CAN-Controllers des STM32F4 übertragen werden. Hierfür stehen spezielle Registersätze für ausgehende CAN-Nachrichten bereit (insgesamt drei sogenannte Mailboxes). Eine Mailbox kann eine CAN-Nachricht (charakterisiert durch Nachrichtentyp, Identifier und Nutzdaten) beliebig lange zwischenspeichern. Erst das Setzen eines spezifischen Kontrollbits veranlasst die Übertragung durch den CAN-Controller.

Bei der modusbasierten Kommunikation wird der CAN-Identifier einer Nachricht in Abhängigkeit von dem aktuellen *Mode-Based Time Window*, dem Mode der Nachricht sowie der verwendeten *verträglichen* Abbildung der Modes auf CAN-Identifier ermittelt. Um zu verhindern, dass die Selektion des zu übertragenden Rahmens, die Bestimmung des Identifier sowie das Ablegen der CAN-Nachricht in der Mailbox den Übertragungsbeginn verzögern, werden diese Operationen rechtzeitig vor dem TSP ausgeführt. Abbildung 3.13 illustriert dies. Bereits vor dem TSP wird ein Timer-Interrupt (zum Zeitpunkt t_0) ausgelöst welcher prüft, ob eine Nachricht für den nachfolgenden modusbasierten Slot zur Übertragung

vorliegt. Liegt keine Nachricht vor wird der Timer so konfiguriert, dass er zum Zeitpunkt t_1 (Start des *Mode-Based Time Window*) auslöst. Soll im kommenden *Mode-Based Time Window* eine Nachricht übertragen werden, wird zuerst der Identifier der Nachricht ermittelt und diese anschließend in der Mailbox abgelegt. Dann wird der Timer so konfiguriert, dass beim Erreichen des TSP (Zeitpunkt t_2) ein weiterer Timer-Interrupt ausgelöst wird, in dem die eigentliche Übertragung der hinterlegten CAN-Nachricht angestoßen wird. Diese wird dann durch den CAN-Controller im Hintergrund ausgeführt und erfordert keine weitere Softwareinteraktion.

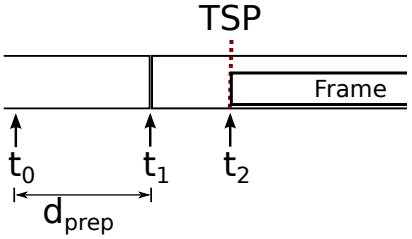


Abbildung 3.13: CAN-Übertragung in einem *Mode-Based Time Window* mit Hilfe von Timer-Interrupts.

Der Konfigurationsparameter $d_{prep} \geq 0$ legt relativ zum Start des *Mode-Based Time Window* fest, wann mit der Vorbereitung einer Nachricht begonnen wird. Falls in dem Slot kein Rahmen versendet werden soll, wird ein Timer-Interrupt zum Zeitpunkt t_1 ausgelöst, um interne Datenstrukturen des Protokollstacks zu aktualisieren (z. B. den Slotzähler zu inkrementieren). Ansonsten erfolgt diese Aktualisierung der Datenstrukturen zum Zeitpunkt t_2, nach dem Start der Übertragung.

Alle zeitkritischen Operationen des Protokollstacks werden im Interruptkontext unterbrechungsfrei ausgeführt. Dies gilt für die (Re-)Synchronisation, die durch einen Empfangsinterrupt der Referencemessage ausgelöst wird, das Laden eines Rahmens in die Mailbox sowie den eigentlichen Start der Übertragung. Die Anwendungen, welche den Protokollstack nutzen laufen hingegen i.d.R. im Nicht-Interrupt-Kontext und können daher für diese zeitkritischen Operationen unterbrochen werden, sodass diese mit minimalen Verzögerungen ausgeführt werden. Nutzt die Anwendung selbst Interrupts sollte diesen eine niedrigere Priorität zugewiesen werden als jenen, die der Protokollstacks verwendet.

Schnittstelle zwischen Anwendung und Protokollstack

Um Anwendungslogik und zeitkritische Protokolllogik voneinander zu isolieren, kommen spezielle Nachrichtenpuffer, sogenannte *Mode-based Transmission Buffer*, zum Einsatz. Ein *Mode-based Transmission Buffer* ist einem Mode fest zugeordnet und stellt eine FIFO-Warteschlange bereit, in der die zu versendenden Nutzdaten zwischengespeichert werden. Wie die Modes kann auch ein *Mode-based Transmission Buffer* pro Matrixzyklus mehreren *Mode-Based Time Windows* zugeordnet werden, ein entsprechendes Slot-Assignment vorausgesetzt. Über zusätzliche Konfigurationsoptionen kann die maximale Anzahl von Übertragungsversuchen, die Semantik des Puffers (s.u.), die maximale Größe der Warteschlange sowie die Ersetzungsstrategie[26] festgelegt werden.

Um eine Nachricht mit einem bestimmten Mode für die Übertragung einzuplanen, kopiert die Anwendung die zu versendenden Nutzdaten lediglich in den *Mode-based Transmission Buffer*, welcher mit dem entsprechenden Mode verknüpft ist. Kurz vor Beginn eines *Mode-Based Time Window* (Zeitpunkt t_0) werden zunächst alle *Mode-based Transmission Buffer* bestimmt, welche mit diesem Time Window assoziiert sind (Slot-Assignment). Unter diesen wird derjenige nicht-leere Puffer, dessen Mode mit der höchsten *Modus-Präferenz* für dieses *Mode-Based Time Window* verknüpft ist, ausgewählt. Dann werden

[26] Die Ersetzungsstrategie legt fest, wie beim Überschreiten der maximalen Warteschlangenlänge mit den neuen Nachrichten (bzw. deren Nutzdaten) verfahren wird.

die in dem Puffer abgelegten Nutzdaten[27] zusammen mit dem ermittelten Identifier in die Mailbox des CAN-Controllers kopiert. Beim Erreichen des TSP wird dann die Übertragung gestartet. Falls diese fehlschlägt (aufgrund einer Störung oder einer verlorenen Arbitrierung), erfolgt ein erneuter Übertragungsversuch in dem nächsten mit dem Puffer assoziierten *Mode-Based Time Window*. Dies wird solange wiederholt, bis die maximale Anzahl von Übertragungsversuchen erreicht oder die Nachricht aufgrund der gewählten Ersetzungsstrategie verdrängt wurde. Bei einer erfolgreichen Übertragung werden die Daten aus dem Puffer entfernt (*singleshot*-Semantik) oder solange erneut übertragen, bis der Inhalt des Puffers durch andere Daten überschrieben wird (*continuous*-Semantik).

Eingehende Rahmen werden in einer zentralen FIFO-Warteschlange innerhalb des Protokollstacks verwaltet. Neben den Nutzdaten der empfangenen CAN-Rahmen werden zusätzliche Metainformationen (Empfangszeitstempel, Time Window, in dem der Rahmen empfangen wurde, sowie dessen Mode[28]) gespeichert und können durch die Anwendung abgerufen werden.

3.3.2 Szenario

Um unseren Protokollstack funktional zu testen, verwenden wir den gleichen Aufbau wie bei der Evaluation des Baseoffsets. Die vier CAN-Knoten $V = \{v_M, v_1, v_2, v_3\}$ benutzen eine identische CAN-Konfiguration. Der Knoten v_M übernimmt die Aufgabe des Time Masters und sendet die Referencemessage zur Synchronisation der Knoten v_1, v_2 und v_3. Aufgrund der Schwierigkeiten bei der Synchronisation des Time Masters mit den anderen Knoten des TTCAN-Busses (vgl. Kapitel 3.3.1) sendet der Time Master selbst keine modusbasierten Rahmen. Stattdessen protokolliert der Knoten alle empfangenen CAN-Rahmen und vergleicht deren Abfolge und Modes mit der erwarteten Abfolge von Rahmen (vgl. Tabelle 3.2, 3.3 und 3.4). Anhand des für jeden Basiszyklus identisch gewählten Frame-Assignment (welches die Rahmen festlegt, die von den einzelnen Knoten zur Laufzeit für die Übertragung eingeplant werden) können Fehler automatisch erkannt sowie auf dieser Basis eine Fehlerrate bestimmt und ausgegeben werden. Um insbesondere die Zuverlässigkeit des *Fast Mode-Signaling* zu testen (der zeitkritischsten Komponente der Implementierung), versuchen in diesem Szenario alle modusbasierten Knoten in allen *Mode-Based Time Windows* – für die die Knoten über Slotzuordnungen verfügen – Rahmen zu übertragen. Dies führt dazu, dass in jedem *Mode-Based Time Window* mindestens zwei Knoten bzw. Rahmen um den Medienzugriff konkurrieren.

Unsere Tests und Messungen wiederholen wir mit zwei verschiedenen Übertragungsraten (250 kBit/s und 500 kBit/s), um sowohl die abgeleiteten Constraints als auch die Funktionalität unserer Implementierung unter verschiedenen Randbedingungen zu testen. Der verwendete Basiszyklus besteht aus 14 bzw. 9 gleich großen Time Windows. In beiden Fällen wird jeweils das Time Window 0 exklusiv für die Übertragung der Referencemessage durch den Time Master v_M genutzt. Die Tabelle 3.1 zeigt die verwendeten CAN- und TTCAN-Konfigurationsparameter. Um zu zeigen, dass die gewählte Konfiguration konform mit den Definitionen und Constraints aus Kapitel 3.2.2 für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* sind, wurden diese ebenfalls in der Tabelle aufgeführt.

Die Zeitquanten werden, unter Verwendung eines Vorteilers, direkt aus dem Takt des Prozessors abgeleitet. Zu Anschauungszwecken haben wir in der Tabelle die Darstellung im Zeitbereich einer Frequenzangabe vorgezogen. Die hieraus resultierenden Ungenauigkeiten in der Tabelle entstammen der Umrechnung und treten daher bei der Verarbeitung innerhalb des CAN-Controllers nicht auf. In beiden Konfigurationen lassen sich, mit Hilfe des durch den Vorteiler abgeleiteten Taktes die jeweiligen Bitzeiten, Zeitsegmente und Zeitquanten exakt als ganzzahliges Vielfaches dieses Taktes darstellen.

[27] Der *Mode-based Transmission Buffer* speichert lediglich die Nutzdaten, der CAN-Identifier muss für jedes *Mode-Based Time Window*, auf Basis der definierten Abbildung der Modes auf CAN-Identifier, individuell bestimmt werden.

[28] Der Mode wird anhand des Identifier sowie dem Time Window auf Basis der verwendeten Abbildung der Modes auf Identifier ermittelt.

		250 kBit/s	500 kBit/s	
CAN	Bitzeit d_{BIT}	$4\mu s$	$2\mu s$	
	Takt des CAN-Controllers d_{CLK}	42 MHz	42 MHz	
	Vorteiler zum Ableiten der Zeitquanten	7	4	
	Anzahl der Zeitquanten pro Bitzeit	24	21	
	Länge eines Zeitquantums d_q	$0,1667\mu s$	$0,095\mu s$	
	Länge des Synchronization Segment	$1 \cdot d_q = 0,1667\mu s$	$1 \cdot d_q = 0,095\mu s$	
	Länge des Propagation Time Segment	$15 \cdot d_q = 2,5005\mu s$	$13 \cdot d_q = 1,235\mu s$	
	Länge des Phase Segment 1	$1 \cdot d_q = 0,1667\mu s$	$1 \cdot d_q = 0,095\mu s$	
	Länge des Phase Segment 2	$7 \cdot d_q = 1,1669\mu s$	$6 \cdot d_q = 0,57\mu s$	
TTCAN	Länge des Basiszyklus d_{Cycle}	9,296 ms	3,320 ms	
	Anzahl der Time Windows	14	10	
	Länge der Time Windows	$166 \cdot d_{BIT}$	$166 \cdot d_{BIT}$	
	Transmission Startpunkt d_{TSP}	$5 \cdot d_{BIT}$	$5 \cdot d_{BIT}$	
	d_{prep}	$9 \cdot d_{TSP}$	$19 \cdot d_{TSP}$	
Constraints	Definition 3.1.1: $delay^{V}_{max}$	$0,415\mu s$	$0,415\mu s$	**Definitionen**
	Definition 3.1.1: $delay^{V}_{min}$	$0\mu s$	$0\mu s$	
	Definition 3.7: $d_{maxOffset,MB}$	$2,441\mu s$	$1,174\mu s$	
	Constraint 3.6: $d^{MAX'}_{maxOffset,MB}$	$2,6672\mu s$	$1,33\mu s$	
	Constraint 3.2: d^{MIN}_{TSP}	$2,441\mu s$	$1,174\mu s$	
	Constraint 3.4: d^{MIN}_{mtw}	$625,3\mu s$	$312,763\mu s$	
		$= 156,32 \cdot d_{BIT}$	$= 156,4 \cdot d_{BIT}$	

Tabelle 3.1: Konfigurationsparameter und berechnete Constraints für das entworfene Szenario.

Die untere Hälfte der Tabelle 3.1 (grau hinterlegt) wendet die Definition und Constraints aus Kapitel 3.2.2 an, um abgeleitete Größen zu bestimmen, die eine Beurteilung der Tauglichkeit der Konfiguration hinsichtlich *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* gestatten. Hierzu gehören die minimale und maximale Signalverzögerung[29] (Definition 3.1.1) ebenso wie der maximale Tickoffset, welcher aufgrund der (Re-)Synchronisation bei dem gewählten Basiszyklus gemäß Definition 3.7 auftreten kann. In dunkelgrau hinterlegt sind die von den Constraints für die Konfiguration festgelegten Randbedingungen abgedruckt, welche eingehalten werden müssen, um z. B. die korrekte Funktionalität des *Fast Mode-Signaling* sicherzustellen (Constraint 3.6). Natürlich wurden die Konfigurationsparameter so gewählt, dass die Constraints erfüllt sind. Hinsichtlich der Wahl des TSP haben wir die Freiheiten gemäß Constraint 3.2 genutzt und den Parameter bewusst größer gewählt, um die Konfiguration robuster zu gestalten.

Unser Szenario definiert die Modes *Emergency*, *Safety* und *Regular*. Die Slotzuordnungen wurden so gewählt, dass pro *Mode-Based Time Window* mindestens zwei, einmal sogar alle drei Modes, um das Medium konkurrieren können. Von den 14 bzw. 9 Time Windows sind vier *Mode-Based Time Windows*. Die verwendeten *Modus-Präferenzen* und *Slot-Assignments* sind in den Tabellen 3.2 sowie 3.3 dargestellt. Das *Frame-Assignment* (Tabelle 3.4) gilt für alle Basiszyklen und wurde mit Absicht so gestaltet, dass alle

[29] Bei der Bestimmung der maximalen Signalverzögerung gehen wir von einer internen Verarbeitungszeit des CAN-Controllers von 75 ns und einem Propagation-Delay von 5 ns aus. Ausgehend von einem Temperaturbereich von $-40°$C bis 85°C liegt die maximale Verzögerung der CAN-Transceiver (laut Datenblatt) bei 260 ns [Mic12b] sowie der maximale Clock Skew des verwendeten Quarzes bei $r_{maxClockSkew} = 100$ ppm [Jau05, Mtr05]. Zum Vergleich, bei Zimmertemperatur liegen die Toleranzen der Quarze im Bereich von 30 ppm bis 50 ppm [BRW07, Joh92]. Die minimale Signalverzögerung haben wir zur Sicherheit mit 0 µs abgeschätzt.

Knoten im letzten Time Window des Basiszyklus miteinander konkurrieren[30], da zu diesem Zeitpunkt aufgrund des Clock Skew die größte Uhrenabweichung zu erwarten ist[31]. Somit ist in diesem Time Window auch die Wahrscheinlichkeit am größten, dass das *Fast Mode-Signaling* nicht korrekt funktioniert. Die gewählte Abbildung der Modes (*Modus-Präferenzen*) auf CAN-Identifier ist neben den *Modus-Präferenzen* in Tabelle 3.3 dargestellt. Wie direkt ersichtlich ist die verwendete Abbildung *verträglich* mit der *Modus-Präferenz*-Funktion (vgl. Definition 3.6).

SA	s_2	s_4	s_6	s_9 bzw. s_{13}
Emergency	v_2			v_1
Safety	v_1	v_2	v_3	v_2
Regular		v_3	v_1	v_3

Tabelle 3.2: Die in dem Szenario verwendete *Slot-Assignment*-Funktion für die *Mode-Based Time Windows*.

MP $\rightarrow$ CAN-Identifier	s_2	s_4	s_6	s_9 bzw. s_{13}
Emergency	$0 \rightarrow 12$			$0 \rightarrow 12$
Safety	$1 \rightarrow 13$	$0 \rightarrow 12$	$0 \rightarrow 12$	$1 \rightarrow 13$
Regular		$1 \rightarrow 13$	$1 \rightarrow 13$	$2 \rightarrow 14$

Tabelle 3.3: Definition der *Modus-Präferenzen* und verträgliche Abbildung der Modes auf CAN-Identifier.

FA		$s_{x,2}$	$s_{x,4}$	$s_{x,6}$	$s_{x,9}$ bzw. $s_{x,13}$
Emergency	v_1				$f_{1,3}$
Emergency	v_2	$f_{2,1}$			
Safety	v_1	$f_{1,1}$			
Safety	v_2		$f_{2,2}$		$f_{2,3}$
Safety	v_3			$f_{3,2}$	
Regular	v_1			$f_{1,2}$	
Regular	v_2				
Regular	v_3		$f_{3,1}$		$f_{3,3}$

Tabelle 3.4: Das in jedem Basiszyklus $x \in S$ verwendete Frame-Assignment des Szenarios.

In unserem Beispiel haben wir die Anzahl der eingesetzten CAN-Identifier für modusbasierte Übertragungen minimiert. So wird der gleiche CAN-Identifier für unterschiedliche Nachrichtentypen und von unterschiedlichen Knoten benutzt. Der CAN-Identifier 12 wird zum Beispiel, von allen drei Knoten für die Übertragung von Rahmen mit unterschiedlichen Modes in unterschiedlichen Time Windows eingesetzt. Ein Umstand, welcher nicht konform mit dem CAN-Standard ist, bei *Mode-Based Scheduling with Fast*

[30] Es sind jeweils diejenigen Rahmen mit grau hinterlegt, denen innerhalb des Time Window die höchste *Modus-Präferenz* zugeordnet wurden und die daher den Wettbewerb gewinnen (müssen). Ein Eintrag $f_{i,j}$ steht für einen eingeplanten Rahmen, des Knotens i. Der Index j nummeriert die eingeplanten Rahmen eines Knotens fortlaufend.

[31] Natürlich könnte der Clock Skew auch der Uhrenabweichung entgegenwirken, dass der Tickoffset zwischen allen Knoten hierdurch jedoch kleiner ist als unmittelbar nach der Synchronisation ist unwahrscheinlich.

Listing 3.1: Logausgaben des Sniffers bei unterschiedlich aktivierten Knoten.

```
1  Knoten 1:
   276388: Slot 2, CycleNo: 132, Frame 13, Bit-Timestamp 49025, Mode Safety
3  834148: Slot 6, CycleNo: 132, Frame 13, Bit-Timestamp 49689, Mode Regular
   1810172: Slot 13, CycleNo: 132, Frame 12, Bit-Timestamp 50851, Mode Emergency
5
   Knoten 2:
7  278342: Slot 2, CycleNo: 37, Frame 12, Bit-Timestamp 62981, Mode Emergency
   558006: Slot 4, CycleNo: 37, Frame 12, Bit-Timestamp 63314, Mode Safety
9  1812070: Slot 13, CycleNo: 37, Frame 13, Bit-Timestamp 64808, Mode Safety

11 Knoten 3:
   555454: Slot 4, CycleNo: 54, Frame 13, Bit-Timestamp 34284, Mode Regular
13 835174: Slot 6, CycleNo: 54, Frame 12, Bit-Timestamp 34616, Mode Safety
   1808734: Slot 13, CycleNo: 54, Frame 14, Bit-Timestamp 35778, Mode Regular
15
   Knoten 1 und 2:
17 279478: Slot 2, CycleNo: 107, Frame 12, Bit-Timestamp 29121, Mode Emergency
   558302: Slot 4, CycleNo: 107, Frame 12, Bit-Timestamp 29453, Mode Safety
19 835446: Slot 6, CycleNo: 107, Frame 13, Bit-Timestamp 29785, Mode Regular
   1811470: Slot 13, CycleNo: 107, Frame 12, Bit-Timestamp 30947, Mode Emergency
21
   Knoten 1 und 3:
23 276574: Slot 2, CycleNo: 164, Frame 13, Bit-Timestamp 34473, Mode Safety
   556294: Slot 4, CycleNo: 164, Frame 13, Bit-Timestamp 34805, Mode Regular
25 835846: Slot 6, CycleNo: 164, Frame 12, Bit-Timestamp 35137, Mode Safety
   1810078: Slot 13, CycleNo: 164, Frame 12, Bit-Timestamp 36299, Mode Emergency
27
   Knoten 2 und 3:
29 279320: Slot 2, CycleNo: 84, Frame 12, Bit-Timestamp 26270, Mode Emergency
   558144: Slot 4, CycleNo: 84, Frame 12, Bit-Timestamp 26602, Mode Safety
31 837024: Slot 6, CycleNo: 84, Frame 12, Bit-Timestamp 26934, Mode Safety
   1812264: Slot 13, CycleNo: 84, Frame 13, Bit-Timestamp 28096, Mode Safety
33

35 Knoten 1, 2 und 3:
   278820: Slot 2, CycleNo: 11, Frame 12, Bit-Timestamp 35530, Mode Emergency
37 557588: Slot 4, CycleNo: 11, Frame 12, Bit-Timestamp 35862, Mode Safety
   836468: Slot 6, CycleNo: 11, Frame 12, Bit-Timestamp 36194, Mode Safety
39 1810700: Slot 13, CycleNo: 11, Frame 12, Bit-Timestamp 37356, Mode Emergency
   279014: Slot 2, CycleNo: 12, Frame 12, Bit-Timestamp 38021, Mode Emergency
41 557726: Slot 4, CycleNo: 12, Frame 12, Bit-Timestamp 38353, Mode Safety
   836494: Slot 6, CycleNo: 12, Frame 12, Bit-Timestamp 38685, Mode Safety
43 1810726: Slot 13, CycleNo: 12, Frame 12, Bit-Timestamp 39847, Mode Emergency
```

Mode-Signaling für TTCAN jedoch keine negativen Auswirkungen hat, solange sichergestellt wird, dass ein Mode in jedem *Mode-Based Time Window* nur maximal einem Knoten zugeordnet ist.

Das Listing 3.1 zeigt exemplarisch die Logausgaben des Knotens v_M und liefert einen ersten Beleg für die prinzipielle Funktionalität unserer Implementierung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN*. Für die Ausgabe wurden nacheinander die einzelnen Knoten aktiviert bzw. wieder deaktiviert, um die prinzipielle Arbeitsweise von *Fast Mode-Signaling* zu prüfen. Entsprechend den vergebenen *Modus-Präferenzen*, setzt sich in jedem *Mode-Based Time Window* jeweils der Knoten mit der höchsten *Modus-Präferenz* durch und überträgt seinen Rahmen. Wie bereits beschrieben versuchen jeweils alle aktivierten modusbasierten Knoten in jedem Time Window, für welches die Knoten eine Slot-Assignment besitzen, einen Rahmen zu übertragen. Die abgedruckten Logausgaben des Listings 3.1 stammen aus einem früheren Testlauf mit einer Übertragungsrate[32] von 100 kBit/s. Bis auf die gewählte

[32] Bei höheren Übertragungsraten beschränkt sich die Ausgabe auf erkannte Abweichungen vom Frame-Assignment (und dessen vorhergesagtem Gewinner des *Fast Mode-Signaling*, vgl. Tabelle 3.4) sowie die Fehlerrate. Die in Listing 3.1 gezeigten detaillierten Ausgaben führen bei höheren Übertragungsraten zu Verzögerungen die Probleme bereiten.

Übertragungsrate ist die Konfiguration im Wesentlichen identisch mit der in Tabelle 3.1 vorgestellten Konfiguration für eine Übertragungsrate von 250 kBit/s. Der erste Zahlenwert einer Zeile repräsentiert jeweils den lokalen Zeitstempel des Time Masters v_M, zu dem dieser den Empfang eines Rahmen detektiert hat. Zusätzlich werden die Basiszyklen durchnummeriert sowie der Empfangszeitpunkt gemäß TTCAN-Level-1 Timestamping ausgegeben (mit einer Auflösung von einer Bitzeit). Die Überschriften innerhalb des Listings geben an, welche Knoten zum Zeitpunkt der nachfolgenden Ausgabe aktiv waren.

3.3.3 Auswertung und Ergebnisse

Um die Robustheit und Funktionalität unseres Protokollstacks zu testen, haben wir mehrere Testläufe für jede Messreihe durchgeführt. Eine Messreihe besteht aus 150 000 *Mode-Based Time Windows*, in denen die Kommunikation gemäß dem aufgestellten Frame-Assignment aus Tabelle 3.4 erfolgt (siehe auch Listing 3.1 ab Zeile 35). Das Frame-Assignment wurde so gewählt, dass jeder Knoten für jedes *Mode-Based Time Window*, für das er über ein Slot-Assignment verfügt, die Übertragung eines Rahmens einplant. So ist gewährleistet, dass in jedem *Mode-Based Time Window* mindestens zwei Rahmen um den Zugriff konkurrieren. Der Knoten v_M registriert alle Übertragungen, vergleicht diese mit den erwarteten Übertragungen gemäß dem Frame-Assignment und berechnet eine Fehlerrate basierend auf den beobachteten Abweichungen. Bei einer Übertragungsrate von 250 kBit/s, mit der vorgestellten Konfiguration, erreichten wir reproduzierbar stets eine Fehlerrate[33] von 0%. Wir haben das gleiche Experiment auch mit der zweiten CAN-Konfiguration mit einer Übertragungsrate von 500 kBit/s wiederholt. Hier kommt unser Prototyp jedoch an seine Grenzen. Im Durchschnitt sind 62,4% aller Übertragungen fehlerhaft, d.h. diese entsprechen nicht den Vorgaben des Frame-Assignment. Zudem unterliegt die Fehlerrate starken Schwankungen, wie mehrfache Wiederholungen der Messreihe zeigten. Hierbei wurden Fehlerraten zwischen 55% bis 80% beobachtet. Als fehlerhaft zählen wir sowohl den Umstand, dass sich der falsche Mode durchsetzt als auch, dass mehr als ein Rahmen pro *Mode-Based Time Window* übertragen wird (d.h. pro Basiszyklus können bis zu vier Fehler auftreten (ein Fehler pro *Mode-Based Time Window*)). Dies kann passieren, wenn beim Erreichen des TSP das Medium bereits durch eine laufende Übertragung belegt ist. In diesem Fall wartet der CAN-Controller bis zum Ende der laufenden Übertragung und sendet dann sofort den Rahmen, sodass auf diese Weise fälschlicherweise mehr als ein Rahmen pro *Mode-Based Time Window* übertragen wird. Hier zeigt sich wieder wie wichtig die Synchronisation und der gleichzeitige Übertragungsstart beim Erreichen des TSP ist. Dieses Verhalten ist letztendlich auf den CAN-Controller des STM32F407 zurückzuführen, diesem fehlen geeignete Betriebsmodi und Einstellungen, um eine Übertragung in diesem Kontext zu verhindern.

Um zu überprüfen, ob die hohen Fehlerraten bei einer Übertragungsrate von 500 kBit/s Resultat eines zu großen Tickoffsets sind, haben wir den maximalen Baseoffset zwischen den modusbasierten Knoten unmittelbar nach dem Empfang der Referencemessage ($d_{baseOffset}$) sowie deren Offset d_{Offset} unmittelbar vor der Übertragung des Rahmens in dem letzten *Mode-Based Time Window* (also die letzte relevante Abweichung vor einer erneuten (Re-)Synchronisation) gemessen. Die so ermittelten Offsets wurden dann anschließend mit den gemäß Constraints 3.6 ermittelten zulässigen oberen Schranken für den maximalen Tickoffset verglichen (2,6672 µs bzw. 1,33 µs bei 250 kBit/s bzw. 500 kBit/s – Tabelle 3.1).

Um diese Offsets messen zu können, werden Flanken an (von außen zugänglichen) GPIO-Pins erzeugt, sobald eine Referencemessage empfangen wird oder ein Knoten versucht seine Übertragung im letzten *Mode-Based Time Window* des Basiszyklus startet[34]. Etwaige Verzögerungen durch die Logik und interne Verarbeitungsverzögerungen des CAN-Controllers beim Start einer Übertragung werden somit hier nicht erfasst. Die unterschiedlichen Offsets wurden in separaten Messungen ermittelt, mit jeweils 7 000 Basiszyklen für beide verwendeten Übertragungsraten. Hierzu wurden die Zeitpunkte, zu denen die jeweiligen Pins der einzelnen Knoten ihren Zustand wechselten, extern aufgezeichnet. Der Time

[33] Die Wiederholungen mitgezählt, kamen wir auf über 1 000 000 Übertragungen in denen kein Fehler auftrat.
[34] Dies erfolgt durch das Setzen eines Kontrollbits der entsprechenden Mailbox, welche die Nachricht enthält.

Master zeigt zusätzlich den Start jedes neuen Basic Cycles über einen separaten Pin an, um die spätere automatische Auswertung der Messergebnisse zu vereinfachen.

Um die Messreihen automatisiert auswerten zu können, haben wir anstelle eines handelsüblichen Logic-Analysers, eine von uns entwickelte Lösung verwendet[35]. Diese basiert auf einem *Digilink Spartan 3E Starter Board* [Xil06] mit einem *Xilinx Spartan 3E FPGA* [Xil13]. Auf dieser Hardware realisierten wir einen Logic-Analyer mit einer Abtastrate von 50 MHz, welcher die Messdaten direkt im CSV-Format[36] über UART ausgibt. Die Abbildung 3.14 zeigt den Versuchsaufbau für die Durchführung der beschriebenen Messreihen.

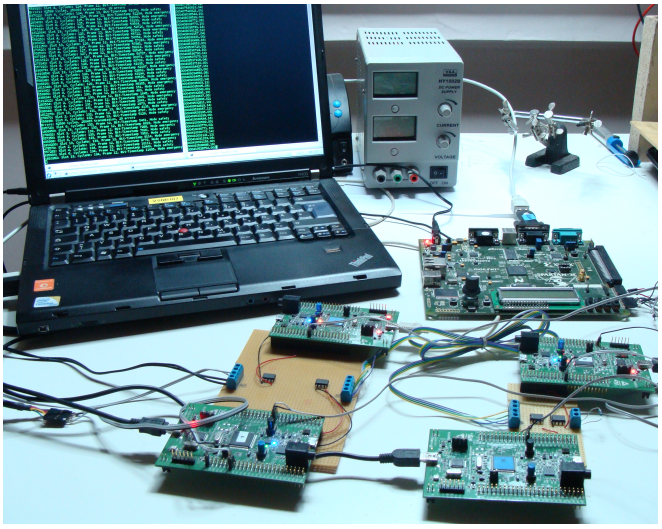


Abbildung 3.14: Messaufbau für die Messung der Offsets.

Abbildung 3.15 stellt die Ergebnisse der Messungen der Offsets für beide Übertragungsraten als Balkendiagramm dar. Die Höhe der Balken gibt den durchschnittlichen Baseoffset $d_{baseOffset}$ unmittelbar nach der Synchronisation bzw. den durchschnittlichen Offset d_{Offset} unmittelbar vor der letzten Übertragung über die gesamte Messreihe hinweg an. Die gemessenen minimalen und maximalen Offsets der Messreihe sind mittels Fehlerbalken dargestellt. Die rot respektive blau eingezeichnete gestrichelte Linie innerhalb der Grafik repräsentiert die obere Schranke für den maximal zulässigen Tickoffset $d^{MAX'}_{maxOffset,MB}$ für 250 kBit/s bzw. 500 kBit/s gemäß Constraint 3.6.

Der mittlere Offset d_{Offset} bei 250 kBit/s ist mit 461,13 ns nur geringfügig größer als der Offset bei 500 kBit/s mit 406,2 ns, obwohl der Basiszyklus bei der Übertragungsrate von 500 kBit/s deutlich kürzer ausfällt, während der gemessene minimale und maximale Offset bei 500 kBit/s geringfügig stärker ausgeprägt ist. Ein ähnliches Bild ergibt sich in Bezug auf den Baseoffset, auch hier ist der mittlere Baseoffset bei 250 kBit/s mit 261,02 ns stärker ausgeprägt als bei einer Übertragungsrate von 500 kBit/s (mit 217,7 ns). Ein möglicher Grund hierfür ist die abweichende Länge der verwendeten Zeitquanten beider Konfigurationen. Da die Zeitquanten bei 500 kBits kleiner sind, wird eine fallende Flanke (durch die diskrete Abtastung des Signalpegels jeweils zu Beginn eines Zeitquantums) im Mittel schneller erkannt als bei der Konfiguration für 250 kBit/s. Auch die maximale Phasenverschiebung zwischen Sender und Empfänger von einem Zeitquantum ist bei 500 kBit/s absolut gesehen kürzer (vgl. Definition 3.8). Die Differenz des maximal beobachteten Baseoffset beider Übertragungsraten beträgt $400\,\text{ns} - 320\,\text{ns} = 80\,\text{ns}$,

[35] Proprietäre Lösungen erlaubten aufgrund ihres undokumentierten Ausgabeformats nur sehr eingeschränkt eine automatische Auswertung oder hatten für die Messungen einen zu kleinen Speicher.

[36] Bei den ausgegebenen Messdaten handelt es sich um einen Zeitstempel sowie den Zustand aller Eingabepins, sobald sich mindestens ein Pegel verändert.

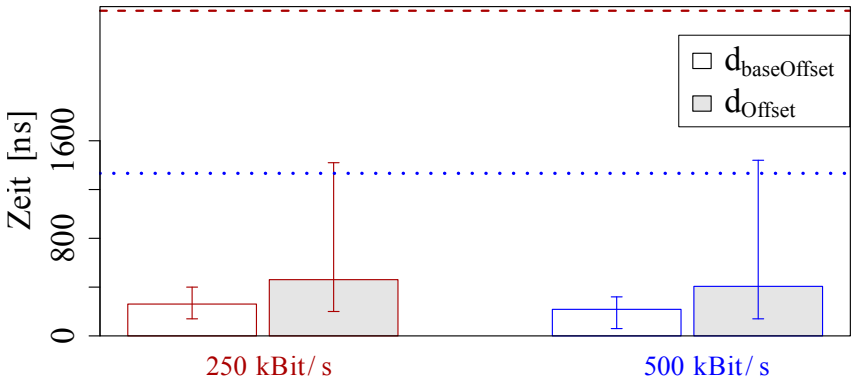


Abbildung 3.15: Gemessene Offsets bei 250 kBit/s und 500 kBits/s.

dies entspricht der Größenordnung der Differenz der Zeitquanten für die beiden Übertragungsraten ($166,6\,\text{ns} - 95\,\text{ns} = 71,6\,\text{ns}$).

Unsere Messergebnisse stützen die bestimmten Fehlerraten insoweit, dass bei einer Datenrate von 250 kBit/s die obere Schranke für den maximal zulässigen Tickoffset gemäß Constraint 3.6 (in Abbildung 3.15 repräsentiert durch die rot gestrichelte Linie) nie überschritten wird. Dementsprechend ist die Fehlerrate von 0%, in Anbetracht des verbleibenden Sicherheitsabstands plausibel. Wird der CAN-Bus mit einer Datenrate von 500 kBit/s betrieben, konnten wir mit unseren Messungen nachweisen, dass der maximal beobachtete Offset die berechnete obere Schranke für den maximal zulässigen Tickoffset (dargestellt durch die blau gestrichelte Linie) überschreitet. Insgesamt liegen bei der in Abbildung 3.15 dargestellten Messreihe in 11,9% der Basiszyklen der gemessene Offsets über dieser Schranke. Die bei dieser Messreihe ermittelte Fehlerrate[37] betrug 55,7%. Die Diskrepanz zwischen diesen beiden Größen deutet darauf hin, dass die Constraints für unsere konkrete Implementierung und unseren CAN-Controller nicht ausreichend detailliert sind. Dies ist insofern nicht ganz überraschend, da Constraint 3.6 bei der Berechnung der oberen Schranke für den maximal zulässigen Tickoffset keine Implementierungseigenschaften, wie Laufzeiten z. B. in Bezug auf die Verarbeitung von Interrupts, berücksichtigt. Ebenso konnten spezifische Verzögerungen die den CAN-Controller sowie dessen genauen internen Abläufe[38] betreffen (Verzögerung für die Verarbeitung und das Speichern von Rahmen sowie die exakten Auslösezeitpunkte der verschiedenen Interrupts für Empfang und Senden von Rahmen oder auftretende Verzögerungen zwischen dem Setzen des Kontrollbits einer Mailbox und dem eigentlichen Übertragungsstart sind nur einige Beispiele hierfür) nicht berücksichtigt werden, da das Datenblatt diesbezüglich keine Aussagen macht. Diese Aspekte bereiteten auch schon bei der Synchronisation Schwierigkeiten. Wie unsere Messungen illustrieren scheinen diese Effekte bei einer Übertragungsrate von 500 kBit/s stark genug, um die hohen Fehlerraten zu verursachen, obwohl die Konfiguration den theoretischen Anforderungen aus Kapitel 3.2.2 genügt.

[37] Die Offsets werden nur einmal pro Basiszyklus ermittelt, während die Fehlerrate sich auf die Anzahl der *Mode-Based Time Windows* bezieht, in denen (mindestens) ein Fehler aufgetreten ist. Unsere Konfiguration verwendet 4 *Mode-Based Time Windows* pro Basiszyklus, d.h. pro Zyklus können maximal 4 Fehler auftreten.

[38] Des Weiteren sind im Datenblatt keine Informationen enthalten die sich darauf beziehen, ob dieser konkrete CAN-Controller nach einer harten Synchronisation den Start einer eigenen Übertragung tatsächlich bis zum Samplepoint gestattet oder nicht (vgl. Kapitel 3.1.1 und die dortigen Erläuterungen zur harten Synchronisation). Auch wenn der CAN-Controller dies unterstützt ist nicht aus dem Datenblatt ersichtlich, bis zu welchem Zeitpunkt das Kontrollbit für den Start der Übertragung (durch die Software) hierfür gesetzt werden muss, da Verarbeitungsverzögerungen seitens des CAN-Controllers ebenfalls nicht dokumentiert sind.

3.3.4 Zusammenfassung

Wie unser Prototyp auf Basis des STM32F4-Mikrocontrollers zeigt, ist eine Implementierung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* mittels handelsüblichen CAN-Controllern und einem in Software realisierten Protokollstack generell möglich. Allerdings ist die maximal erreichbare Übertragungsrate, für die eine zuverlässige Funktion sichergestellt werden kann, bei dieser Form der Realisierung beschränkt. Dies liegt vor allem daran, dass die Eigenschaften der Hardware (insbesondere des CAN-Controllers) sowie die auftretenden Verzögerungen nicht vollständig bekannt und kontrollierbar sind. Der Versuch die Constraints entsprechend anzupassen, um diese Eigenschaften zu berücksichtigen, erscheint aufgrund der nicht dokumentierten Eigenschaften und variablen Verzögerungen jedoch wenig sinnvoll und würde zudem den Overhead der *Mode-Based Time Windows* weiter erhöhen. Stattdessen konzentrieren wir uns lieber auf die Entwicklung eines eigenen modusbasierten TTCAN-Controllers mittels eines FPGA, bei denen die Einflussfaktoren kontrollierbar und die Verzögerungen deterministisch sind (vgl. Kapitel 3.4).

Dies gilt insbesondere, da die Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* mittels eines Softwarestacks weitere Nachteile hat. So führt sie zu einer engen Kopplung zwischen der Anwendung und den zeitkritischen Protokollfunktionen und birgt die Gefahr, dass die Anwendung durch Fehler, wie z. B. Speicherlecks oder lange kritische Bereiche, die Funktionsweise des Protokollstacks und damit die gesamte Kommunikation beeinträchtigen kann. Erschwerend kommt hinzu, dass das deterministische Verhalten sowie Garantien von *Mode-Based Scheduling*, allein auf einer robusten Umsetzung von *Fast Mode-Signaling* beruhen. Die Realisierung von *Fast Mode-Signaling* auf Basis der CAN-Arbitrierung ist zwar sehr effizient, erfordert jedoch eine sehr hohe (und indirekt von der Datenrate abhängige) Synchronisationsgenauigkeit. Die erforderliche Genauigkeit ist gerade bei höheren Datenraten und beschränkter Kontrolle über alle Verzögerungen (Verarbeitung durch den CAN-Controller, Verzögerungen beim Auslösen von Interrupts etc.) schwierig zu gewährleisten bzw. verringert die Effizienz (z. B. durch kurze (Re-)Synchronisationsintervalle) wie unsere Evaluation gezeigt hat. Diese beiden Aspekte machen den Einsatz von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* basierend auf einem einzig in Software realisierten Protokollstack, insbesondere im Kontext sicherheitskritischer Systeme, bedenklich.

3.4 Funktionale Erweiterung von CAN für Mode-Based Scheduling with Fast Mode-Signaling

Wie Kapitel 3.3 illustriert, ist eine Implementierung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* mittels Standard CAN-Controllern zwar möglich und als Proof-of-Concept zum Testen der entwickelten Konzepte geeignet, jedoch für einen praktischen Einsatz in sicherheitskritischen Umgebungen nur eingeschränkt nutzbar. Gerade für einen Einsatz in sicherheitskritischen Anwendungsdomänen wird eine robuste Lösung benötigt, welche in der Lage ist, harte Echtzeitgarantien hinsichtlich der Protokollfunktionalitäten zu gewähren und eine Isolation zwischen Anwendung und Protokollstack bei Fehlern sicherzustellen. Zusätzlich gilt es, die Verlässlichkeit von *Fast Mode-Signaling* gegenüber Synchronisationsungenauigkeiten zu verbessern.

Die ersten beiden Probleme lassen sich beheben, indem wir die Protokolllogik in Hardware realisieren, d.h. in Form eines externen TTCAN-Controllers, welcher direkt die erforderliche Unterstützung für TTCAN und *Mode-Based Scheduling* implementiert (z. B. durch Bereitstellung der beschriebenen Pufferstrukuren in Hardware – vgl. Kapitel 3.3.1). Durch die direkte Unterstützung entfallen die Einschränkungen, die sich durch die Verwendung von Standard CAN-Controllern ergeben, insbesondere sind exakte Aussagen bzgl. auftretender Verzögerungen und die vollständige Kontrolle des Verhaltens des Controllers bei der Übertragung möglich. Bei herkömmlichen CAN-Controllern erschöpft sich die Kontrolle über die

Übertragung eines Rahmens darin, diesen in einer Mailbox abzulegen und den Sendevorgang anzustoßen. Ab diesem Zeitpunkt ist keine Kontrolle mehr möglich. Ist das Medium frei, beginnt der CAN-Controller mit der Übertragung. Ansonsten wird entweder auf eine Übertragung verzichtet[39] (falls der Controller noch an der Arbitrierung teilnehmen kann, diese aber verliert) oder die Übertragung gestartet, nachdem das Medium wieder frei ist. Integriert man die Unterstützung für *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* direkt in den TTCAN-Controller, können spezielle Hardware-Timer (innerhalb des Controllers) Übertragungen präziser anstoßen, als dies durch einen externen Protokollstack umsetzbar ist, der erst noch mit dem TTCAN-Controller interagieren muss. Durch die Integration des Timers in den CAN-Controller lässt sich sicherstellen, dass der CAN-Controller sich tatsächlich bis zum Erreichen seines Samplepoint (für das SOF-Bit) noch an einer bereits gestarteten Übertragung beteiligen kann. Durch die direkte Unterstützung von TTCAN bzw. *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* innerhalb des CAN-Controllers können auch die Übertragungen mehrerer Rahmen pro Time Window unterbunden werden.

Ein Teil dieser Funktionen wird von dem TTCAN-Standard als Bestandteil vollwertiger TTCAN-Controller bereits gefordert [HMFH02]. Jeder TTCAN-Controller benötigt einen Triggerspeicher, in dem die zu übertragenden Rahmen abgelegt werden können. Der TTCAN-Controller startet die Übertragung der im Triggerspeicher hinterlegten Rahmen zu festgelegten Zeitpunkten (z. B. beim Start eines Time Window) automatisch. Hierdurch wird zwar die geforderte Entkopplung zwischen Anwendung und den zeitkritischen Protokollfunktionalitäten und ein möglichst präzisen Übertragungsstart gewährleistet, aber natürlich keine Unterstützung für *Mode-Based Scheduling* bereitgestellt.

Die geplanten TTCAN-Controller mit Unterstützung für TTCAN-Level-2 bieten eine bessere Hardwareunterstützung für die Synchronisation, eine verbesserte Auflösung der NTU und eine Korrektur des Clock Skew. Hierdurch wird die Synchronisationsgenauigkeit erhöht, sodass längere Basiszyklen möglich sind. Jedoch wird auch durch die verbesserte Synchronisationsgenauigkeit die konzeptionelle Schwäche der Realisierung des *Fast Mode-Signaling* gegenüber *hinreichend zeitgleichen* Übertragungsstarts und den hieraus möglicherweise resultierenden nicht deterministischen Ergebnissen des Wettbewerbs[40] nicht behoben.

Die zentrale Fragestellung, der wir uns in diesem Kapitel widmen, lautet: Welche Konzepte sind notwendig, um eine gegenüber Synchronisationsfehlern und -ungenauigkeiten robuste, zuverlässige und deterministische Realisierung des *Fast Mode-Signaling* auf Basis der CAN-Arbitrierung zu entwickeln. Hierzu betrachten wir zunächst in Kapitel 3.4.1 die bereits identifizierten Herausforderungen und Schwierigkeiten bei der bisherigen Realisierung. Anschließend stellen wir basierend auf dieser Analyse unsere BOT-Erweiterung (**B**ring-your-**o**wn-**T**ick) für CAN vor, welche eine robuste Umsetzung des *Fast Mode-Signaling* ermöglicht (Kapitel 3.4.2). Die Umsetzung von BOT erfordert jedoch eine Modifikation des Verhaltens des (TT)CAN-Controllers auf CAN-Protokollebene. Aufgrund fehlender Schnittstellen und der harten Echtzeitanforderung ist eine Realisierung weder in Software noch mit Standard CAN-Controllern möglich. Um die Funktionalität von BOT nachzuweisen, haben wir daher einen Prototyp für einen solchen (TT)CAN-Controller auf Basis eines FPGA entwickelt. Das Verhalten des Controllers wurde mit VHDL spezifiziert (Very High Speed Integrated Circuit Hardware Description Language, [IEE00]). Kapitel 3.4.3 beschreibt den internen Aufbau des Controllers sowie die Integration von BOT. Die Evaluation des TTCAN-Controllers sowie der BOT-Erweiterung erfolgte dann mit Hilfe von Simulationen (Kapitel 3.4.4). Kapitel 3.4.5 fasst die Ergebnisse zusammen und liefert eine Bewertung der BOT-Erweiterung und deren Bedeutung für die Umsetzung des *Fast Mode-Signaling*.

[39] Wir gehen davon aus, dass die automatische erneute Übertragung bei einer verlorenen Arbitrierung deaktiviert ist. Diese Konfigurationsoption bieten die meisten CAN-Controller.

[40] So kann sich ein Knoten mit einer niedrigeren *Modus-Präferenz* durchsetzen, sofern dieser seine Übertragung vor den anderen Knoten startet, weil die anderen Knoten beim Erreichen ihres TSP ein bereits belegtes Medium vorfinden.

3.4.1 Hindernisse bei der Realisierung des Fast Mode-Signaling

Eine zuverlässige Abbildung des *Fast Mode-Signaling* durch die CAN-Arbitrierung erfordert eine hohe Synchronisationgenauigkeit der modusbasierten Knoten (vgl. Kapitel 3.2.2). Nur so ist gewährleistet, dass alle Knoten (nahezu) gleichzeitig ihren TSP erreichen und ihre Übertragungen starten, so dass sich die SOF-Bits der Rahmen aller konkurrierenden Knoten überlappen und der Knoten mit der höchsten *Modus-Präferenz* die Arbitrierung gewinnt. Die obere Schranke für den maximal zulässigen Offset $d^{MAX}_{maxOffset,MB}$, der dies sicherstellt, wird von Constraint 3.5 definiert. Bei einer größeren Abweichung könnte es passieren, dass sich ein Knoten mit einer schnelleren lokalen Uhr gegenüber einem Knoten mit einer langsameren lokalen Uhr durchsetzt, obwohl letzterer einen Rahmen mit einer höheren *Modus-Präferenz* übertragen möchte. Ursächlich hierfür ist, dass der Knoten mit der langsameren Uhr zu dessen geplanten Übertragungsstart (TSP) ein bereits belegtes Medium vorfindet und daher seine Übertragung nicht startet. Mit Standard (TT)CAN besteht die einzige Möglichkeit diesem Problem zu begegnen darin, den Basiszyklus hinreichend kurz zu wählen, sodass die (Re-)Synchronisationen für einen hinreichend kleinen Tickoffset sorgen (vgl. Definition 3.7 und 3.8). Versucht man zusätzlich auch noch weitere Einflussfaktoren in den Constraints zu berücksichtigen (wie z. B. Verarbeitungszeiten eines CAN-Controllers oder Implementierungsdetails eines Protokollstacks), resultieren die Unsicherheiten und Indeterminismen häufig in sehr ungünstigen oberen Schranken, welche die Länge des Basiszyklus sehr stark beschränken.

Die von uns angestrebte Realisierung der Protokolllogik in Hardware (FPGA) erlaubt die Herleitung exakter oberer Schranken (z. B. können die Verarbeitungszeiten taktgenau bestimmt werden) und gestattet die Einbettung der Konzepte von *Mode-Based Scheduling* direkt in den Controller. Über die Implementierung von TTCAN-Level 2 sowie zusätzliche Techniken, wie *Early-Timestamping* in Verbindung mit einer höheren Abtastrate[41], kann zudem die Synchronisationsgenauigkeit wesentlich erhöht werden.

Trotz dieser Maßnahmen bleibt die Beschränkung des (Re-)Synchronisationsintervalls (und damit der Länge der Basiszyklen) dennoch die einzige Möglichkeit, die Einhaltung der oberen Schranke für den maximal zulässigen Tickoffset $d^{MAX}_{maxOffset,MB}$ – und somit die korrekte Funktionalität des *Fast Mode-Signaling* – zu garantieren. Dies führt wiederum indirekt zu einer Verschlechterung der Bandbreitennutzung und widerspricht der eigentlichen Intention von *Mode-Based Scheduling*. Aus diesem Grund haben wir BOT entwickelt, eine funktionale Erweiterung für TTCAN-Controller, mit deren Hilfe sich *Fast Mode-Signaling* trotz Synchronisationsungenauigkeiten robust und deterministisch umsetzen lässt.

3.4.2 Bring-your-own-Tick-Erweiterung für Fast Mode-Signaling

Die grundlegende Idee unserer *Bring-your-own Tick (BOT)*-Erweiterung besteht darin, es einem modusbasierten Knoten in einem *Mode-Based Time Window* zu ermöglichen, sich nachträglich in eine bereits (aus Sicht des Knotens zu früh) gestartete Arbitrierung einzuklinken. Dies gestattet es dem Knoten doch noch die Arbitrierung zu gewinnen, sofern sein Mode die höchste *Modus-Präferenz* aller konkurrierenden Modes aufweist. Hierzu definieren wir zusätzlich zu dem TSP das sogenannte Transmission Start Window (TSW). Bei dem TSW handelt es sich um eine Zeitspanne konfigurierbarer Länge. Das TSW eines *Mode-Based Time Window* endet mit dessen TSP. Jeder Knoten, der eine Übertragung in dem *Mode-Based Time Window* plant, überwacht das zu dem Time Window gehörende TSW auf Übertragungsstarts. Erkennt ein solcher Knoten einen (aus seiner Sicht) verfrühten Übertragungsstart innerhalb des TSW – d.h., eine Übertragung, welche beginnt, bevor der Knoten seinen eigenen TSP (gemäß seiner lokalen Uhr) erreicht –, so gestattet BOT es ihm, sich in die bereits laufende Übertragung (bzw. genauer die Arbitrierung) einzuklinken.

[41] Eine höhere Abtastrate reduziert den maximalen Baseoffset, da die maximale Verzögerung für die Erkennung einer Flanke reduziert wird. Aufgrund der Implementierung der Flankenerkennung geht die Länge des Abtastintervalls d_{CLK} in die Verarbeitungszeit auf Seiten des Empfängers in Definition 3.3 ein (d.h. $d^{b,max}_{rxProcDelay,e} = d_{CLK} + x$, wobei x für weitere Verarbeitungsverzögerungen steht).

Durch das (rechtzeitige) Einklinken in eine laufende Übertragung stellt die BOT-Erweiterung die korrekte Funktion des *Fast Mode-Signaling* auch dann sicher, wenn der Tickoffset zwischen den modusbasierten Knoten größer als $d^{MAX}_{maxOffset,MB}$ (Constraint 3.5) ist. Die einzige Voraussetzung ist, dass der tatsächliche Tickoffset zwischen den modusbasierten Knoten kleiner als das gewählte TSW ist, sodass ein (verfrühter) Übertragungsstart trotzdem in das TSW aller Knoten fällt, welche eine Übertragung planen. Somit kann die Robustheit des *Fast Mode-Signaling* gegenüber Synchronisationsungenauigkeiten frei über die Länge des TSW skaliert werden, ohne die Länge des Basiszyklus beschränken zu müssen. Natürlich muss der maximale Tickoffset für den gewählten Basiszyklus weiterhin bei der Dimensionierung des *Mode-Based Time Window*, des TSP (Constraint 3.2 und 3.4) und nun auch bei der Wahl des TSW berücksichtigt werden.

Die Umsetzung der BOT-Erweiterung muss direkt innerhalb des CAN-Controllers erfolgen, um die harten Echtzeitanforderungen erfüllen zu können, welche das Einklinken in die laufende Übertragung erfordert. Formal stellt dieser Vorgang eine Verletzung des CAN-Standards dar, die BOT-Erweiterung ist jedoch insofern verträglich, dass CAN-Controller ohne BOT die mittels dieser Erweiterung gesendeten Rahmen korrekt empfangen und interpretieren können.

Um das Einklinken in die laufende Übertragung zu ermöglichen – so dass sich die Bits der Identifier korrekt überlagern – wird eine sehr genaue Synchronisation zwischen den CAN-Controllern benötigt. Als Referenzzeitpunkt dient hierbei jedoch nicht der Empfang der Referencemessage, sondern stattdessen die fallende Flanke des SOF-Bits der Übertragung, in die sich diese einklinken wollen. Die fallende Flanke des SOF-Bits löst bei einem Knoten eine harte Synchronisation innerhalb des CAN-Zustandsautomaten für das Bit-Timing aus. Daraufhin wird das Bit-Timing mit dem Ende des Synchronisationssegments neu gestartet. Dies führt dazu, dass es sich bei jedem Knoten so darstellt, als ob die detektierte fallende Flanke des SOF-Bits innerhalb des Synchronisationssegments liegt. Sämtliche für die folgenden Bitübertragungen relevanten Zeitpunkte (SamplePoint, Ende des Bits) beziehen sich auf den Zeitpunkt des Empfangs der fallenden Flanke des SOF-Bits und sind relativ zu dem so festgelegten Synchronisationssegment ausgerichtet. Dementsprechend stellt die harte Synchronisation auf die fallende Flanke des SOF-Bits selbst eine Ticksynchronisation der CAN-Controller dar, die nur Auswirkungen auf die aktuelle Übertragung bzw. den Empfang des aktuellen Rahmens hat[42]. Durch den Umstand, dass die harte Synchronisation jeweils zu Beginn des Rahmens erfolgt, wird der Einfluss des Clock Skews auf die Übertragung minimiert[43].

Der *harte Baseoffset*, der aus einer harten Synchronisation resultiert, wird durch die maximale Signalverzögerung sowie die Länge der Zeitquanten beschränkt. Hierdurch wird auch die Abweichung des Bit-Timings der einzelnen Knoten zueinander (und somit der Samplepoints, Synchronisationssegmente etc.) auf ein für CAN zulässiges Maß begrenzt.

Definition 3.12 (Maximaler harter Baseoffset durch die harte Synchronisation)

Sei V die Menge der Knoten eines TTCAN-Busses. Des Weiteren sei $v_T \in V$ ein Knoten, dessen fallende Flanke des SOF-Bits von einem Knoten $v_E \in V$ als Erstes empfangen wird. Dann beträgt der maximale harte Baseoffset zwischen dem CAN-Controller von v_T und dem CAN-Controller von v_E

$$d_{hardOffset,v_E} \leq delay_{max}(v_T, v_E) + d_q^{v_E}$$

wobei $d_q^{v_E}$ die konfigurierte Länge der Zeitquantums des Knotens[44] v_E bezeichnet.

[42] Der Name BOT – *Bring-your-own Tick* – leitet sich von dem Umstand ab, dass jede Übertragung ihren eigenen Referenzzeitpunkt (Tick) mitbringt.

[43] Ganz im Gegensatz zu der (Re-)Synchronisation von TTCAN, welche sich auf die Referencemessage (und somit den Beginn des Basiszyklus) bezieht und z. B. die Position des TSP vorgibt.

Wie aus der Definition ersichtlich, hängt der maximale harte Baseoffset einer Übertragung bei der harten Synchronisation nur von technischen Kenngrößen des CAN-Busses ab, aber nicht von der Abweichung (dem Tickoffset) der lokalen Uhren der (modusbasierten) TTCAN-Knoten, deren Referenzzeitpunkt sich auf den Empfang der Referencemessage bezieht.

Die grundlegende Idee von BOT, sich an einer bereits laufenden Übertragung zu beteiligen (sich also einzuklinken), wird im CAN-Protokoll bereits genutzt. Bei CAN erfolgt das Einklinken im Gegensatz zu BOT allerdings an einer anderen Stelle innerhalb des Rahmens und in einem anderen Kontext. CAN selbst verwendet das Einklinken bei jeder Übertragung eines Rahmens für die Realisierung des Acknowledge-Mechanismus. Ein weiteres Beispiel ist die Fehlersignalisierung, dort wird eine laufende Übertragung durch einen Fehlerrahmen überlagert, sofern ein Empfänger mit der Fehlersignalisierung beginnt. Auch hier erfolgt ein Einklinken in die laufende Übertragung. Aus technischer Sicht handelt es sich bei dem Einklinken in eine laufende Übertragung (auf dem BOT beruht) also um einen sehr robusten und häufig verwenden Ansatz im CAN-Protokoll.

Um die Abläufe beim Einklinken besser zu verstehen, betrachten wir den von CAN verwendeten Acknowledge-Mechanismus genauer. Dieser beruht auf der Idee, dass alle Knoten, welche einen fehlerfreien Rahmen empfangen haben, diesen aktiv quittieren. Um den erfolgreichen Empfang zu signalisieren überlagern die Empfänger gezielt ein rezessives Bit (rezessiver Signalpegel) innerhalb des zu quittierenden Rahmens durch von ihnen gesendetes dominantes Bit (dominanter Signalpegel). Hierfür existiert innerhalb des CAN-Rahmenformats ein spezielles Bestätigungsfeld (Acknowledge-Feld), bestehend aus einem Acknowledge-Slot (ACK-Slot), für die Signalisierung des korrekten Empfangs, sowie einem rezessiven Delimiter-Bit (vgl. Abbildung 3.1). Der Sender[45] eines Rahmen überträgt innerhalb des ACK-Slots stets ein rezessives Bit. Jeder Empfänger, der bis zum Erreichen des Bestätigungsfeldes keinen Fehler detektiert hat, sendet ein dominantes Bit (dominanter Signalpegel) innerhalb des ACK-Slots. Die dominanten Pegel der verschiedenen (erfolgreichen) Empfänger und das rezessive Bit des Sender überlagern sich. Empfängt der Sender innerhalb des ACK-Slots ein dominantes Bit, so hat mindestens ein Knoten den Rahmen korrekt empfangen. Erkennt ein Empfänger hingegen einen CRC-Fehler, so sendet er ein rezessives Bit innerhalb des ACK-Slot und beginnt mit der Fehlersignalisierung (durch das Senden eines Fehlerrahmens) nach dem Abschluss des Bestätigungsfeldes.

Das von den Empfängern für die Übertragung innerhalb des Acknowledge-Slots verwendete Bit-Timing basiert auf dem Empfangszeitpunkt der fallenden Flanke des SOF-Bits. Dieses löst beim Empfänger eine harte Synchronisation aus und startet das Bit-Timing mit dem Ende des Synchronisationssegments. Aufgrund der Signalverzögerung zwischen Sender und Empfänger gelingt hierbei nie eine perfekte Synchronisation, der maximale hieraus resultierende harte Baseoffset zwischen zwei Knoten ist jedoch beschränkt (vgl. Definition 3.12) und erlaubt ein Einklinken.

Abbildung 3.16 illustriert den Ablauf und das Bit-Timing anhand zweier Knoten. Dargestellt ist der Ausgangspegel von Sender s_1 sowie die Aus- und Eingangspegel des Empfängers v_1. Unterhalb der Signalverläufe ist die knotenlokale Einteilung der Zeit in Zeitquanten sowie die Zuordnung der Zeitquanten zu den verschiedenen Segmenten des Bit-Timings angegeben. Die Farben geben die Zuteilung der Zeitquanten zu den Segmenten des Bit-Timings an.

Zum Zeitpunkt t_0 beginnt s_1 mit der Übertragung seines SOF-Bits. Dessen fallende Flanke wird von v_1 zum Zeitpunkt t_1 empfangen. Detektiert wird die fallende Flanke erst beim Beginn des nachfolgenden Zeitquantums aufgrund der diskreten Abtastzeitpunkte. Daraufhin startet v_1 sein Bit-Timing neu, sodass das Zeitquantum, in dem die fallende Flanke liegt, zum Synchronisationssegment wird (vgl. Kapitel 3.1.1, Definition 3.1). Zum Zeitpunkt t_2 beginnt s_1 mit der Übertragung des ersten Bits seines Identifiers und den weiteren Bits seines Rahmens. Die Übertragung des CRC-Delimiter ist zum Zeitpunkt t_3 abgeschlossen.

[44] Der Summand d_q^{vE} berücksichtigt den Umstand, dass der Empfänger das Medium nur zu Beginn eines Zeitquantums abtastet und die Zeitquanten von Sender und Empfänger nicht aufeinander ausgerichtet sind.

[45] Da der Bestätigungsmechanismus auch in Datenanforderungsrahmen eingesetzt wird, kann ein Rahmen auch von mehreren Sender zeitgleich gesendet werden, auch in diesem Fall funktioniert das Einklinken (vgl. Kapitel 3.1.1).

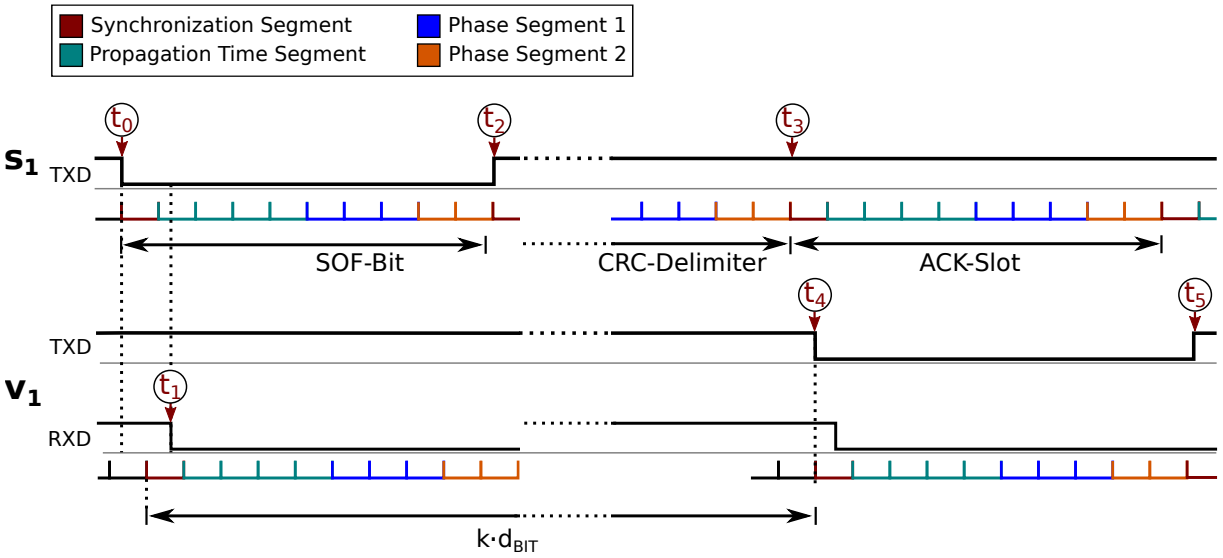


Abbildung 3.16: Ablauf des Acknowledge-Mechanismus mit einem Sender und einem Empfänger.

Für den Knoten v_1 beginnt der ACK-Slot, gemäß seines lokalen Bit-Timings zum Zeitpunkt t_4 mit dem Beginn des Synchronisationssegments. v_1 wird daraufhin zum Sender und überträgt sein dominantes ACK-Bit[46]. Wie in der Grafik dargestellt bezieht sich das Bit-Timing von v_1 (und somit auch der Übertragungsstart des ACK-Bits) auf das Synchronisationssegment des SOF-Bits. Dieses wird durch den Empfangszeitpunkt der fallenden Flanke des SOF-Bits von s_1 (durch die harte Synchronisation) festgelegt. In diesem Beispiel beginnt der ACK-Slot $k \cdot d_{Bit}$ nach dem Synchronisationssegment[47]. Zum Zeitpunkt t_5 ist die Übertragung des ACK-Bits abgeschlossen und v_1 gibt seine Rolle als Sender wieder auf.

Wie anhand des Acknowledge-Mechanismus ersichtlich nutzt BOT (bezogen auf das Einklinken) einen bereits in CAN existierenden Mechanismus. Der wesentliche Unterschied besteht darin, dass BOT das Einklinken an einer anderen Position innerhalb des Rahmens gestattet[48], nämlich im Arbitrierungsfeld. Zusätzlich kombiniert BOT das Einklinken mit einem TSW sowie einer Transmission Start Preamble (siehe unten), um hierfür eine robuste Realisierung des *Fast Mode-Signaling* zu ermöglichen. Durch das Einklinken im Arbitrierungsfeld ermöglicht es BOT einem Knoten, sich noch nachträglich an einer bereits laufenden Arbitrierung zu beteiligen und diese, abhängig von der Priorität seines Rahmens, zu gewinnen.

In dem in Abbildung 3.16 dargestellten Ablauf haben wir angekommen, dass es während der Übertragung zu keinen Bit-Resynchronisationen kommt. Bit-Resynchronisationen können z. B. aufgrund eines zu hohen Clock Skew zwischen den Knoten auftreten. Häufiger sind diese jedoch in der Praxis anzutreffen, wenn Knoten ihre Rolle wechseln und vom Sender zum Empfänger oder umgekehrt werden [HB99]. Bit-Resynchonisationen sind innerhalb des Arbitrierungsfeldes sehr häufig, wenn mehrere Knoten um den Medienzugriff konkurrieren. Dann führen die (nahezu) gleichzeitigen Übertragungsstarts dieser Knoten in Verbindung mit den unterschiedlichen Signalverzögerungen oft dazu, dass sich Empfänger auf die fallende Flanke (des SOF-Bit) unterschiedlicher Sender[49] (hart) synchronisieren. Als Folge hiervon richtet jeder Empfänger sein Bit-Timing auf den aus seiner lokalen Sicht frühesten Sender aus. Verliert dieser die Arbitrierung, so müssen sich die auf diesen Knoten synchronisierten Empfänger, auf einen anderen

[46] Aufgrund des *loop-delays* liegt dieses erst mit einer kurzen Verzögerung am Eingang von v_1 an.

[47] In dem dargestellten Beispiel erfolgt während der Übertragung des Rahmens keine Bit-Resynchronisationen (vgl. Definition 3.2).

[48] Die Position innerhalb des Rahmens spielt jedoch keine Rolle. Das Einklinken ist nicht auf das Bestätigungsfeld beschränkt, sondern kann z. B. wie bei der Fehlersignalisierung zu (fast) jedem Zeitpunkt innerhalb des Rahmens erfolgen.

[49] Jeweils abhängig davon, zu welchem Sender die bei ihnen am frühesten eingetroffene Flanke gehört.

Sender synchronisieren. Bei der nächsten fallenden Flanke synchronisieren sich die Empfänger wieder auf denjenigen Sender, dessen fallende Flanke den Empfänger nun zuerst erreicht. Innerhalb der Übertragung wird hierfür die Bit-Resynchronisation verwendet, welche den auftretenden Phasenfehler[50] durch eine Verkürzung bzw. Verlängerung der Phasensegmente zu kompensieren versucht (vgl. Definition 3.2). Auch innerhalb des Bestätigungsfelds ist dieser Effekt zu beobachten, hier fällt der ursprüngliche Sender weg, stattdessen werden die vorherigen Empfänger zu Sendern und übermitteln eine fallende Flanke. Auch diese löst eine Bit-Resynchronisation aus und sorgt dafür, dass die Knoten ihr Bit-Timing erneut auf den aus ihrer Sicht frühesten Sender ausrichten und damit die Abweichungen bezüglich des Bit-Timings reduzieren.

Auch der Bit-Resynchronisations-Mechanismus stellt sicher, dass BOT und insbesondere das Einklinken korrekt funktioniert indem dieser die Abweichung zwischen den Bit-Timings der Knoten reduziert. Lediglich die Signalverzögerung zwischen den Knoten lässt sich auf diesem Weg nicht kompensieren. Die Wahl des Propagation Time Segments (Constraint 3.1) stellt aber sicher, dass sich auch bei maximalen Signalverzögerungen die einzelnen Bits des Identifiers auf dem Medium korrekt überlagern.

Die Signalverzögerung kann zu einem maximalen harten Baseoffset zwischen einem Sender s_1 und einem Knoten v_1 (der sich hart auf s_1 synchronisiert) von $delay_{max}(s_1, v_1) + d_q^{v_1}$ führen. Klinkt sich v_1 in die Übertragung von s_1 ein, so sendet dieser sein Bit zu Beginn seines Synchronisationssegments (gemäß seiner lokalen Uhr). Weil das Propagation Time Segment jedes Knotens jedoch mindestens doppelt so groß wie die maximale Signalverzögerung ist (Constraint 3.1: $\forall v \in V.d_{PropSeg}^v \geq 2 \cdot delay_{max}^V$) und das Synchronisations- und Phasensegment zusammen (mindestens) zwei Zeitquanten umfassen[51], trifft ein von v_1 gesendeter Pegel – auch bei einem maximalen harten Baseoffset von $delay_{max}(s_1, v_1) + d_q^{v_1}$ – bei allen anderen Knoten rechtzeitig[52] ein. Rechtzeitig bedeutet, bevor die Knoten ihren Samplepoint für dieses Bit erreichen und den Buspegel interpretieren.

Evtl. auftretende Bit-Resynchronisationen führen zwar zu einer Verlängerung oder Verkürzung der Phasensegmente, sind aber bezüglich der Überlagerung unproblematisch, da diese jeweils nur durch eine fallende Flanke ausgelöst werden. In diesem Fall liegt jedoch bereits der finale dominante Pegel am Bus an, unabhängig von dem von v_1 gesendeten Pegel.

Da das Einklinken aus Sicht des Bit-Timings kein Hindernis darstellt, kommen wir nun zu den weiteren Konzepten der BOT-Erweiterung und beschreiben die Abläufe noch einmal im Detail. Zunächst definieren wir formal das Transmission Start Window.

[50] Hierbei handelt es sich um die gemessene Abweichung zwischen dem Empfang einer fallenden Flanke und der Lage des Synchronisationssegments (gemäß dem lokalen Bit-Timing des Knotens) in dem die Flanke eigentlich hätte empfangen werden sollen. Ein Phasenfehler von Null entspricht dem Empfang der fallenden Flanke innerhalb des Synchronisationssegementes und Bedarf keiner Korrektur (vgl. Definition 3.2).

[50] Da der Bestätigungsmechanismus auch in Datenanforderungsrahmen eingesetzt wird, kann es sich auch um mehrere Sender handeln (vgl. Kapitel 3.1.1).

[51] Wir gehen hier implizit davon aus, dass die Zeitquanten der Knoten eines Netzes ähnlich dimensioniert werden.

[52] Diese Situation (zwei aktive Sender in der Arbitrierungsphase mit maximalem harten Baseoffset zueinander) kann auch bei der regulären Arbitrierung auftreten. Dies ist der Fall, wenn sich ein Sender hart auf den anderen synchronisiert, sich aber dennoch an der aktuellen Arbitrierung beteiligt. Hierfür muss dieser Knoten nur vor dem Erreichen seines Samplepoints mit dem Senden seines eigenen SOF-Bits beginnen. Dies ist zulässig, da trotz harter Synchronisation die Erkennung eines belegten Mediums erst bei der Interpretation des anliegenden Buspegels zum Samplepoint erfolgt (vgl. Abbildung 3.9 sowie entsprechende Erläuterungen in Kapitel 3.2.2.3 und 3.1.1). Die Vorgaben für das Bit-Timing (bzgl. der Konfiguration des Propagation Time Segments und der weiteren Segmente) stellen auch in diesem Fall die korrekte Funktion der CAN-Arbitrierung sicher. Dies gilt auch bei mehr als zwei aktiven Sendern.

Definition 3.13 (Transmission Start Window (TSW))

Ein Transmission Start Window (TSW) bezeichnet ein Intervall mit einer festen Dauer d_{TSW} und endet mit dem konfigurierten TSP eines Mode-Based Time Window. Ist s ein Mode-Based Time Window, welches zum Zeitpunkt t_s startet, dann entspricht das TSW dem Intervall $[t_s + d_{TSP} - d_{TSW}, t_s + d_{TSP}]$. Die Dauer der Transmission Start Windows d_{TSW} ist ein Konfigurationsparameter, der für alle Knoten und Mode-Based Time Windows identisch gewählt wird. d_{TSW} ist so zu wählen, dass sich die TSW angrenzender Mode-Based Time Windows nicht überlappen.

Jeder modusbasierte Knoten mit aktiver BOT-Erweiterung, welcher in einem *Mode-Based Time Window* einen Rahmen senden möchte, überwacht während des TSW dieses *Mode-Based Time Window* das Medium. Sobald innerhalb des TSW der Übertragungsstart eines Rahmens detektiert wird, klinken sich diese Knoten in die laufende Übertragung ein, auch wenn deren eigentlicher TSP (gemäß ihrer lokalen Uhr) noch nicht erreicht wurde[53]. Somit stellt die BOT-Erweiterung innerhalb des TSW sicher, dass das *Fast Mode-Signaling* auch bei einer Synchronisationsungenauigkeit größer als $d^{MAX}_{maxOffset,MB}$ korrekt arbeitet, solange die Synchronisationsungenauigkeit die konfigurierte Länge des TSW nicht überschreitet (d.h. $d_{TSW} > d_{maxOffset,MB}$ gewählt wurde, vgl. Definition 3.7). Die Konfiguration des Parameters d_{TSW} hat somit direkten Einfluss auf die Robustheit des *Fast Mode-Signaling*. Um zu gewährleisten, dass eine Übertragung hierbei nicht die lokalen Grenzen des *Mode-Based Time Window* verletzt, ist es grundsätzlich sinnvoll $d_{TSW} \leq d_{TSP}$ zu wählen[54].

Eine einfache Variante BOT zu implementieren besteht darin, unmittelbar nach der Detektion einer fallenden Flanke innerhalb des TSW sofort mit der Übertragung des eigenen Rahmens (mit dem SOF-Bit) zu beginnen. Dies würde aber dazu führen, dass auch kurze Störimpulse innerhalb des TSW fälschlicherweise einen Übertragungsstart auslösen könnten. Außerdem würde diese Variante von BOT den entsprechenden CAN-Mechanismus zur Unterdrückung (Filterung) dieser Art von kurzzeitigen Störimpulsen aushebeln[55]. Eine aufgrund eines Störimpulses innerhalb des TSW ausgelöste Übertragung würde vor dem Erreichen des TSP erfolgen. Hierbei besteht die Gefahr, die Integrität von benachbarten Time Windows zu gefährden (z. B. wenn ein Störimpuls zu Beginn des TSW eines ohnehin vorauseilenden Knotens aufgetreten ist).

Damit die BOT-Erweiterung gegen diese Klasse von Störungen abgesichert ist, erfolgt ein Einklinken in eine Übertragung nicht direkt bei der Erkennung einer fallenden Flanke eines potentiellen SOF-Bits, sondern erst nach dem erfolgreichen Empfang einer Transmission Start Preamble.

Definition 3.14 (Transmission Start Preamble (TP))

Sei M die Menge der Modes, SA eine Slot-Assignment-Funktion und $\Sigma = \{0,1\}$ ein Alphabet, wobei 0 ein dominantes Bit und 1 ein rezessives Bit codiert. Des Weiteren bezeichne $C_{m,s} \subseteq \Sigma^+$ den dem Mode $m \in M$ für Slot s zugeordneten CAN-Identifier (inkl. des führenden SOF-Bits). Dann gilt für die konfigurierbare Transmission Start Preamble $p_s \in \Sigma^+$ eines Mode-Based Time Window s, dass

$$\forall m \in M, c \in C_{m,s}.(SA(s,m) \text{ definiert } \Rightarrow isPrefix(p_s, c))$$

Das Prädikat $isPrefix : \Sigma^* \times \Sigma^*$ prüft, ob das erste Wort ein Präfix des zweiten ist. Für jede Transmission Start Preamble eines Mode-Based Time Window s muss gelten, dass $|p_s| \geq 1$.

[53] Der Grund hierfür kann sowohl eine schneller laufende Uhr des anderen Knotens als auch eine zu langsam laufende Uhr des lokalen Knotens sein.

[54] Dies entspricht auch der Intention des TSP, die Grenzen der *Mode-Based Time Window* trotz Synchronisationsungenauigkeiten zu wahren.

[55] Wird bei einem unbelegten Medium eine fallende Flanke erkannt, so erfolgt eine harte Synchronisation, da es sich um ein potentielles SOF-Bit handelt. Die Interpretation des Buspegels erfolgt jedoch erst beim Erreichen des Samplepoints. Liegt beim Erreichen des Samplepoint anstelle eines dominanten ein rezessiver Pegel am Bus an, so wurde die harte Synchronisation mit hoher Wahrscheinlichkeit durch eine Störung verursacht. Daher interpretiert der Knoten das Medium (weiterhin) als unbelegt und startet deshalb auch keine Fehlersignalisierung.

Die minimale TP eines *Mode-Based Time Window* gemäß Definiton 3.14 besteht nur aus dem SOF-Bit[56]. Bei der Verwendung der BOT-Erweiterung wird für jedes *Mode-Based Time Window* s eine TP p_s konfiguriert, die pro Slot individuell festgelegt wird. Die Verwendung einer längeren TP (länger als das SOF-Bit) erhöht die Fehlertoleranz gegen ein unbeabsichtigtes Einklinken (bei verfälschtem Identifier, langandauernder Störung die als SOF-Bit interpretiert wird, einen im falschen Slot gesendeten Rahmen etc.).

Nachdem wir die einzelnen Konzepte von BOT vorgestellt haben, definieren wir nun das Übertragungsverhalten modusbasierter Knoten mit BOT. Sei $v \in V_{MB}$ ein modusbasierter Knoten mit BOT und $Z \in \Theta$, $S \in Z$ mit $s \in S$ ein *Mode-Based Time Window* für das der Knoten v eine Slotzuordnung für den Mode $m \in M$ besitzt, d.h. $SA(m,s) = v$. Des Weiteren sei $f \in F$ ein sendebereiter Rahmen[57] dieses Knotens für den Slot s, d.h. $FA(S,s,m,v) = f$. Der Knoten v beginnt die Übertragung seines Rahmens $f \in F$,

A) falls v zwischen dem (lokalen[58]) Beginn des TSW des *Mode-Based Time Window* s und seinem TSP ein SOF-Bit empfangen hat und die ersten $|p_s|$ empfangenen Bits des gesendeten Rahmens mit der TP p_s übereinstimmen. Dann klinkt sich der Knoten v in die laufende Übertragung durch das Senden des $|p_s|+1$. Bits seines Rahmens f ein. Hierbei startet er die Übertragung dieses Bits mit dem Erreichen des $|p_s|+1$. Synchronizationssegments[59].

B) beim Erreichen seines TSP mit der Übertragung des SOF-Bit des Rahmens f, sofern das Medium zu diesem Zeitpunkt nicht belegt ist und nicht bereits aufgrund von Fall A eine Übertragung von f erfolgt oder bereits erfolgte.

Während der beschriebene Fall B für alle modusbasierten Knoten gilt, ist Fall A spezifisch für modusbasierte Knoten mit BOT und beschreibt den Ablauf beim Einklinken in eine laufende Übertragung. Die Abbildung 3.17 skizziert den Ablauf nach Fall A anhand eines Beispiels. Dargestellt sind die Ausgangspegel der Knoten v_1 und v_2 sowie der überlagerte Pegel auf dem Bus. In dem Beispiel erreicht Knoten v_1 seinen TSP vor v_2 und beginnt (da das Medium unbelegt ist und er keinen Übertragungsstart innerhalb seines TSW erkannt hat) zum Zeitpunkt t_0 seine Übertragung. Knoten v_2 erkennt den Übertragungsstart (fallende Flanke) innerhalb seines TSW (Zeitpunkt t_1), führt eine harte Synchronisation durch und startet die Erkennung der TP. Zum Zeitpunkt t_2 (Samplepoint von v_2) hat v_2 die TP erkannt und klinkt sich in die Übertragung ein, indem v_2 selbst zum Sender wird und zum Beginn des nächsten Synchronisationssegments (Zeitpunkt t_3) mit der Übertragung des vierten Bits seines Rahmens beginnt. Das von v_2 hierfür verwendete Bit-Timing basiert auf der durchgeführten harten Synchronisation (Zeitpunkt t_1) des von v_1 gesendeten SOF-Bits. Knoten v_1 verliert die Arbitrierung (Zeitpunkt t_4 – Samplepoint von v_1 für das fünfte Bit) und v_2 kann seinen Rahmen übertragen. Aufgrund der BOT-Erweiterung konnte sich der Knoten mit der höheren *Modus-Präferenz* bei der Arbitierung durchsetzen, und das trotz des Umstandes, dass v_1 seinen TSP deutlich früher (mehr als eine Bitzeit) als v_2 erreicht hat. Ohne BOT hätte v_2 die Arbitrierung verloren, weil das Medium zu dem Zeitpunkt zu dem v_2 seinen TSP erreicht hätte, bereits belegt gewesen wäre.

[56] Auch wenn die TP nur aus dem SOF-Bit besteht, so muss dieses Bit vollständig empfangen werden. Damit sich eine Störung auswirkt, muss diese bis zum Samplepoint anhalten. Dann wäre jedoch auch CAN ohne BOT-Erweiterung von der Störung betroffen und würde (nach sechs rezessiven Bits in Folge) einen Fehler signalisieren.

[57] Der Rahmen f sei derjenige sendebereite Rahmen des Knotens v mit der höchsten *Modus-Präferenz* für Slot s.

[58] Der entsprechende Zeitpunkt wird durch die lokale Uhr des Knotens festgelegt.

[59] Der Start dieses Segments basiert auf dem lokalen Bit-Timing des Knotens und somit letztendlich auf der harten Synchronisation beim Empfang des SOF-Bits und ggf. durchgeführter Bit-Resynchronisationen.

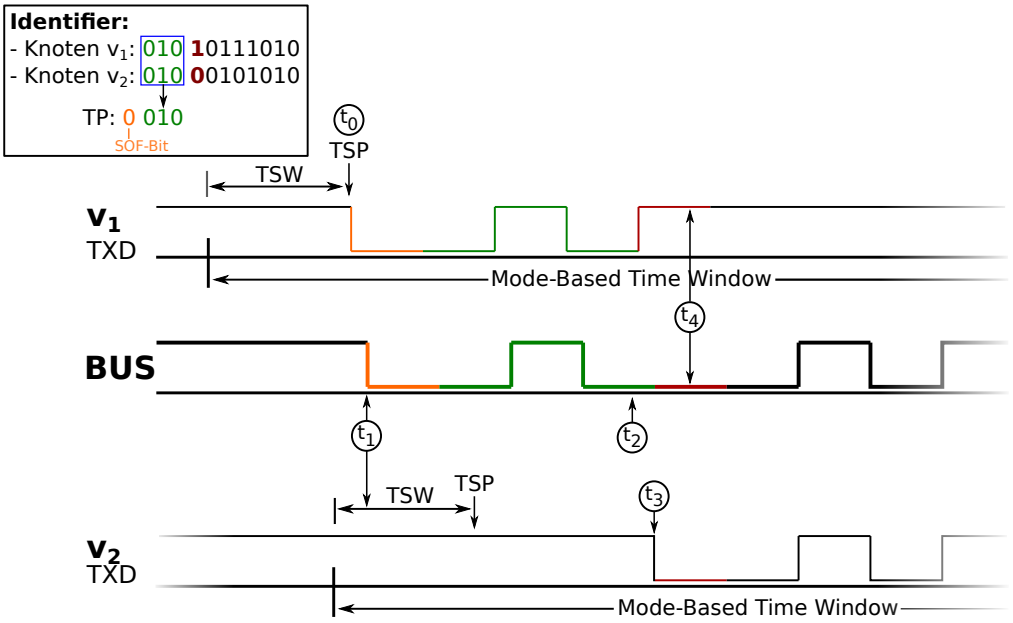


Abbildung 3.17: Einklinken in eine laufende Übertragung mit der BOT-Erweiterung.

Das UML-Zustandsdiagramm [Obj22] in Abbildung 3.18 spezifiziert das Verhalten eines Knotens mit BOT-Erweiterung bei der Übertragung eines Rahmens innerhalb eines *Mode-Based Time Window*. Um das Diagramm übersichtlich zu gestalten, werden nur die Abläufe bei einer Übertragung innerhalb des *Mode-Based Time Windows* dargestellt[60]. Sofort nach dem Start wird im Zustand `Wait` auf den Beginn des TSW des nächsten *Mode-Based Time Window* gewartet (Ereignis `TSW`). Falls beim Erreichen des TSW ein Rahmen für die Übertragung in dem *Mode-Based Time Window* vorliegt[61] (`getFrameForCurrentSlot` durchsucht den Nachrichtenpuffer), wird abhängig von dem aktuellen Zustand des Mediums entweder nach einer passenden TP gesucht (rot hinterlegt) oder darauf gewartet, dass eine gerade laufende Übertragung abgeschlossen wird (Zustand `WaitForFreeMedium`). Ist das Medium bis zum Erreichen des TSP immer noch belegt, wird der Rahmen verworfen (bzw. gemäß der gewählten Strategie behandelt) und auf das nächste *Mode-Based Time Window* gewartet.

Für die Beschreibung des Verhaltens führen wir die folgenden Ereignisse ein: Bei Beginn des Synchronisationssegments bzw. dem Erreichen des Samplepoint (basierend auf dem lokalen Bit-Timing) werden die Ereignisse `transmissionPoint` bzw. `samplePoint` erzeugt. Das Ereignis `MediumUpdate` wird bei einer Änderung des Medienzustandes ausgelöst. Der Zustand des Mediums kann mit der Variable `MediumState` abgefragt werden. `bitValue` erlaubt die Abfrage des Signalpegels zum letzten Samplepoint.

Im Zustand `SearchPreamble` wird auf den Start einer Übertragung innerhalb des TSW gewartet. Wird eine fallende Flanke erkannt, erfolgt eine harte Synchronisation und beim Erreichen des Samplepoint wird das Ereignis `samplePoint` generiert. Innerhalb des rot hinterlegten Bereichs erfolgt der Vergleich des Präfix des empfangenen Rahmens mit der TP. Bei Abweichungen wird in den Zustand `WaitForFreeMedium` gewechselt. Sind die ersten $|p_s|$ Bits der Übertragung mit der TP identisch, klinkt sich der Knoten mit der Übertragung des $|p_s|+1$. Bits seines Rahmens in die laufende Arbitrierung ein. Wurde bis zum TSP keine Übertragung gestartet (deren Präfix mit p_s übereinstimmt) und das Medium ist zu

[60] Die (Re-)Synchronisation beim Empfang einer Referencemessage, harte Synchronisation und Bit-Synchronisationen sind nicht Bestandteil dieses Zustandsdiagramms.

[61] Die hier vorgestellte Umsetzung der BOT-Erweiterung erfordert, dass ein zu sendender Rahmen rechtzeitig vor Beginn des TSW übergeben wird.

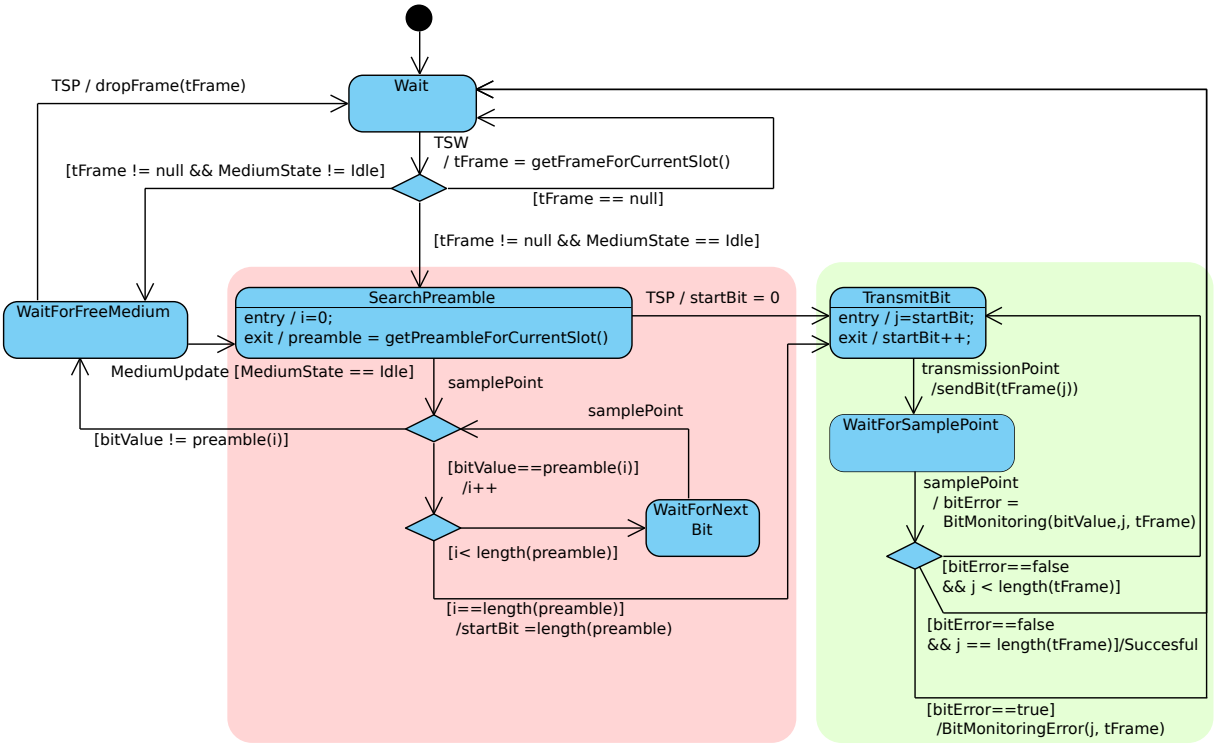


Abbildung 3.18: Verhaltensspezifikation der BOT-Erweiterung zur Übertragung von Rahmen in *Mode-Based Time Windows*.

diesem Zeitpunkt frei, beginnt der Knoten mit der Übertragung seines SOF-Bits (Zustandsübergänge `SearchPreamble` zu `TransmitBit`).

Innerhalb des grün hinterlegten Bereichs erfolgt die Umsetzung der bitweisen Übertragung sowie das Bitmonitoring. Abhängig von der Position innerhalb des Rahmens übernimmt die Funktion `bitMonitoringError`, falls erforderlich, die Fehlersignalisierung und entscheidet dabei auch, ob es sich in dem jeweiligen Fall lediglich um eine verlorene Arbitrierung handelt oder tatsächlich ein Übertragungsfehler vorliegt.

3.4.3 TTCAN FPGA Core

Eine Realisierung von BOT in Software, unter Verwendung handelsüblicher CAN-Controller, scheidet aufgrund der harten Echtzeitanforderungen sowie der notwendigen Modifikation der Logik für den Medienzugriff aus, welche für das Einklinken in eine laufende Arbitrierung erforderlich sind. Um die Funktionalität und Konzepte von BOT testen und evaluieren zu können, haben wir einen eigenen TTCAN-Controller entworfen, welcher BOT integriert. Hierfür verwenden wir einen Cyclone II EP2C5T144C6 FPGA (Field Programmable Gate Array) von Altera [Alt08]. Das Verhalten des TTCAN-Controllers selbst wurde in VDHL spezifiziert. Bei der Spezifikation haben wir besonderen Wert darauf gelegt, synthesefähigen VDHL-Code für die Zielplattform zu entwickeln[62], der sich direkt auf dem ausgewählten FPGA einsetzen lässt. Daher steht einer Verwendung dieses Designs auf realer Hardware prinzipiell nichts entgegen.

Da der Aufbau eines Hardwareprototypen für die Evaluation sowie entsprechende Messungen jedoch

[62] Das bedeutet, dass sowohl die Ressourcenbeschränkungen als auch die Einschränkungen hinsichtlich der maximalen Taktraten für den konkreten FPGA Typ eingehalten werden.

sehr aufwendig wären und den Zeitrahmen gesprengt hätten, erfolgte die Evaluation der Funktionalität mittels Simulationen. Hierzu haben wir neben dem TTCAN-Controller auch das Verzögerungsverhalten des CAN-Busses (inkl. CAN-Transceiver und Signallaufzeiten) in VHDL beschrieben. Dies erlaubt es, in VHDL einen vollständigen CAN-Bus mit mehreren Knoten (TTCAN-Controllern samt CAN-Transceiver) zu spezifizieren und deren Verhalten zu simulieren. Auf diese Weise konnten wir BOT in einem simulierten CAN-Bus testen. Für die Simulationen selbst kam der VHDL-Simulator GHDL [Tri14] sowie die von Altera bereitgestellten Werkzeuge für die VHDL-Entwicklung (Quartus [Alt14b] und ModelSim [Alt14a]) zur Anwendung.

Zunächst betrachten wir den grundlegenden Aufbau unseres TTCAN-Controllers sowie dessen Realisierung. Danach folgt zunächst die Evaluation der Standard CAN-Funktionalität des Controllers und abschließend die Evaluation der BOT-Erweiterung im speziellen (Kapitel 3.4.4). Kapitel 3.4.5 fasst die Ergebnisse in einer abschließenden Bewertung zusammen.

Aufbau, Architektur und Implementierung

Der Fokus beim Design unseres TTCAN-Controllers liegt auf der Umsetzung der MAC-Funktionalität gemäß CAN-Standard. Gleichzeitig sollen geeignete Mechanismen zur Unterstützung der Synchronisation gemäß TTCAN-Level-1 und TTCAN-Level-2 bereitgestellt werden. Um die Komplexität zu reduzieren, verzichten wir auf die Implementierung der Fehlersignalisierung sowie Overload-Frames. Die BOT-Funktionalität ist in einem eigenen Modul gekapselt und kann bei der Synthese aktiviert oder deaktiviert werden. Somit ist ein Vergleich zwischen dem Verhalten mit und ohne BOT hinsichtlich des Medienzugriffs sowie den resultierenden Auswirkungen auf das *Fast Mode-Signaling* möglich. Abbildung 3.19 gibt einen Überblick über die Architektur des TTCAN-Controllers und dessen funktionalen Komponenten. Die mit gestrichelten Linien eingezeichneten Komponenten wurden nicht oder nur rudimentär umgesetzt, da es sich hierbei nicht um essentielle Funktionen im Hinblick auf die Kommunikation handelt, sondern lediglich um Anbindungen für externe oder innerhalb des FPGAs synthetisierte Mikrocontroller.

Der TTCAN-Controller ist als Schichtenarchitektur realisiert. Die einzelnen Schichten bzw. deren Komponenten lassen sich hinsichtlich ihrer Funktionalität CAN bzw. TTCAN zuordnen. Die Zuordnung ist in der Abbildung 3.19 kenntlich gemacht. Auch die Architektur des TTCAN-Controllers spiegelt den Umstand wider, dass TTCAN als Protokoll selbst auf CAN aufsetzt und dessen MAC-Funktionalität verwendet. Im Folgenden beschreiben wir die einzelnen funktionalen Komponenten sowie Besonderheiten bei deren Umsetzung, beginnend mit den Komponenten, welche die Funktionen für das CAN-Protokoll implementieren.

Das komplette FPGA-Design arbeitet intern mit einem Takt von 100 MHz. Die Komponente `QuantaTimerLogic` generiert aus diesem Takt das periodische Signal `timeQuantaEnd`, dessen Periode der Länge eines Zeitquantums entspricht. Zusätzlich stellt diese Komponente einen zentralen Zähler zur Verfügung, welcher mit jedem Takt erhöht wird. Dieser Zähler übernimmt die Aufgabe der Network Time Unit (NTU), fungiert als knotenlokale Uhr und bildet die Grundlage für die Ermittlung von Zeitstempeln.

Zu den Aufgaben der `BitTimingLogic`-Komponente gehören alle Funktionen des Medienzugriffs, wie die Abtastung des Mediums, die Erkennung eines potentiellen Übertragungsbeginns sowie die Durchführung der harten Synchronisation bzw. Bit-Resychronisation. Neben dem Empfang koordiniert diese Komponente auch die Übertragung einzelner Bits, unter Berücksichtigung des Bit-Timings. Hierzu sind der Empfangsprozess `RXProcess` und der Sendeprozess `TransmitterProcess` eng miteinander gekoppelt. Bei einem Übertragungsstart erfolgt eine harte Synchronisation des Empfangsprozesses. Diese wird entweder durch eine fallende Flanke (eines empfangenen SOF-Bits) oder durch den Beginn einer eigenen Übertragung (Signal `nodeIsTransmitter`) ausgelöst – abhängig davon, welches Ereignis (zuerst) eintritt. Die harte Synchronisation startet das Bit-Timing mit dem Ende des Synchronisationssegments neu, sodass die fallende Flanke innerhalb dieses Segments liegt. Am Ende jeder Bitzeit (zu

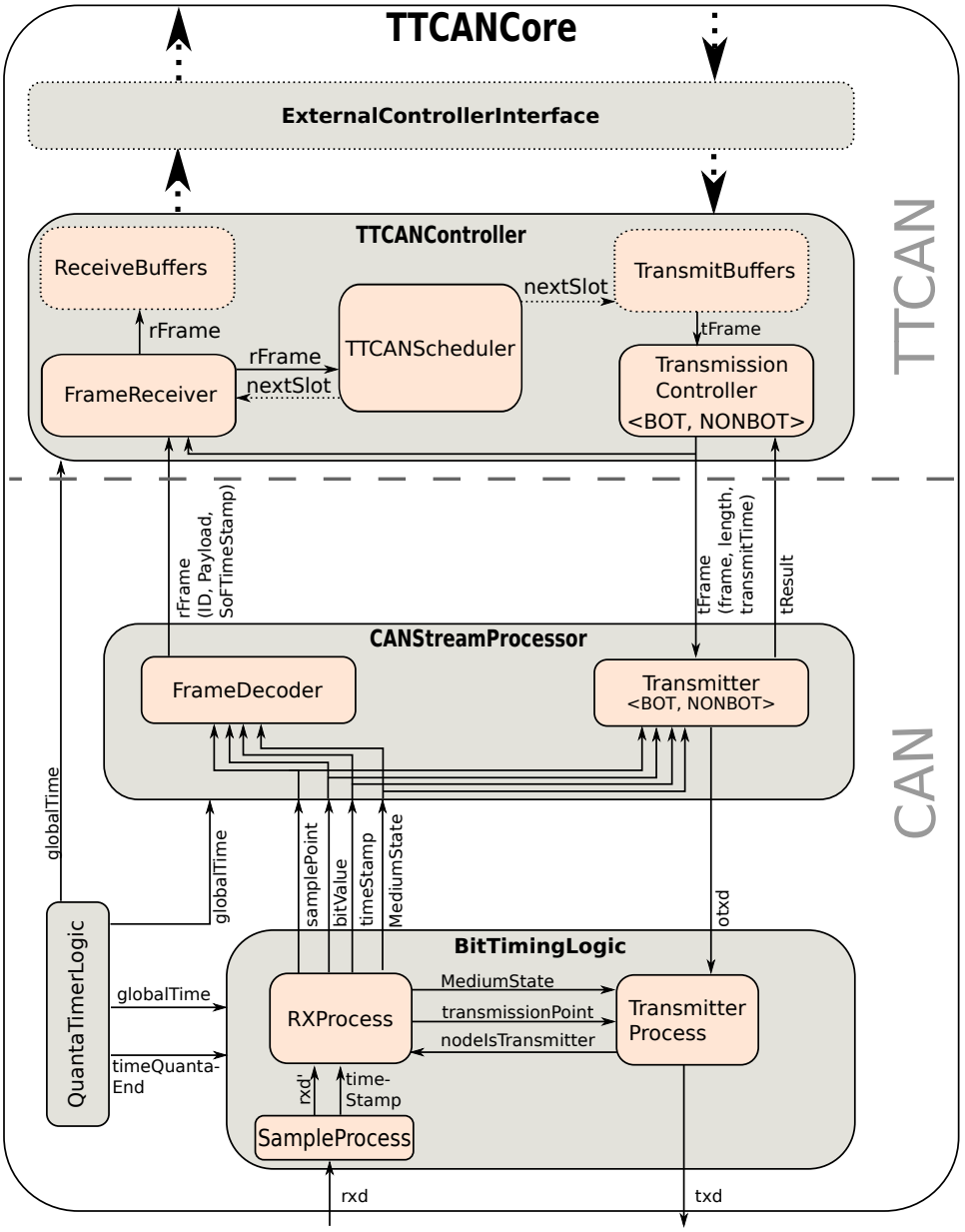


Abbildung 3.19: Architektur des entworfenen TTCAN-Controllers.

Beginn des Synchronisationssegments) signalisiert der `RXProcess` dem Übertragungsprozess über das Signal `transmissionPoint`, dass dieser das nächste zu übertragende Bit anlegen kann[63]. Die Kopplung der beiden Prozesse `RXProcess` und `TransmitterProcess` stellt zudem sicher, dass Bit-Resynchronisationen (bei der Erkennung von *Early Edges*[64]) auch von dem Sendeprozess berücksichtigt werden, sodass sich Korrekturen von Phasenfehlern auch auf die Übertragungszeitpunkte der nachfolgenden Bits auswirken. Während einer laufenden Übertragung (unabhängig davon, ob der Knoten

[63] Die Verzögerung zwischen dem Anlegen des `transmissionPoint`-Signals und dem Anlegen des neuen Bits beträgt einen Takt. Ob der Knoten ein Bit anlegt hängt davon ab, ob er in dieser Bitzeit als Sender fungiert oder nicht (auch ein Empfänger eines Rahmens wird im Acknowledge-Feld zum Sender).

[64] Fungiert ein Knoten selbst als Sender erfolgt keine Korrektur von *Late Edges*, um zu verhindern, dass ein Bit länger als eine Bitzeit am Bus anliegt (vgl. Kapitel 3.1.1).

Sender oder Empfänger ist) wird hierzu bei jeder fallenden Flanke[65] durch `RXProcess` geprüft, ob eine Bit-Resynchronisation erforderlich ist und diese, falls notwendig durchgeführt (vgl. Kapitel 3.1.1). Des Weiteren interpretiert `RXProcess` beim Erreichen des Samplepoints den anliegenden Buspegel und bestimmt, ob ein dominantes oder rezessives Bit übertragen wurde. Dieses Bit wird dann an den übergeordneten `CANStreamPocessor` weitergereicht. Zusätzlich bestimmt der `RXProcess` anhand des Buspegels den aktuellen Medienzustand (`idle` oder `busy`).

Die Abtastung des Pegels des `RXD`-Pins des angeschlossenen CAN-Transceivers erfolgt mit einer Frequenz von 100 MHz durch den `SampleProcess`. Im Gegensatz zu einer einzigen Abtastung am Anfang des Zeitquantums können somit exakte Zeitstempel, z. B. bezogen auf den Übertragungsstart von SOF-Bits, ermittelt werden. Zusätzlich handelt es sich um eine Maßnahme zur Vermeidung von Metastabilitäten, welche in der Praxis Probleme bereiten können. Metastabilitäten [Alt09] resultieren häufig aus der synchronen Abtastung eines asynchronen Signals, wie dies bei dem `RXD`-Signal der Fall ist und können auftreten, wenn die Abtastung zu einem Zeitpunkt erfolgt, indem das Signal gerade von Low auf High oder umgekehrt umschaltet. Damit der jeweilig anliegende Wert korrekt erfasst werden kann, muss dieser für eine bestimmte hardwarespezifische Zeitspanne vor und nach der Taktflanke, mit der die Abtastung erfolgt, stabil sein. Ist dies nicht gewährleistet, kann das abgetastete Signal innerhalb des FPGAs temporär auf einem Pegel zwischen High und Low liegen, bevor sich dieser stabilisiert, d.h. das Signal seinen endgültigen Wert angenommen hat. Wird das Signal direkt weiterverarbeitet, z. B. um innerhalb eines Automaten einen Zustandswechsel auszulösen, können sich hieraus schwer zu lokalisierende Fehler ergeben, da sich der Wert des Signals aufgrund der Stabilisierung bis zum nächsten Verarbeitungsschritt verändert haben kann. Um diese Art von Fehler zu vermeiden versucht man sicherzustellen, dass sich der Wert stabilisiert, bevor eine auf diesem Wert basierende Verarbeitung erfolgt.

Die Abbildung 3.20 aus [Alt09] verdeutlicht die Ursachen und Auswirkungen von Metastabilitäten anhand von Signalverläufen. In dieser Abbildung wechselt der Wert des Input Signals (zweites Signal von oben) während der Abtastung (steigende Taktflanke des Clock-Signals). Für eine sichere Abtastung müsste das Signal jedoch über die Zeitspanne $t_{SU} + t_H$ stabil sein. Die beiden unteren Signalverläufe zeigen mögliche Ergebnisse dieser Abtastung. Beide führen dazu, dass das Signal zunächst einen Wert zwischen High und Low annimmt, bis sich schließlich ein stabiler Wert einstellt. Dies ist entweder der alte oder der neue Wert des Eingangssignals, wie die beiden Ausgangssignale A und B in ihrem Verlauf illustrieren. Ein vorheriges Auslesen und Verarbeiten, z. B. in der Zeitspanne t_{CO}, führt unter Umständen zu dem beschriebenen Fehlerbild.

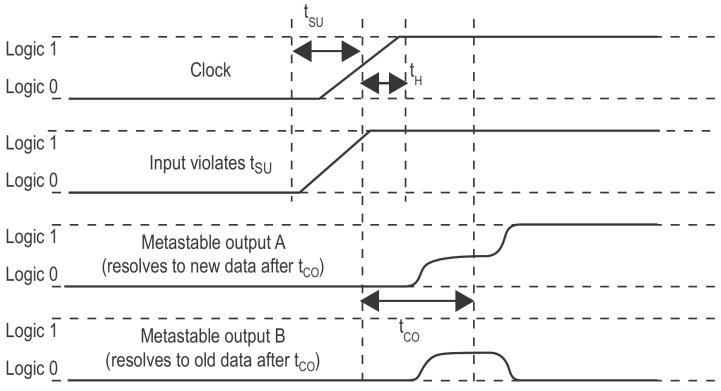


Abbildung 3.20: Metastabilität bei der Abtastung asynchroner Signale [Alt09].

[65] Eine fallende Flanke wird durch den Vergleich zwischen dem Buspegel beim letzten Samplepoint und dem aktuell (zu Beginn des Zeitquantum) abgetasteten Buspegel ermittelt.

Um Metastabilitäten innerhalb der synchron arbeitenden Zustandsautomaten unserer `BitTiming-Logic`-Komponente zu vermeiden, wird das asynchrone Eingangssignal *einsynchronisiert*. Hierzu wird das Signal nach dem Abtasten in eine mehrstufige FlipFlop Kaskade geladen (Abbildung 3.21), die ein Schieberegister bildet. Mit jeder Taktflanke wird der gespeicherte Signalwert von einem FlipFlop zum nächsten weitergereicht. Diese zusätzliche Verzögerung erlaubt es einem potentiell metastabilen Signal mit hoher Wahrscheinlichkeit einen stabilen Zustand anzunehmen, bevor dieses weiterverarbeitet wird. Unsere Implementierung nutzt ein Schieberegister der Länge drei, wodurch sich eine Verzögerung von $3 \cdot d_{clock} = 30\,\text{ns}$ ergibt, bis ein Eingangssignal dieses durchlaufen hat. Das Eingangssignal für den `RXProcess` (welches den Buspegel des CAN-Busses repräsentiert) wird jeweils zu Beginn jedes Zeitquantums durch die Übernahme des aktuellen Signalwertes des letzten FlipFlops gebildet.

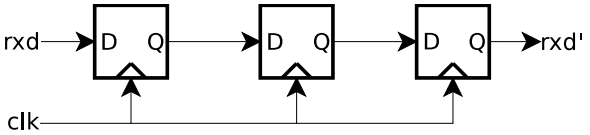


Abbildung 3.21: Kaskade von D-FlipFlops zur Vermeidung von Metastabilitäten beim Einlesen asynchroner Signale.

Um die TTCAN-Synchronisation (Level-2) möglichst exakt zu realisieren, muss die fallende Flanke des SOF-Bits sehr genau erfasst werden. Hierfür bietet die Abtastung mit 100 MHz ideale Voraussetzungen. Die Erfassung der fallenden Flanken erfolgt durch den Vergleich der abgetasteten Werte der FlipFlops 2 und 3. Wird eine fallende Flanke erkannt speichert der `SampleProcess` den aktuellen Zeitstempel[66], korrigiert um die konstante Durchlaufzeit der ersten FlipFlop Stufe. Dieser Zeitstempel wird `RXProcess` bereitgestellt. Bei dem mit dem abgetasteten Wert übergebenen Zeitstempel handelt es sich stets um den Zeitpunkt der zuletzt erfassten fallenden Flanke. `RXProcess` speichert den Zeitstempel der fallenden Flanke bei einem SOF-Bit und übergibt diesen bei jedem erfolgreich empfangenen Rahmen an den `TTCAN-Controller`. Handelt es sich bei dem Rahmen um eine Referencemessage, dient dieser Zeitstempel für die Durchführung der (Re-)Synchronisation der lokalen Uhren. *Early-Timestamping* hat natürlich keinen Einfluss auf Ungenauigkeiten, die aus dem Propagation-Delay oder den Signalverzögerungen der CAN-Transceiver resultieren. Diese Form der Implementierung erfüllt die Anforderungen an einen TTCAN-Controller des Level-2 bezüglich der Erfassung des Empfangszeitpunktes des SOF-Bits einer Referencemessage.

Wird der Beginn einer Übertragung erkannt, übergibt der `RXProcess` die einzelnen empfangenen Bits beim Erreichen des Samplepoint an den `CANStreamProcessor`. Der `FrameDecoder` decodiert das Rahmenformat und extrahiert Identifier und Payload, entfernt Stuffing-Bits und überprüft das Rahmenformat sowie die Checksumme. Wurde der Rahmen fehlerfrei empfangen, wird dieser zusammen mit den zusätzlich gesammelten Informationen (Zeitstempel der fallenden Flanke des SOF-Bits) an den `TTCANController` übergeben. Der `CANStreamProcessor` enthält neben dem `FrameDecoder` auch den `Transmitter`-Prozess. Dieser ist für die Übertragung eines CAN-Rahmens verantwortlich und übergibt jeweils das nächste zu sendende Bit an den untergeordneten `TransmitterProcess` der `BitTimingLogic`-Komponente. Dieser sendet das Bit unmittelbar[67], sofern das Medium gerade frei ist. Ist das Medium belegt wird das Bits gesendet, sobald der `RXProcess` dies mittels des Signals `transmissionPoint` anfordert (d.h. sobald das nächste Synchronissationssegment startet). Der `Transmitter` des `CANStreamProcessor` implementiert auch das Bit-Monitoring, indem er das gesendete Bit mit dem zum Samplepoint anliegenden Signalpegel vergleicht. Bei der aktuellen Implemen-

[66] Der Vorgang zu einem zeitkritischen Ereignis den dazugehörenden Zeitstempel mit möglichst geringer Verzögerung zu erfassen, wird auch als *Early-Timestamping* [BCG09] bezeichnet.

[67] Mit dem Start des nächsten Timequanta.

tierung wird die Übertragung bei einer Abweichung zwischen gesendetem und empfangenem Bit einfach abgebrochen. Gemäß CAN-Standard ist dieses Verhalten jedoch nur innerhalb des Arbitrierungsfeldes korrekt, außerhalb dieses Bereiches müsste eine Fehlersignalisierung erfolgen, die jedoch aktuell nicht implementiert ist.

Für den `Transmitter` des `CANStreamProcessor` werden zwei unterschiedliche Implementierungen bereitgestellt. Die erste Variante setzt das Standard CAN-Verhalten um: Sobald dem `Transmitter` von der übergeordneten Schicht ein CAN-Rahmen für den Versand übergeben wird (beim Erreichen des TSP) und das Medium ist frei, beginnt dieser mit der Übertragung. Ist das Medium bereits belegt, erfolgt eine Meldung an die darüberliegende Schicht.

Die zweite Version implementiert die BOT-Erweiterung gemäß Abbildung 3.18. Hierbei wird der zu übertragende Rahmen bereits zu Beginn des konfigurierten TSW an den `Transmitter` übergeben. Zusätzlich hierzu wird auch der geplante TSP sowie die Transmission Start Preamble (TP) übergeben. Ist das Medium bis zum Erreichen des TSP frei, beginnt der `Transmitter` die Übertragung des Rahmens zum geplanten TSP. Wird vor dem TSP aber nach dem Start des TSW, ein Übertragungsstart erkannt, so vergleicht der `Transmitter` die empfangene Bitsequenz mit der TP. Sind diese identisch, klinkt sich der Knoten, nach dem vollständigen Empfang der TP, in die Übertragung und damit den Wettbewerb um das Medium ein.

Der `TTCANController` realisiert die für das TTCAN-Protokoll benötigten Funktionen. Der `FrameReceiver` verarbeitet die empfangenen Rahmen und speichert diese in Puffern `ReceiveBuffers` – inkl. Metainformationen, wie Empfangszeitstempel, Time Window und Basiszyklus – zwischen. Referencemessages werden darüber hinaus zusätzlich an den `TTCANScheduler` weitergeleitet, damit dieser die (Re-)Synchronisation durchführen und einen neuen Basiszyklus starten kann. Time Windows und TSPs beziehen sich auf den Empfangszeitpunkt der Referencemessage[68]. Ist der Controller als Time Master konfiguriert übernimmt der `TTCANScheduler` auch den rechtzeitigen Versand der Referencemessage.

Die `TransmitBuffers` speichern ausgehende Nachrichten zwischen und stellen diese zu Beginn des jeweiligen Time Window dem `TransmissionController` zur Verfügung, welcher die Rahmen wiederum rechtzeitig an den `Transmitter` weiterreicht. Die Weitergabe erfolgt bei der BOT-Variante zu Beginn des TSW, bei der klassischen Variante zum TSP. Das geplante `ExternalControllerInterface` dient der Anbindung des TTCAN-Controllers an einen externen Mikrocontroller, entweder über einen externen Bus wie den Serial Peripheral Interface Bus (SPI [Mot00]) oder über einen internen Bus, wie zum Beispiel dem Wishbone-Bus [Ope10], für einen im FPGA synthetisierten Mikrocontroller.

3.4.4 Evaluation des TTCAN-Controllers und der BOT-Erweiterung

Um unseren TTCAN-FPGA Core zu validieren, wurden verschiedene Testsysteme in VHDL spezifiziert und diese mittels Simulationen hinsichtlich ihres korrekten Verhaltens evaluiert. In einem ersten Schritt erfolgte die Spezifikation von Testbenches, um die einzelnen funktionalen Komponenten des Designs automatisiert zu testen. Zum Test der Interaktion mehrerer TTCAN-Controller wurden weitere Testsysteme spezifiziert, die jeweils aus zwei Instanzen des TTCAN-FPGA Core (Knoten v_1 und v_2) bestehen, die über ein – ebenfalls simuliertes – Medium miteinander kommunizieren. Das simulierte Medium berücksichtigt sowohl die Signalverzögerungen der CAN-Transceiver als auch das Propagation-Delay, um die realen Abläufe möglichst genau nachzubilden. Das Verhaltensmodell des Mediums ist ebenfalls in VDHL beschrieben, sodass eine nahtlose Simulation ermöglicht wird.

[68] Dies gilt ebenfalls für den Time Master. Durch die Ausrichtung der Time Windows und TSPs an dem Empfangszeitpunkt der eigenen Referencemessage wird die Baseoffset bei der Ticksynchronisation zwischen Time Master und den anderen Knoten reduziert, weil auch der Time Master von der Signalverzögerung des (eigenen) CAN-Transceivers betroffen ist.

	Übertragungsrate	**1 MBit/s**	
CAN	Bitzeit d_{BIT}	1μs	
	Takt des CAN-Controllers	100 MHz	
	d_{CLK}	10 ns	
	Vorteiler zum Ableiten der Zeitquanten	5	
	Anzahl der Zeitquanten pro Bitzeit	20	
	Länge eines Zeitquantum d_q	$0,05\,\mu$s	
	Länge des Synchronization Segments	$1 \cdot d_q = 0,05\,\mu$s	
	Länge des Propagation Time Segments	$15 \cdot d_q = 0,75\,\mu$s	
	Länge des Phase 1 Time Segments	$2 \cdot d_q = 0,1\,\mu$s	
	Länge des Phase 2 Time Segments	$2 \cdot d_q = 0,1\,\mu$s	
	Segment Jump Width (SJW)	2	
	$d^{v_1,v_2,\min}_{PropagationDelay} = d^{v_2,v_1,\min}_{PropagationDelay}$	$0,100\,\mu$s	
	$d^{v_1,v_2,\max}_{PropagationDelay} = d^{v_2,v_1,\max}_{PropagationDelay}$	$0,100\,\mu$s	
	$d^{x,\max}_{txTransDelay}$, $d^{x,\min}_{txTransDelay}$	$0,080\,\mu$s	
	$d^{x,\max}_{rxTransDelay}$, $d^{x,\min}_{rxTransDelay}$	$0,080\,\mu$s	
	$d^{x,\max}_{txProcDelay}$, $d^{x,\min}_{txProcDelay}$	$0,010\,\mu$s	
	$d^{x,\min}_{rxProcDelay}$	$0,030\,\mu$s	
	$d^{x,\max}_{rxProcDelay}$	$0,040\,\mu$s	
TTCAN	Länge des Basic Cycle d_{Cycle}	1,66 ms	
	Anzahl der Time Windows	10	
	Länge der Time Windows $d_{mtw,s}$	$166 \cdot d_{BIT}$	
	Transmission Startpunkt d_{TSP}	$2 \cdot d_{BIT}$	
Constraints	Definition 3.1.1: $delay^V_{max}$	$0,31\mu$s	**Definitionen**
	Definition 3.1.1: $delay^V_{min}$	$0,30\,\mu$s	
	Definition 3.7: $d_{maxOffset,MB}$	$0,011\,\mu$s	
	Constraint 3.6: $d^{MAX'}_{maxOffset,MB}$	$0,85\mu$s	
	Constraint 3.2: d^{MIN}_{TSP}	$0,011\,\mu$s	
	Constraint 3.4: d^{MIN}_{mtw}	$164,31\,\mu$s	

Tabelle 3.5: Konfigurationsparameter und berechnete Constraints für die Testsysteme.

Das Verhalten der CAN-Transceiver wird in VHDL mit Hilfe konstanter Verzögerungsglieder realisiert. Hierbei nehmen wir zur Vereinfachung an, dass stets die maximale Signalverzögerung auftritt und dass diese sowohl für fallende als auch steigende Signalflanken identisch ist[69]. Unser Modell erlaubt unterschiedliche Signalverzögerungen für jeden simulierten Transceiver sowie unterschiedliche Verzögerungen bzgl. Senden und Empfangen. Das Progagation-Delay hängt in der Realität von der physikalischen Länge des elektrischen Leiters ab und kann in unserem Modell daher konfiguriert werden. In dem Evaluationsszenario haben wir als maximale Ausdehnung für unseren CAN-Bus 20 Meter gewählt und auf dieser Basis das maximale Propagation-Delay bestimmt. Da die Knoten identisch sind, sind auch die auftretenden Verzögerungen identisch, dies gilt z. B. auch für die Signalverzögerungen der CAN-Transceiver. Weil in unseren Simulationen die Verzögerungen durch die CAN-Transceiver sowie das Propagation-Delay konstant und bekannt sind, kann hier auch eine untere Grenze $delay^V_{min}$ für die Signalverzögerungen bestimmt und bei der Ermittlung der Constraints eingesetzt werden.

[69] Die maximalen Signalverzögerungen können den Datenblättern der Tranceiver entnommen werden. Bei einer Unterscheidung zwischen fallender und steigender Flanke sind wir von der maximalen Verzögerung ausgegangen.

Unsere Testsysteme gestatten ebenfalls die Simulation des Clock Skew zwischen den TTCAN-Controllern. Um die Funktionalität unseres TTCAN-Controllers zu demonstrieren, evaluieren wir zunächst die Standard CAN- und TTCAN-Funktionalitäten. Anschließend demonstrieren wir die Funktionalität und die Vorteile von BOT bei einer modusbasierten Übertragung mit einem für *Fast Mode-Signaling* (ohne BOT) zu hohen Tickoffset. Tabelle 3.5 stellt die bei den Simulationen verwendeten Konfigurationsparameter zusammen. Bei der Berechnung des maximalen Tickoffsets $d_{maxOffset,MB}$ gehen wir von einem Clock Skew von $r_{maxClockSkew} = 100\,\text{ppm}$ aus (vgl. Kapitel 3.3.2). Zusätzlich enthält die Tabelle weitere Kenngrößen unserer Implementierung, wie die Verarbeitungsverzögerung. Diese können aufgrund des synchronen Designs exakt bestimmt werden[70]. Für sämtliche Größen in der Tabelle die Bezug auf einen Knoten x nehmen, gilt $x \in \{v_1, v_2\}$.

Der untere Teil der Tabelle 3.1 (grau hinterlegt) wendet die Definitionen und Constraints aus Kapitel 3.2.2 an, um abgeleitete Größen zu bestimmen, die eine Beurteilung der Tauglichkeit der Konfiguration hinsichtlich *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* gestatten. Hierzu gehören die minimale und maximale Signalverzögerung (Definition 3.1.1) ebenso wie der maximale Tickoffset, welcher aufgrund der (Re-)Synchronisation bei dem gewählten Basiszyklus gemäß Definition 3.7 auftreten kann. In dunkelgrau hinterlegt sind die von den Constraints für die Konfiguration festgelegten Randbedingungen abgedruckt, welche eingehalten werden müssen, um z. B. die korrekte Funktionalität des *Fast Mode-Signaling* (ohne BOT) sicherzustellen (Constraint 3.6).

3.4.4.1 Evaluation des TTCAN-Controllers

Im Folgenden betrachten wir verschiedene Testszenarien zur Validierung der Standard CAN-Funktionalitäten unseres TTCAN-Controllers. Wir beginnen mit dem Empfang der Referencemessage und dem *Early-Timestamping* Mechanismus. Um das Verhalten zu dokumentieren, zeigen wir die Verläufe ausgewählter interner Signale der VHDL-Spezifikation und erläutern diese. Die Simulation selbst wurde mittels GHDL durchgeführt. Simuliert werden jeweils zwei TTCAN-Controller, die über ein (simuliertes) Medium interagieren. Die Signale von Knoten v_1 sind stets in blau dargestellt, die des Knotens v_2 in rot. Um die Abbildung möglichst übersichtlich zu gestalten, beschränken wir uns auf relevante Ausschnitte des kompletten Signalverlaufs. Längere Zeitspannen, in denen keine (für das Beispiel) relevanten Signalveränderungen stattfinden, werden durch Lücken repräsentiert. Der Knoten v_1 übernimmt in den folgenden Beispielen die Rolle des Senders und Time Masters.

Die Abbildung 3.22 zeigt den Empfang des SOF-Bits einer Referencemessage mit dem CAN-Identifier 4, gesendet von v_1. Die Signale `txd` und `rxd` zeigen jeweils den aktuell anliegenden Pegel der Sende- bzw. Empfangspins des jeweiligen TTCAN-Controllers. Diese Pins sind in der Realität mit den entsprechenden Ein- bzw. Ausgangspins des CAN-Transceivers verbunden. In unserem Szenario sendet v_1 zum Zeitpunkt t_0 sein SOF-Bit. Das von ihm gesendete Signal ist, aufgrund des *loop-delay*, an dessen eigenem `RXD`-Pin erst zum Zeitpunkt t_1 (nach 160 ns) sichtbar. Zum Zeitpunkt t_2 erreicht die fallende Flanke den `RXD`-Pin von v_2. Diese Gesamtverzögerung von 260 ns setzt sich aus der Sendeverzögerung des CAN-Transceivers von v_1 (80 ns), dem Propagation-Delay (100 ns) sowie der Empfangsverzögerung des CAN-Transceivers von v_2 zusammen (80 ns).

Bei den anderen Signalen von v_2 handelt es sich um interne Signale, die Bestandteil der `BitTiming-Logic` sind. Die fallende Flanke des Signals `syncSegment` zeigt an, dass es sich bei dem vorherigen Segment um ein Synchronisationssegment handelte. Eine steigende Flanke von `timeQuantaEnd` kennzeichnet den Beginn eines neuen Zeitquantums. Das Signal `synced_rxd` zeigt den Inhalt des Schieberegisters des `SampleProcess`, während `globalTime` die lokale Uhrzeit des Knotens repräsentiert und mit jedem Takt inkrementiert wird.

[70] Die Unterschiede in Bezug auf $d^{x,\min}_{rxProcDelay}$ und $d^{x,\max}_{rxProcDelay}$ ergeben sich aufgrund der Abtastung des asynchronen Eingangssignals mit einer Abtastrate von 100 MHz.

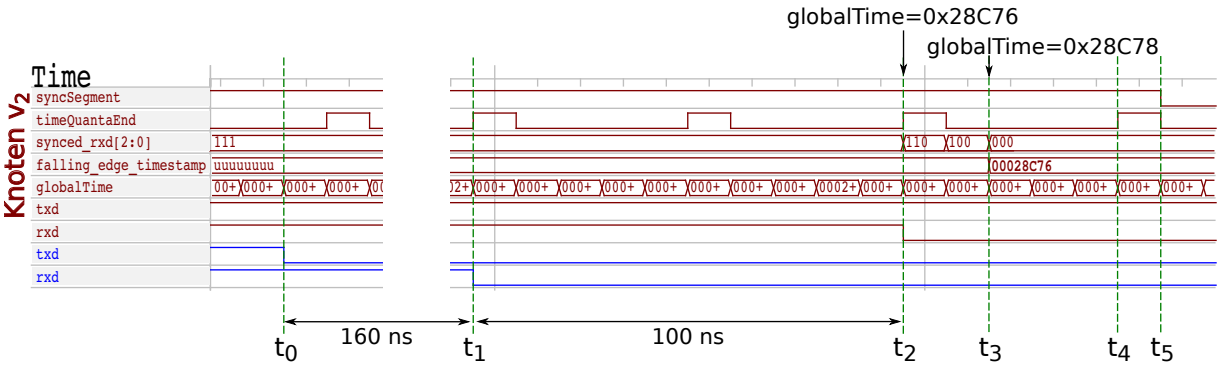


Abbildung 3.22: Early-Timestamping von Referencemessages zur Verbesserung der Synchronisationsgenauigkeit.

Zwischen dem Zeitpunkt t_2 und t_3 wird die empfangene fallende Flanke des SOF-Bits (Zeitpunkt t_2) mit jedem Takt um eine Position durch das Schiebregister (`synced_rxd`) weitergeschoben. Zum Zeitpunkt t_3 wird der Abtastwert an den `RXProcess` übergeben. Dieser setzt den Zeitstempel `falling_edge_timestamp` auf den Zeitpunkt des Empfangs der fallenden Flanke. Hierbei wird der Zeitpunkt t_2 verwendet und nicht der Zeitpunkt, zu dem das Signal an den `RXProcess` übergeben wird. Vergleiche hierzu den Wert des Signals `falling_edge_timestamp` zum Zeitpunkt t_3 (innerhalb des Signalverlaufs abgedruckt) mit dem Wert des globalen Zeitgebers zum Zeitpunkt t_2 (in schwarz über dem Signalverlauf dargestellt). Zum Zeitpunkt t_4 beginnt das nächste Zeitquantum. Aufgrund des Vergleichs des jetzt übergebenen Abtastwerts (`synced_rxd`) mit dem abgetasteten Wert beim letzten Samplepoint, erkennt die `BitTimingLogic` ein potentielles SOF-Bit und startet das Bit-Timing neu (harte Synchronisation). Hierdurch wird das vorherige Zeitquantum (zwischen den Zeitpunkten t_2 und t_4) zum Synchronisationssegment erklärt (Zeitpunkt t_5, fallende Flanke des Signals `syncSegment`).

Early-Timestamping erreicht, dass bei der Ticksynchronisation des TTCAN-Controllers der tatsächliche Empfangszeitpunkt der fallenden Flanke des SOF-Bits verwendet wird. Je höher die gewählte Abtastrate ist, desto genauer kann dieser bestimmt werden. Natürlich gelingt es nicht sämtliche Verzögerungen (CAN-Transceiver, Propagation-Delay) zu kompensieren, wie dieses Beispiel anschaulich illustriert. Handelt es sich in dem Beispiel um eine Referencemessage, so beträgt der resultierende Baseoffset aus der Synchronisation 100 ns (Zeitpunkte t_1 und t_2). Dies liegt darin begründet, dass beide Knoten identische Transceiver verwenden, sodass das *loop-delay* des Time Masters genau der gemeinsamen Signalverzögerung beider Transceiver bei der Übertragung an v_2 entspricht. Daher resultiert der Baseoffset in diesem Beispiel allein aus dem Propagation-Delay (aufgrund des *Early-Timestamping* und des Umstandes, dass der Time Master ebenfalls den Empfangszeitpunkt der Flanke des eigenen SOF-Bits (t_1) als Referenzzeitpunkt für seine Ticksynchronisation nutzt).

Die harte Synchronisation dient dazu das Bit-Timing so auszurichten, dass die nachfolgenden Flanken jeweils innerhalb des Synchronisationssegments empfangen werden. Die Abbildung 3.23 zeigt hierzu exemplarisch den weiteren Verlauf, der in Abbildung 3.22 gestarteten Übertragung. Gezeigt wird der Empfang einer steigenden Flanke zum Zeitpunkt t_0 durch v_2. Diese Flanke steht zum Zeitpunkt t_1 für die interne Verarbeitung zur Verfügung und liegt somit noch innerhalb des Synchronisationssegments[71].

[71] Das Synchronisationssegment endet mit dem Zeitpunkt t_1, somit steht der abgetastete Wert (nach Durchlaufen des Schieberegisters) gerade rechtzeitig für die Verarbeitung zur Verfügung. Die fallende Flanke von `syncSegment` (Zeitpunkt t_2) kennzeichnet das Zeitquantum als Synchronisationssegment.

Abbildung 3.23: Auswirkungen der harten Synchronisation auf nachfolgende Bits.

Der in Abbildung 3.23 gezeigte Anwendungsfall geht von einem idealen ClockSkew von 0 ppm zwischen den knotenlokalen Uhren aus. Ist der Clock Skew jedoch hinreichend groß, werden die auf das SOF-Bit folgenden Flanken nicht mehr innerhalb des Synchronisationssegments empfangen. In diesem Fall wird der ermittelte Phasenfehler durch die Anpassung der Phasen Segmente kompensiert (Bit-Resynchronisation, vgl. Kapitel 3.1.1). Abbildung 3.24 zeigt die Korrektur einer empfangenen *Late Edge*. Wir beschränken uns in Abbildung 3.24 auf die Darstellung der Signale von Knoten v_2, der die Bit-Resynchronisation durchführt.

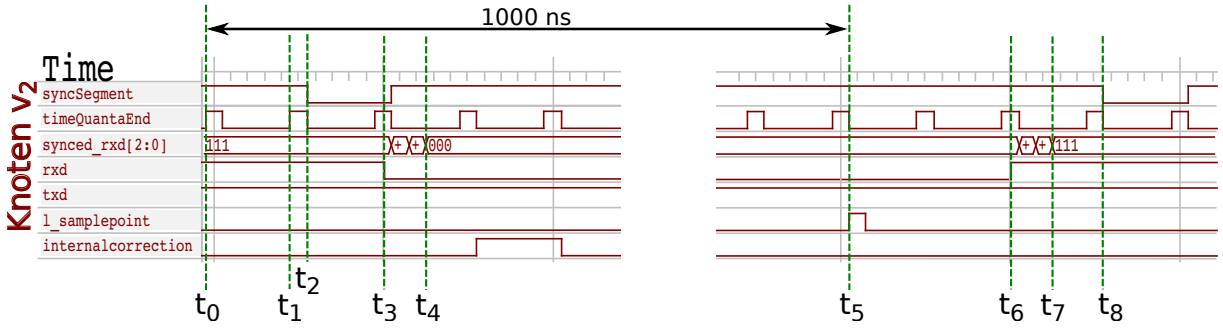


Abbildung 3.24: Bit-Resynchronisation bei einer *Late Edge* durch v_2.

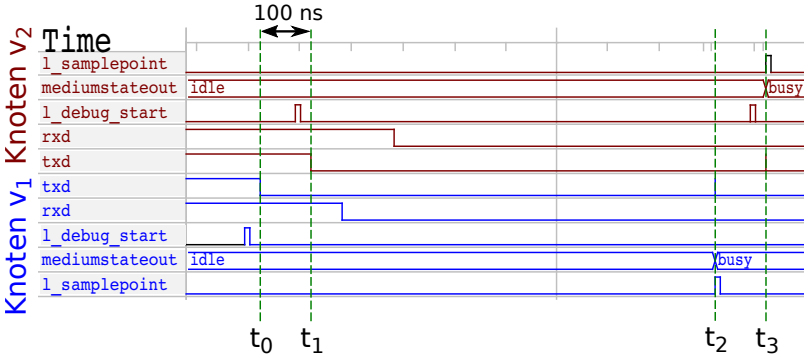
Zusätzlich zu den bereits bekannten Signalen enthält Abbildung 3.24 das Signal `l_samplepoint`. Die steigende Flanke dieses Signals markiert den Samplepoint, also den Zeitpunkt zu dem die Interpretation des anliegenden Signalpegels erfolgt. Die Zeitpunkte t_0 und t_1 kennzeichnen Beginn und Ende des Synchronisationssegments (erkennbar an der fallenden Flanke des Signals `syncSegment` zum Zeitpunkt t_2). Die zum Zeitpunkt t_3 empfangene fallende Flanke (`rxd`-Signal) steht jedoch erst zwei Zeitquanten nach dem Synchronisationssegment für die Verarbeitung zur Verfügung (Zeitpunkt t_4). Somit handelt es sich bei der empfangenen Flanke um eine *Late Edge* mit einem Phasenfehler von zwei Zeitquanten. Infolgedessen wird das erste Phasensegment um zwei Zeitquanten verlängert, sodass zwischen dem Beginn des Synchronisationssegments und dem Samplepoint (t_0 und t_5) insgesamt nicht 18, sondern 20 Zeitquanten (1000 ns) liegen (vgl. Tabelle 3.5). Durch diese Korrektur liegt die Flanke des nächsten Bits (Zeitpunkt t_6 bzw. t_7) wieder innerhalb des Synchronisationssegments (Zeitpunkt t_8). Das Listing 3.2 zeigt noch einmal die entsprechenden Log-Ausgaben des Zustandsautomaten des Prozesses `RxProcess` bei der Durchführung der Phasenkorrektur (Zeile 4).

Listing 3.2: Korrektur einer *Late Edge* durch Knoten v_2.

```
BitTimingLogic@32155ns: Potential SOF detected (dominant edge in idle region)!
BitTimingLogic@33005ns: SOF: Start of Frame detected at 3139 at end of sync segment
  (endtime) 3145
BitTimingLogic@34255ns: Resync needed, dominant edge outside sync segment
BitTimingLogic@34255ns: Late Edge detected! Updating quantaCounter applying correction (2
  timequanta)
```

Nun betrachten wir den Ablauf bei der Arbitrierung, wenn zwei Knoten nahezu gleichzeitig beginnen einen Rahmen zu übertragen (Abbildung 3.25). Knoten v_1 versucht einen Rahmen mit Identifier 7 zu senden, Knoten v_2 einen Rahmen mit dem Identifier 5. Wir gehen wieder von einem idealen Clock Skew aus; der Tickoffset zwischen beiden Knoten resultiert alleine aus der Signalverzögerung beim Empfang der Referencemessage, wie in Abbildung 3.22 gezeigt, und beträgt 100 ns (siehe oben). Zusätzlich zu den bereits bekannten Signalen wird der Verlauf des Signals `l_debug_start` dargestellt. Dessen steigende Flanke signalisiert den geplanten Übertragungsstart (TSP) gemäß der lokalen Uhr des jeweiligen Knotens. Das Signal `mediumstateout` repräsentiert den durch den Knoten detektierten momentanen Medienzustand.

(a) Übertragung des SOF-Bits.



(b) Übertragung des ersten abweichenden Bits des Identifier.

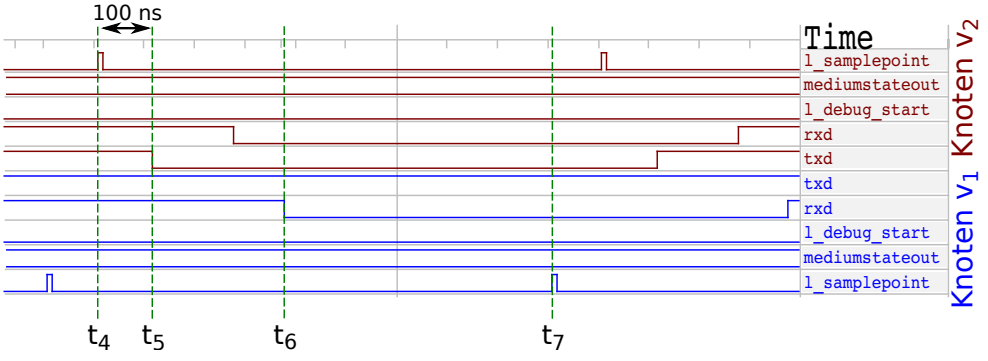


Abbildung 3.25: Ablauf der Arbitrierung mit einem Tickoffset von 100 ns im Detail.

Abbildung 3.25a zeigt den Beginn der Übertragung des SOF-Bits. Diese beginnt gemäß der lokalen Uhren der beiden Knoten gleichzeitig. Aufgrund des Tickoffsets bei der Synchronisation liegt der tatsächliche Übertragungsbeginn (Zeitpunkte t_0 und t_1) jedoch 100 ns auseinander. In diesem Beispiel ist

gut erkennbar, dass erst zum Samplepoint (Zeitpunkt t_2 bzw. t_3) die Knoten den anliegenden Buspegel interpretieren und damit das Medium als belegt erkennen (vgl. Kapitel 3.1.1).

Die Abbildung 3.25b zeigt die Übertragung des ersten Bits, in denen sich die beiden Identifier unterscheiden. Zum Zeitpunkt t_4 sendet v_1 einen rezessiven Pegel[72], während v_2 zum Zeitpunkt t_5 einen dominanten Pegel sendet. Diese überlagern sich auf dem Medium zu einem dominanten Pegel (siehe `rxd` Signal von v_1, Zeitpunkt t_6). Bei der Interpretation des Medienzustands zum Zeitpunkt t_7 erkennt v_1, dass der von ihm gesendete Pegel nicht mit dem empfangenen Pegel übereinstimmt und bricht die eigene Übertragung des Rahmens ab. Somit gewinnt v_2 die Arbitrierung und kann seinen Rahmen vollständig übertragen. Das Listing 3.3 zeigt die gekürzten Logausgaben von v_1, die den entsprechenden Ablauf noch einmal dokumentieren.

Listing 3.3: Knoten v_1 verliert die Arbitrierung.

```
TTCANController@3164405ns: TTCANScheduler: Enqueuing Frame with ID7 in slot 9
CANStreamProcessor_Transmitter@3169415ns: FT: Transmitting first bit
BitTimingLogic@3169455ns: Potential SOF detected (dominant edge in idle region)!
CANStreamProcessor_Transmitter@3169515ns: FT: Transmitting first bit
CANStreamProcessor_Transmitter@3181315ns: FT: Bits monitoring failed -> Arbitration lost
  or transmission error?
TTCANController@3181325ns: F_R: Result of Transmission approach: arbitrationfailed for
  FRAME ID 7
CANStreamProcessor_FrameDecoder@3215425ns: FD: Frame successfully received with ID 5 at
  (local SOF time)316949 current local time 321522
TTCANController@3215435ns: F_R: Valid Frame received with ID 5
```

Wir betrachten nun das gleiche Szenario noch einmal, allerdings mit einem Tickoffset von einer Bitzeit (d.h. 1 µs) zwischen v_1 und v_2. Die beiden TSPs der Knoten sind mit den Zeitpunkten t_0 und t_2 in Abbildung 3.26 markiert. Zum Zeitpunkt t_1 führt der Knoten v_2 eine harte Synchronisation auf die fallende Flanke des SOF-Bits von Knoten v_1 durch. Der TSP von Knoten v_2 (Zeitpunkt t_2) liegt jedoch vor dessen Samplepoint (Zeitpunkt t_3). Da die Interpretation des anliegenden Buspegels erst mit dem Samplepoint erfolgt, ist das Medium zum Zeitpunkt t_3 aus Sicht von v_2 noch unbelegt (Signal `mediumstateout`). Daher beginnt v_2 mit der Übertragung des SOF-Bits seines Rahmens. Das von v_2 verwendete Bit-Timing bezieht sich auf den Referenzzeitpunkt t_1. Der weitere Ablauf der Arbitrierung entspricht der Darstellung in Abbildung 3.25b und auch das Ergebnis ist identisch: Knoten v_2 gewinnt die Arbitrierung.

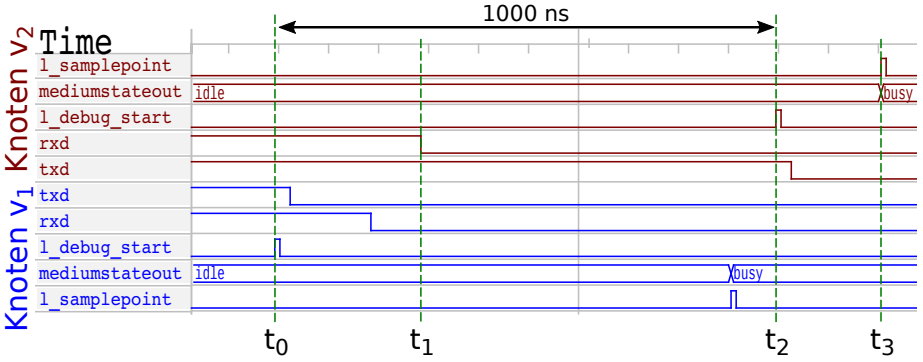


Abbildung 3.26: Start der eigenen Übertragung trotz des vorherigen Empfangs einer fallenden Flanke.

[72] In der Abbildung nur indirekt daran erkennbar, dass Zeitpunkt t_4 zwei Zeitquanten hinter dem Samplepoint von v_1 (Signal `l_samplepoint`) liegt.

Die absolute Abweichung der TSPs beider Knoten (steigende Flanke von `l_debug_start`) von einer Bitzeit (1 µs) liegt gerade noch innerhalb der oberen Schranke für den maximal zulässigen Tickoffset (bei dieser Konfiguration), die eine korrekte Funktionalität des *Fast Mode-Signaling* garantiert[73]. Die obere Schranke für den maximalen Tickoffset beträgt für dieses Szenario (Definition 3.11, Ausdruck (3.1))

$$\begin{aligned} d^{MAX,s}_{maxOffset,MB} &= \min_{v_1 \in V_s, v_2 \in V_s} \{ d^{v_2}_{PropSeg} + d^{v_2}_{PhaseSeg1} + delay_{\min}(v_1, v_2) | v_1 \neq v_2 \} \\ &= 17 \cdot d_q + 300\,\text{ns} = 1,150\,\mu\text{s}. \end{aligned}$$

3.4.4.2 Evaluation der BOT-Erweiterung

Wie unsere vorherigen Testszenarien belegen, arbeitet der TTCAN-Controller entsprechend der CAN-Spezifikation und auch die Mechanismen für die harte Synchronisation sowie die Bit-Resynchronisation funktionieren. Die Verwendung von *Early-Timestamping* hilft bei der Erhöhung der Synchronisationsgenauigkeit – wie das Beispiel aus Abbildung 3.22 illustriert – durch die Reduktion des Baseoffsets beim Empfang der Referencemessage.

In unserem Testaufbau beträgt die obere Schranke für den maximal zulässigen Tickoffset, für den eine zuverlässige Funktion des *Fast Mode-Signaling* sichergestellt ist, 1,150 µs. Für die Evaluation unserer BOT-Erweiterung wählen wir nun ein Szenario, bei dem diese maximale obere Schranke verletzt wird und simulieren das Verhalten einmal mit und einmal ohne BOT-Erweiterung und vergleichen die Resultate. Die Tabelle 3.6 zeigt die gewählten Konfigurationsparameter für die BOT-Erweiterung, alle anderen Parameter entsprechen weiterhin den Vorgaben aus Tabelle 3.5. Die in der Tabelle angegebene Transmission Start Preamble (TP) verwendet die Repräsentation des Rahmens auf dem Medium und enthält sowohl das SOF-Bit als auch Stuffing Bits (jeweils blau bzw. rot hervorgehoben). Dies gilt ebenso für die dargestellten CAN-Identifier, welche im Rahmen des Tests, von den Knoten v_1 und v_2, verwendet werden. Der unterstrichene Teil der Identifier entspricht der TP.

Szenario	Tickoffset	$2,1\mu$s
	Transmission Start Window (TSW)	$d_{TSW} = 5 \cdot d_{Bit}$ $= 5\mu\text{s} = d_{TSP}$
	Länge der Transmission Start Preamble	11 Bit
	Transmission Start Preamble	**0** 0000 **1** 00001
	CAN-Identifier von Knoten 1	7
	→ Bitrepräsentation inkl. SOF- und Stuffing-Bits	**0** 0000 **1** 00001 11
	CAN-Identifier von Knoten 2	5
	→ Bitrepräsentation inkl. SOF- und Stuffing-Bits	**0** 0000 **1** 00001 01

Tabelle 3.6: Transmission Start Preamble für die BOT-Erweiterung.

In unserem Szenario konkurrieren die beiden Knoten v_1 und v_2 um das gleiche *Mode-Based Time Window*. Knoten v_1 plant die Übertragung eines Rahmens mit dem CAN-Identifier 7, v_2 die Übertragung eines Rahmens mit dem Identifier 5. Der (gewählte) Tickoffset zwischen den beiden Knoten v_1 und v_2 beträgt 2,1 µs. Das Szenario wurde so konstruiert, dass der Knoten v_2 bezüglich der Arbitrierung im Nachteil ist, d.h., v_2 erreicht seinen TSP erst 2,1 µs nachdem v_1 seinen TSP bereits erreicht und seine Übertragung gestartet hat.

[73] Wobei dies nur gilt, weil wir in unserer Simulation die minimale Signalverzögerung kennen, und sich diese zu unseren Gunsten auswirkt. Bei einer minimalen Signalverzögerung kleiner als 150 *ns*, wäre eine korrekte Funktion nicht mehr gewährleistet.

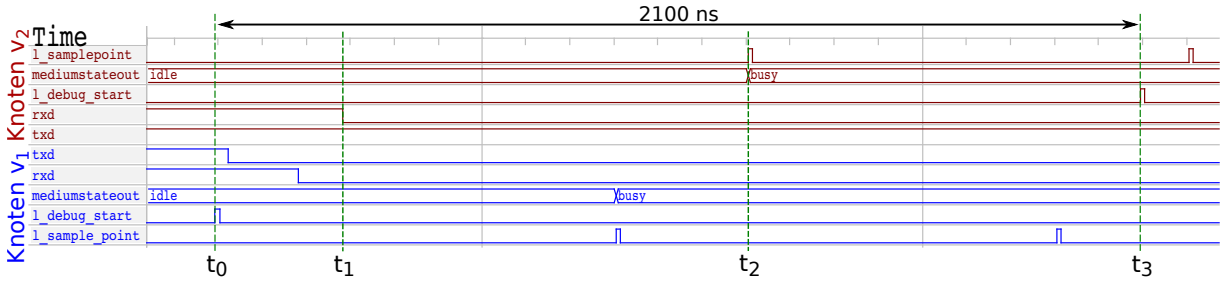


Abbildung 3.27: Fehlerhafte Ausführung des *Fast Mode-Signaling* aufgrund eines zu großen Tickoffsets.

Die Abbildung 3.27 zeigt das entsprechende *Mode-Based Time Window*, in welchem beide Knoten versuchen ihren jeweiligen Rahmen zu übertragen. Die geplanten Zeitpunkte der Übertragung (TSPs) der jeweiligen Knoten sind mit t_0 bzw. t_3 markiert und erkennbar an der steigenden Flanke des Signals `l_debug_start`. Diese beiden Zeitpunkte sind gemäß der jeweiligen knotenlokalen Uhren der beiden Knoten identisch, absolut betrachtet liegen diese jedoch $2,1\,\mu s$ auseinander. Der Knoten v_1 beginnt zum Zeitpunkt t_0 mit seiner Übertragung und einen Takt später liegt das Ausgangssignal am `TXD`-Pin des CAN-Transceivers an. Knoten v_2 führt beim Erfassen der fallenden Flanke des SOF-Bits eine harte Synchronisation durch und startet sein Bit-Timing neu (Signal `rxd` von v_2, Zeitpunkt t_1). Der Samplepoint von v_2 ist in der Abbildung mit t_2 markiert. Zu diesem Zeitpunkt interpretiert v_2 den anliegenden Buspegel und erkennt das Medium als belegt (Signal `mediumstateout`). Der Knoten v_2 erreicht seinen TSP zum Zeitpunkt t_3, da dass Medium jedoch bereits belegt ist, verzichtet v_2 auf die Übertragung seines Rahmens in diesem *Mode-Based Time Window*. Daraufhin gewinnt v_1 den Wettbewerb um das *Mode-Based Time Window*, obwohl sein Rahmen eine niedrigere *Modus-Präferenz* als der Rahmen von v_2 besitzt.

Das Listing 3.4 zeigt die Logausgaben von Knoten v_1 und v_2 und belegt das dargestellte Verhalten noch einmal.

Listing 3.4: Logausgabe von Knoten v_1 und v_2 bei zu großem Clockoffset.

```
Knoten v1
TTCANController@3164405ns: TTCANScheduler: Slot9
TTCANController@3164405ns: TTCANScheduler: Enqueuing Frame with ID7 in slot 9
CANStreamProcessor_Transmitter@3217315ns: FT: Transmission of frame successfull
TTCANController@3217325ns: F_R: Result of Transmission approach: success for FRAME ID 7


Knoten v2
TTCANController@3166505ns: TTCANScheduler: Slot9
TTCANController@3166505ns: TTCANScheduler: Enqueuing Frame with ID5 in slot 9
CANStreamProcessor_FrameDecoder@3216575ns: FD: Frame successfully received with ID 7 at
  (local SOF time)316749 current local time 321437
```

Nun betrachten wir das identische Szenario mit aktivierter BOT-Erweiterung. Da der Tickoffset zwischen den beiden Knoten kleiner als das eingestellte TSW ist, muss sich Knoten v_2 nach der Erkennung der TP in die Arbitrierung einklinken und diese dann in deren weiteren Verlauf gewinnen. Dementsprechend muss als Ergebnis innerhalb dieses *Mode-Based Time Window* der Rahmen mit dem CAN-Identifier 5 übertragen werden. Die Abbildungen 3.28 und 3.29 zeigen, wie die BOT-Erweiterung im Detail arbeitet. Die Ausgangssituation ist identisch, wieder liegen die TSPs der beiden Knoten $2,1\,\mu s$ auseinander (vgl. Zeitpunkte t_0 und t_3 in Bezug auf das Signal `l_debug_start`). Der Samplepoint von v_2 ist, wie zuvor, mit t_2 markiert. Zu diesem Zeitpunkt interpretiert v_2 den anliegenden Buspegel und erkennt das Medium wieder als belegt. Da sich die fallende Flanke des SOF-Bits (Zeitpunkt t_1) jedoch innerhalb des

TSW befindet, startet v_2 den Detektor für die TP (steigende Flanke des Signals `preamblematcher-start`, unmittelbar nach dem Samplepoint[74] (t_2)). Der Knoten v_2 erreicht seinen TSP zum Zeitpunkt t_3, da das Medium jedoch bereits belegt ist (Signal `mediumstateout`), sendet v_2 selbst zunächst nicht – erkennbar an dem durchgängig rezessiven Pegel des Signals `txd` von v_2.

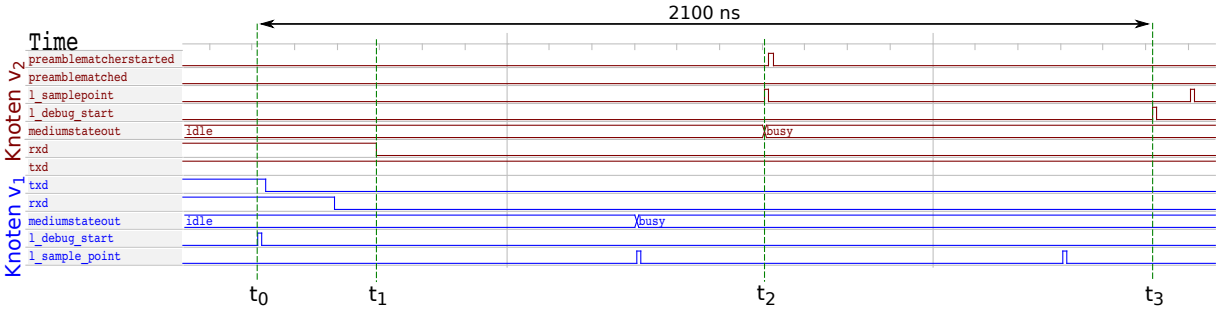


Abbildung 3.28: Ablauf einer Übertragung mit der BOT-Erweiterung.

Abbildung 3.29 beginnt mit dem Empfang und der Interpretation des letzten Bits der TP durch v_2 (steigende Flanke des Signals `l_sample_point`, welches den Samplepoint markiert, ein Takt vor Markierung t_4). Vor diesem Zeitpunkt wurden sämtliche Bits nur durch den Knoten v_1 gesendet. Direkt nach der Interpretation des Pegels des Signals `rxd` wird die vollständige Erkennung der TP signalisiert (Signal `preamblematched`, Zeitpunkt t_4). Beginnend mit dem nächsten Bit (genauer dessen Synchronisationssegment, Signal `l_transmissionpoint`, vor Markierung t_5) klinkt sich v_2 in die Arbitierung ein und sendet selbst. Das erste von v_2 gesendete Bit ist eine Null, d.h. v_2 sendet einen dominanten Pegel (Signal `txd` Markierung t_5). Zum Zeitpunkt t_6 interpretiert Knoten v_1 den Pegel seines `rxd`-Signals und stellt mittels Bitmonitoring fest, dass der anliegende Buspegel nicht mit dem von ihm gesendeten Pegel (`txd`-Signal zum Zeitpunkt t_5) übereinstimmt. Da v_1 ein rezessives Bit sendet, jedoch ein dominantes Bit empfängt, erkennt v_1, dass er die Arbitrierung verloren hat und stellt seine Übertragung ein. Dementsprechend gewinnt v_2 die Arbitrierung und kann seinen Rahmen (CAN-Identifier 5) innerhalb des *Mode-Based Time Window* übertragen.

Im weiteren Verlauf der Übertragung synchronisiert sich v_1 auf die Übertragung von v_2 durch mehrere Bit-Resynchronisationen. Dies ist notwendig, da die von v_2 gesendeten fallenden Flanken von v_1 als *Late Edges* empfangen werden. Die durchgeführte Phasenkorrektur ist an der schrittweisen Annäherung der Samplepoints beider Knoten (Markierungen t_6 und t_7 sowie t_8 und t_9) ersichtlich.

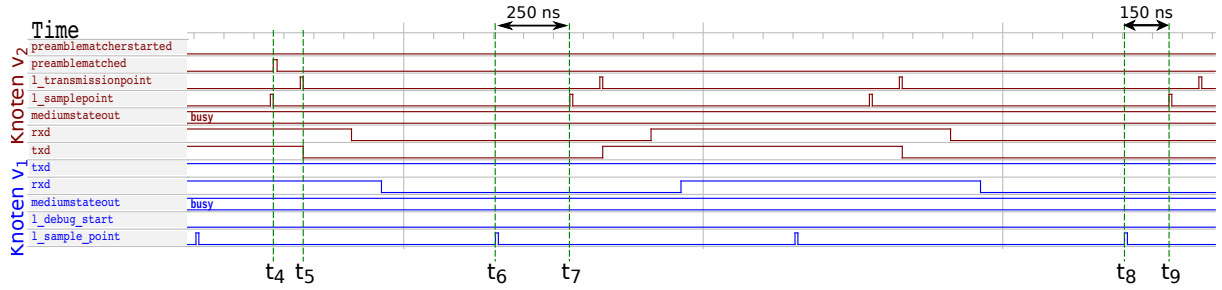


Abbildung 3.29: Erkennung der Transmission Start Preamble und Einklinken von v_2 in die Arbitrierung.

[74] Das Starten des Detektors nach dem Samplepoint erspart es, diesen wieder zu deaktivieren, falls die harte Synchronisation durch eine Störung ausgelöst wurde.

Das Listing 3.5 zeigt die relevanten Ausschnitte der Logdateien, welche die Arbeitsweise von BOT charakterisieren und den Ablauf dokumentieren.

Listing 3.5: Logausgabe von Knoten 1 und Knoten 2 bei Einsatz der BOT-Erweiterung.

```
1  Knoten v_1
   TTCANController@3164405ns: TTCANScheduler: Enqueuing Frame with ID7 in slot 9
3  TTCANController@3166505ns: TTCANScheduler: Slot9
   CANStreamProcessor_TransmitterBOT268:56:@3181315ns: BFT: Bits monitoring failed ->
     Arbitration lost or transmission error?
5  TTCANController@3181325ns: F_R: Result of Transmission approach: arbitrationfailed for
     FRAME ID 7
   CANStreamProcessor_FrameDecoder@3215575ns: FD: Frame successfully received with ID 5 at
     (local SOF time)316750 current local time 321337
7  TTCANController@3215585ns: F_R: Valid Frame received with ID 5

9
   Knoten v_2
11 TTCANController@3166505ns: TTCANScheduler: Enqueuing Frame with ID5 in slot 9
   BitTimingLogic@3170305ns: SOF: Start of Frame detected at 316940 at end of sync segment
     (endtime) 316925
13 CANStreamProcessor_TransmitterBOT@3180565ns: BFT: Preamble detected
   CANStreamProcessor_TransmitterBOT@3180575ns: BFT: Valid preamble found -> appending to
     already running transmission
15 CANStreamProcessor_TransmitterBOT@3180585ns: BFT: Preparing first bit after Transmission
     Start Window for transmission
   CANStreamProcessor_TransmitterBOT@3216565ns: BFT: Transmission of frame successfull
17 TTCANController@3216575ns: F_R: Result of Transmission approach: success for FRAME ID 5
```

Wie das Beispiel zeigt, stellt der Einsatz von BOT auch bei einem größeren Tickoffset[75] der Knoten, durch ein Einklinken in eine bereits laufende Arbitrierung, eine korrekte Funktionsweise des *Fast Mode-Signaling* sicher. Voraussetzung hierfür ist die korrekte Konfiguration der TP sowie die Wahl eines hinreichend großen TSW. Durch die geeignete Wahl des TSW können auf diese Weise auch Tickoffsets, die deutlich größer als eine Bitzeit sind, ausgeglichen werden, wie in diesem Beispiel demonstriert.

3.4.5 Bewertung von BOT

Wie unsere Evaluation zeigt, ermöglicht BOT auch bei größeren Tickoffsets die Sicherstellung der korrekten Funktionsweise des *Fast Mode-Signaling* mittels der CAN-Arbitrierung. Über die Länge des TSW kann in der Konfiguration gesteuert werden wie groß der maximal zulässige Tickoffset zwischen den Knoten sein darf, der durch BOT kompensiert wird. Die Festlegung der TP stellt darüber hinaus sicher, dass BOT nicht fälschlicherweise verfrühte Übertragungen aufgrund von Störungen innerhalb des TSW startet. Ohne TP würde eine einfache Umsetzung von BOT eine Übertragung starten, sobald eine fallende Flanke innerhalb des TSW erkannt wird. Dies könnte jedoch dazu führen, dass auch kurze Störimpulse eine Übertragung auslösen. Die Forderung nach einer TP mit einer Länge größer eins verhindert dies effektiv, da zunächst mindestens ein Bit (das SOF-Bit) vollständig empfangen werden muss, bevor BOT eine eigene Übertragung veranlasst. Zusätzlich erlaubt die konfigurierbare Länge der TP sehr feingranular festzulegen, wann und unter welchen Bedingungen der BOT-Mechanismus überhaupt eine Übertragung startet, sodass hierüber zusätzlich die Robustheit erhöht wird.

In Bezug auf die Nutzung der BOT-Erweiterung ergeben sich keine zusätzlichen Aufwände, weder für die Konfiguration noch zur Laufzeit. Die für eine sinnvolle Dimensionierung des TSW notwendige Abschätzung des maximal auftretenden Tickoffsets, ist ohnehin bei der Konfiguration der Länge der *Mode-*

[75] Ein Tickoffset, welcher die obere Schranke für den maximalen Tickoffset für *Fast Mode-Signaling* gemäß Constraint 3.11 verletzt.

Based Time Windows erforderlich[76] (vgl. Kapitel3.2.2). Durch BOT entfallen jedoch die Beschränkungen der maximalen Länge des Basiszyklus, da die korrekte Funktionalität des *Fast Mode-Signaling* von der gewählten Länge des TSW und nicht von dem maximalen Tickoffset abhängt (Constraint 3.5).

Die Realisierung von BOT innerhalb der TTCAN-Controller ist sehr einfach möglich und steht nicht im Konflikt mit den Konzepten des CAN- bzw. TTCAN-Protokolls. Vielmehr handelt es sich um eine funktionale Erweiterung des CAN-Controllers. Auch ergeben sich keine Einschränkungen hinsichtlich der gleichzeitigen Nutzung von TTCAN-Controllern mit BOT und Standard CAN-Controllern, da CAN-Nachrichten, welche unter Verwendung mittels BOT versendet wurden, weiterhin fehlerfrei von Standard CAN-Controllern empfangen und ausgewertet werden können. Zusammenfassend ergeben sich durch den Einsatz von BOT nur Vorteile und keine Nachteile, weder leidet die Robustheit noch die Kompatibilität oder die nutzbare Bandbreite.

3.5 Beurteilung der Robustheit von Mode-Based Scheduling with Fast Mode-Signaling für TTCAN

Durch die Integration der BOT-Erweiterung in TTCAN-Controller wird eine zuverlässige und robuste Umsetzung von *Fast Mode-Signaling* ermöglicht. Hierdurch wird *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* genauso robust, wie das native TTCAN-Protokoll und kann somit auch in sicherheitskritischen Anwendungen eingesetzt werden. Über die Länge des TSW und der TP lässt sich die Robustheit des *Fast Mode-Signaling* darüber hinaus nahezu beliebig skalieren.

Auch die Mechanismen zur Fehlerbehandlung von CAN (Fehlersignalisierung sowie Fehlerzähler) funktionieren ohne Einschränkung in Verbindung mit *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN*. Über das Bit-Monitoring sowie die primäre und sekundäre Fehlersignalisierung werden zuverlässig Übertragungsfehler erkannt und signalisiert. Gleichzeitig sorgen diese beiden Mechanismen dafür, dass Knoten die ständig fehlerhafte Rahmen senden oder solche fälschlicherweise signalisieren von der Kommunikation oder der Fehlersignalisierung ausgeschlossen werden.

Die obligatorische Fehlersignalisierung des CAN-Standards hat in Bezug auf TTCAN und *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* jedoch auch Nachteile. So erhöht sich der Overhead, da die Time Windows so dimensioniert werden müssen, dass trotz maximaler Dauer einer sich überlappenden Fehlersignalisierung (zum spätestmöglichen Zeitpunkt) die Integrität der benachbarten Time Windows gewahrt wird. Dies gilt nicht nur für die Dimensionierung von *Mode-Based Time Windows*, sondern auch für Exclusive und Arbitration Time Windows.

Ein weiteres Fehlerbild, welches nur für zeitgetriggerte Protokolle relevant ist, betrifft die Verletzung des Ablaufplans durch einen fehlerhaften Knoten, indem dieser außerhalb der ihm zugeordneten Slots sendet oder die Grenzen der ihm zugewiesenen Slots (z. B. aufgrund einer fehlerhaften Synchronisation) verletzt [Tem98]. Handelt es sich hierbei um korrupte Rahmen (und der fehlerhafte Knoten interpretiert und empfängt Fehlerrahmen korrekt), so sorgen die Mechanismen zur Fehlerbehandlung von CAN, auch bei TTCAN dafür, dass der Störer von der Kommunikation ausgeschlossen wird. Sind die durch den fehlerhaften Knoten gesendeten Rahmen jedoch korrekt (Rahmenformat, Bit-Stuffing und CRC-Checksumme), verfügt TTCAN über keine Konzepte, um hierauf zu reagieren und den Knoten über sein fehlerhaftes Verhalten zu informieren oder von der weiteren Kommunikation auszuschließen.

Eine Möglichkeit solchen Fehlern bei zeitgetriggerten Protokollen zu begegnen, besteht im Einsatz von Bus Guardians. Diese kennen den zulässigen Ablaufplan eines Knotens, überwachen dessen Buszugriffe und unterbinden nicht zulässige Medienzugriffe physikalisch [Tem98]. Hierdurch erhöhen sich jedoch die Kosten[77] für eine Busanbindung, da der Bus Guardian in der Regel als eigenständiger Chip realisiert werden muss und zusätzlich für jeden Knoten benötigt wird. Des Weiteren benötigt der Bus Guardian

[76] Dies gilt nicht nur für *Mode-Based Time Windows*, sondern auch für *Exclusive Time Windows* und *Arbitrating Time Windows*.
[77] Da sich der Bus Guardian mit dem Bus synchronisieren muss, um seine Aufgabe erfüllen zu können, muss dieser das jeweilige

Informationen über den Ablaufplan, um unzulässige Zugriffe auf den Bus unterbinden zu können, wodurch sich der Konfigurationsaufwand erhöht. Allerdings existieren zur Zeit weder Bus Guardians für TTCAN noch sind diese im Standard beschrieben oder vorgesehen (vgl. [NSL08, Kapitel 4.2.2.2] und [ISO04]). Ansonsten könnten etwaige Bus Guardians auch zusammen mit *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* genutzt werden, indem man die *Mode-Based Time Windows* wie Arbitrating Time Windows behandelt oder die Bus Guardians, wie die TTCAN-Controller, funktional erweitert. Allerdings sind die Bus Guardians auf lokale Überprüfungen beschränkt, d.h., den korrekten Ablauf und Ausgang des *Fast Mode-Signaling* können diese aufgrund der fehlenden Informationen darüber, welche Knoten jeweils eine Übertragung innerhalb eines konkreten *Mode-Based Time Window* planen (Frame-Assignment), nicht überwachen bzw. gewährleisten. Stattdessen ließe sich lediglich die Einhaltung des statischen Slot-Assignment prüfen.

3.6 Ausblick und Perspektiven

Obwohl CAN bereits 1986 entwickelt wurde, besitzt es immer noch eine vorherrschende Position als Kommunikationsprotokoll im Automobilbereich, aber auch in der Automatisierung. So liegt die Anzahl der CAN-Schnittstellen im Jahr 2012 bei mehr als 4 Milliarden und der Trend der standardmäßigen Integration von CAN-Controllern in Mikrocontroller (z. B. ARM Mikrocontroller von NXP [NXP13], Texas-Instruments [Tex14] oder STMicroelectronics [STM11]) wird auch in Zukunft für ein stetiges Wachstum sorgen [Eis12]. Durch diese Integration sinken die Kosten für eine Busanbindung, was der Verwendung von CAN wiederum zuträglich ist.

Einer der Gründe für den Erfolg von CAN ist neben der Robustheit durch den integrierten Schutz vor zerstörenden Kollisionen die Einfachheit und Klarheit des Protokolls in Bezug auf dessen Nutzung sowie die Möglichkeit der Priorisierung von Nachrichten. Mit dem Aufkommen von zeitgetriggerten Protokollen, speziell für den Einsatz in verteilten echtzeitfähigen Systemen (vgl. [Kop97]), wurde auch eine entsprechende Erweiterung für CAN diskutiert [JTN05] und in Form von TTCAN realisiert.

Um den wachsenden Bandbreitenbedarf der Industrie zu decken [NP12, Eis12], wird einerseits versucht die Bandbreite des CAN-Busses durch verbessertes Scheduling sowie Frame-Packaging effizienter zu nutzen, andererseits schreiten die Arbeiten am *CAN with Flexible Data-Rate Standard (CAN FD)* voran [ZWT09, Har12, Rob12]. Zum aktuellen Zeitpunkt gibt es bereits erste Hersteller von ARM-Mikrocontrollern, die CAN FD-Controller in ihre Produkte integrieren[78]. CAN FD hebt die Beschränkung der Übertragungsrate von CAN auf 1 MBit/s auf und erhöht gleichzeitig die maximal mögliche Nutzdatenmenge pro Rahmen. Dieser Schritt ist notwendig, um sich auf Dauer gegenüber Konkurrenten, wie FlexRay (10 MBit/s) oder Realtime Ethernet, behaupten zu können.

Hierzu führt CAN FD ein neues Rahmenformat ein. Dieses erhöht die maximale Nutzlast von 8 Bytes auf maximal 64 Bytes und verwendet einen hieran angepassten CRC-Mechanismus. Zusätzlich erlaubt CAN FD die Nutzdaten innerhalb des Rahmens mit einer höheren Übertragungsrate zu senden. Die Beschränkung der maximalen Übertragungsrate von CAN resultiert aus dem kollisionsgeschützten Arbitrierungsmechanismus, welcher den prioritätsbasierten Medienzugriff realisiert. Für dessen Umsetzung ist es notwendig, dass das *Propagation Time Segment* mindestens doppelt so groß gewählt wird wie die zulässige maximale Signalverzögerung zwischen zwei Knoten innerhalb des CAN-Busses (vgl. Constraint 3.1). Nur so ist gewährleistet, dass sich bei einer Übertragung mehrerer Stationen die einzelnen Bits (z. B. des Arbitrierungsfeldes) korrekt überlagern, ohne dass es zu Fehlern kommt. Da die Signallaufzeit über das physikalische Medium ebenso einen Beitrag zur Signalverzögerung liefert (Propagation-Delay), limitiert

Busprotokoll nahezu vollständig implementieren. Infolgedessen liegen die Kosten für einen Bus Guardian mindestens auf dem Niveau des jeweiligen Controllers für die Busanbindung.

[78] http://www.can-newsletter.org/hardware/semiconductors/141015_first-samples-with-can-fd-support_smart_atmel/

die Übertragungsrate auch die maximale Länge des CAN-Busses – je höher die Übertragungsrate, desto kürzer ist die maximal zulässige Buslänge.

Zielsetzung bei CAN FD ist die Beibehaltung des zentralen Konzepts der Arbitrierung sowie eine Minimierung der erforderlichen Anpassungen von bestehender Hard- und Software bei gleichzeitiger Erhöhung der Übertragungsrate. Deshalb behält CAN FD sowohl die Hostschnittstelle (weitestgehend) unverändert bei als auch den Physical Layer von CAN, sodass sich die Anpassungen auf den CAN-Controller selbst beschränken. Da jeder CAN FD Controller gleichzeitig auch das CAN-Protokoll beherrscht, ist eine schrittweise Einführung von CAN FD Knoten in ein bestehendes Bussystem mit CAN-Knoten möglich.

Die Erhöhung der durchschnittlichen Übertragungsrate erreicht CAN FD durch den Einsatz zweier verschiedener Übertragungsraten. Für die Felder des CAN-Rahmens, bei denen mehrere Stationen gleichzeitig auf das Medium zugreifen und daher eine korrekte Überlagerung der gesendeten Bits notwendig ist (Arbitrierungsfeld und Acknowledgement-Feld), verwendet CAN FD weiterhin eine gemäß dem CAN-Standard zugelassene Übertragungsrate von 125 kBit/s - 1 MBit/s. Zu Beginn des Datenfeldes schalten die CAN-Knoten auf eine zweite, schnellere Übertragungsrate um, welche nur für die Übertragung der Nutzdaten verwendet wird. Dies ist möglich, da zu diesem Zeitpunkt nur noch ein Knoten sendet. Am Ende des CRC-Feldes (mit dem Empfang des CRC-Delimiter) wird wieder auf die langsamere Übertragungsrate umgeschaltet. Für die Bestätigung kommt der Standard Acknowledge-Mechanismus von CAN zum Einsatz. Das Umschalten auf die langsamere Übertragungsrate ist notwendig, da wie bei der Arbitrierung, mehrere Knoten gleichzeitig als Sender fungieren und nur bei diesen Geschwindigkeiten – ohne die Buslänge weiter zu beschränken – eine korrekte Überlagerung der gesendeten Bits gewährleistet ist.

Im Automotive Umfeld wurden mit (Standard) CAN-Transceivern in der Daten-Phase Übertragungsraten von 5 MBit/s erreicht. Limitiert wurde die Übertragungsrate hierbei im Wesentlichen durch die minimal benötigte Impulsdauer der eingesetzten CAN-Transceiver. Mit geeigneten CAN-Transceivern (NXP TJA1040) konnten im Labor aber Übertragungsraten von 15 MBit/s in der Datenphase erreicht werden [Har12].

Wie anhand der Beschreibung ersichtlich, betreffen die notwendigen Anpassungen beim Übergang von CAN zu CAN FD primär die Funktionsweise des verwendeten CAN-Controllers. Die grundlegenden zentralen Konzepte, wie z. B. die kollisionsgeschütze Arbitrierung anhand von Identifiern, sind identisch. Da TTCAN das CAN-Protokoll unverändert nutzt und lediglich auf den CAN-Arbitrierungsmechanismus angewiesen ist (für die Verwendung innerhalb der Arbitrating Time Windows), kann TTCAN auch auf CAN FD aufsetzen. Hierfür ist weder eine Anpassung des CAN FD noch des TTCAN-Standards notwendig[79].

Mode-Based Scheduling with Fast Mode-Signaling für TTCAN wiederum setzt lediglich auf einem auf konzeptueller Ebene erweiterten TTCAN-Protokoll auf. Die Anforderungen an die MAC-Schicht sind die gleichen wie bei TTCAN und beschränken sich im Wesentlichen auf die Bereitstellung eines kollisionsgeschützten, prioritätsbasierten Arbitirierungsmechanismus. Da CAN FD diese Anforderungen erfüllt, lässt sich auch *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* mit CAN FD umsetzen und profitiert gleichermaßen von der höheren Übertragungsrate sowie der größeren Nutzdatenmenge pro Rahmen. Da der Arbitrierungsmechanismus von CAN FD mit dem von CAN identisch ist, ist auch die BOT-Erweiterung unverändert mit CAN FD anwendbar. Somit profitiert *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* von der Entwicklung von CAN FD und ist damit in Bezug auf diese Entwicklung zukunftssicher nutzbar und bietet gegenüber CAN FD die gleichen Vorteile wie gegenüber klassischem CAN.

[79] Hierbei nehmen wir an, dass das Erfassen des Empfangszeitpunktes der SOF-Bits, welches von TTCAN für die Synchronisation benötigt wird, entsprechend im CAN FD-Controller implementiert ist.

4 Technologie-unabhängige Realisierung von Mode-Based Scheduling with Fast Mode-Signaling

Wie in dem vorangegangenen Kapitel dargestellt, ist eine Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* auf Basis des TTCAN-Protokolls in nahezu optimaler Art und Weise möglich. Durch die Abbildung von *Fast Mode-Signaling* auf den CAN-Arbitrierungsmechanismus entsteht kein (zusätzlicher) Overhead, welcher die nutzbare Bandbreite reduziert, und durch die BOT-Erweiterung wird ein sehr hoher Grad an Zuverlässigkeit und Robustheit erreicht.

Die für das aktive *Fast Mode-Signaling* dort verwendete kollisionsgeschützte Arbitrierung von CAN setzt eine spezielle Kodierung der übertragenen Bits durch dominante und rezessive Signalpegel voraus und erfordert eine korrekte Überlagerung der gesendeten Bits. Hieraus ergeben sich physikalisch begründete Anforderungen an die minimal zulässige Bitzeit, welche die maximale Übertragungsrate beschränken (vgl. Constraint 3.1, Kapitel 3.1.1). Auch CAN FD ist hiervon betroffen, kompensiert dies aber zum Teil durch die Verwendung unterschiedlicher Übertragungsraten innerhalb eines Rahmens (vgl. Kapitel 3.6).

Deshalb haben wir eine alternative Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* für eine breite Palette unterschiedlichster – sowohl drahtgebundenen wie auch drahtlosen – Kommunikationstechnologien entworfen, welche nicht auf die besonderen Eigenschaften von CAN angewiesen ist. Auch hier steht wieder die Umsetzung von *Fast Mode-Signaling* im Fokus, da es sich um die Schlüsseltechnologie für den bandbreiteneffizienten und robusten Einsatz von *Mode-Based Scheduling* in sicherheitskritischen Anwendungen handelt. Um ein möglichst großes Spektrum von Technologien abzudecken, setzen wir weder eine besondere Kodierung für kollisionsgeschützte Übertragungen voraus noch dass ein Knoten während der eigenen Übertragung dazu in der Lage ist, gleichzeitig auf dem Medium zu lauschen[1]. Hierzu realisieren wir *Fast Mode-Signaling*, indem wir die *Modus-Präferenzen* auf unterschiedlich lange Verzögerungen vor dem Start der eigentlichen Übertragung abbilden. Je höher die zugeordnete *Modus-Präferenz* (Priorität) ist, desto kürzer ist die Verzögerung. Umgesetzt werden die unterschiedlich langen Verzögerungen durch die Verwendung von Backoffslots. Diese Form des passiven *Fast Mode-Signaling* lässt sich aufgrund der geringen Anforderungen leicht in existierende TDMA-basierte Kommunikationsprotokolle integrieren, ohne umfangreiche konzeptuelle Modifikationen an diesen vornehmen zu müssen (vgl. Kapitel 5 und 6).

Kapitel 4.1 fasst den aktuellen Stand der Technik zusammen und beleuchtet hierbei auch Kommunikationstechnologien, welche bereits Verzögerungen für die Priorisierungen einsetzen. Anschließend beschreibt Kapitel 4.2 detailliert unseren Lösungsansatz für die Realisierung von *Fast Mode-Signaling* mittels Backoffslots. Nach einer allgemeinen Betrachtung der Möglichkeiten zur Abbildung von Modes bzw. *Modus-Präferenzen* auf Backoffslots untersuchen wir Optimierungsstrategien zur Reduktion des resultierenden Overheads (Kapitel 4.3). Auf diesen Vorüberlegungen aufbauend, definieren wir eine geeignete zeitliche Struktur für Mikro- und Backoffslots sowie Constraints für deren Konfiguration, welche eine zuverlässige deterministische Funktionsweise von *Fast Mode-Signaling* gewährleisten (Kapitel 4.4).

[1] Eine Einschränkung, die insbesondere für den Bereich der drahtlosen Kommunikation zutrifft.

4.1 Stand der Technik

Der Einsatz von Verzögerungen bei der Übertragung von Rahmen zur Abbildung von Prioritäten kommt in verschiedenen Kommunikationsprotokollen in unterschiedlichen Ausprägungen und Realisierungen zum Einsatz. Beispiele lassen sich sowohl im drahtlosen Bereich als auch bei drahtgebundenen Feldbussen finden. Wir beschränken uns an dieser Stelle auf die Betrachtung von IEEE 802.11 und Byteflight, als wichtige Vertreter dieser beiden Kategorien. Die Kapitel 5.6 und 6.1 liefern Abgrenzungen zu weiteren drahtlosen Ansätzen im Bereich der Sensornetze (WirelessHart, ISA 100.11a) sowie FlexRay, dem designierten Nachfolger von Byteflight. Weitere Strategien, um Prioritäten zu repräsentieren, existieren in Form von *Busy tone*-Protokollen und *Binary Countdown*-Protokollen, auf die wir im Folgenden auch kurz eingehen werden.

4.1.1 IEEE 802.11

Der IEEE 802.11 Standard [IEE99] wurde 1997 verabschiedet und spezifiziert ein technisches Verfahren für den Aufbau und Betrieb lokaler Funknetzwerke unter Verwendung des 2.4 GHz Bandes[2]. Seit 1997 wurde der IEEE 802.11 Standard kontinuierlich weiterentwickelt, um höhere Übertragungsraten und eine bessere Unterstützung für Quality-of-Service (QoS) zu ermöglichen (IEEE 802.11e [IEE07]).

Bei IEEE 802.11 greifen alle Knoten wahlfrei auf ein gemeinsames Medium in Form eines Funkkanals zu. Um Kollisionen zu vermeiden, nutzt IEEE 802.11 nicht-persistentes CSMA/CA (*Carrier sense multiple access with collision avoidance*) als Medienzugriffsverfahren. Hierbei hört ein Knoten zunächst das Medium ab und startet seine eigene Übertragung erst, wenn dieses eine bestimmte Zeitspanne frei ist.

IEEE 802.11 spezifiziert zwei unterschiedliche Betriebsarten: *Distributed Coordination Function (DCF)* sowie *Point Coordination Function (PCF)*. DCF regelt den wahlfreien konkurrierenden Zugriff auf das Medium ohne zentralen Koordinator. Bevor ein Knoten, der DCF verwendet, sendet, muss das Medium zunächst für die feste Zeitspanne DIFS (DCF Interframe Space) frei sein. Um die Wahrscheinlichkeit von Kollisionen zu reduzieren, wird diese Zeitspanne noch um einen zufälligen Random Backoff verlängert. Erst, wenn während des gesamten Zeitraums das Medium frei ist, beginnt der Knoten (unverzüglich) mit der Übertragung seines Rahmens. Der Random Backoff bestimmt sich aus dem Wert eines Backoffzählers, der zufällig aus einem Intervall von Werten (Contention Window) gezogen und mit der festen Dauer des Backoffslots $d_{\text{backoffSlot}}$ multipliziert wird.

Im Gegensatz hierzu übernimmt bei der *PCF* der sogenannte *Point Coordinator (PC)* die Kontrolle über den Mediumzugriff. Hierzu sendet der PC periodisch ein Beacon Frame, welches unter anderem Management Informationen enthält, wie zum Beispiel den Zeitpunkt des nächsten Beacon Frames sowie die Aufforderung an neue Stationen, sich anzumelden. Die erfolgreiche Übertragung des Beacon Frame leitet dann eine Contention Free Period ein, die nicht unterbrochen wird und in der der PC die anderen Knoten pollt. Der PC nutzt für den Medienzugriff ebenfalls CSMA/CA. Bevor der PC seinen Beacon Frame sendet, muss das Medium zunächst für die Zeitspanne PIFS (PCF Interframe Space) frei sein.

IEEE 802.11 erlaubt den simultanen Betrieb von PCF und DCF in einer Funkzelle. Die Abbildung 4.1 zeigt ein entsprechendes Beispiel. Der Vorrang des PC vor den Knoten, die DCF verwenden, wird durch unterschiedliche Längen der Interframe Spaces (IFS) realisiert. Es gilt, $d_{\text{SIFS}} < d_{\text{PIFS}} < d_{\text{DIFS}}$, wobei d_X die Dauer des IFS X bezeichnet. Die Dauer d_{SIFS} – des *Short Interframe Space* – wird als Pause zwischen den einzelnen Rahmen einer zusammenhängenden Rahmensequenz verwendet. Dadurch, dass das SIFS das kürzeste IFS ist, kann kein anderer Knoten eine Rahmensequenz durch eine eigene Übertragung[3] unterbrechen. Die Zeitspanne SIFS kommt ebenfalls als IFS zwischen den Rahmen der Contention Free

[2] Der IEEE 802.11 Standard definiert sowohl Physical- als auch MAC-Layer. Neben dem 2.4 GHz Band spezifiziert IEEE 802.11a zusätzlich auch eine physikalische Schicht für das 5 GHz Band.

[3] Wir beschränken uns an dieser Stelle auf die vereinfachte Annahme, dass sich alle Knoten in Kommunikationsreichweite zueinander befinden (Single-Hop-Netzwerk).

Period zum Einsatz. So ist sichergestellt, dass die Contention Free Period nicht durch Übertragungen von Knoten, die DCF verwenden, unterbrochen wird.

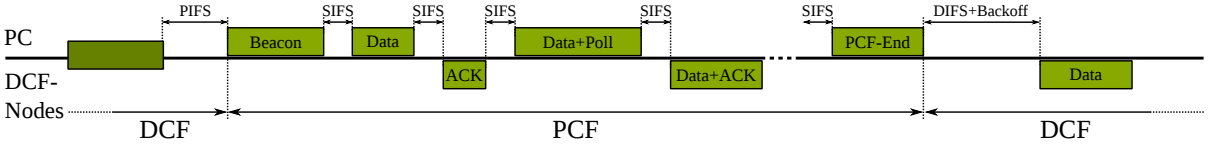


Abbildung 4.1: Beispiel für den simultanen Betrieb von PCF und DCF in einer Zelle [TH08].

IEEE 802.11e verbessert die Unterstützung von QoS, durch die Einführung von 8 *Traffic Categories (TCs)* bei der *Enhanced DCF (EDCF)* Betriebsart. Hierbei wird jeder TC ein AIFS (Arbitration Interframe Space) zugeordnet. Die Priorisierung der TC erfolgt wiederum über die Länge der ihnen zugeordneten AIFS und somit letztendlich über die Länge der Wartezeit. So ist AIFS für die TC *Voice* zur Übertragung von Sprachdaten kleiner als der AIFS der TC *Best-Effort*. Um Kollisionen zu vermeiden, wird beim Mediumzugriff, wie bei DCF, das AIFS um einen Random-Backoff ergänzt. Die Kompatibilität zu IEEE 802.11 wird gewahrt, indem $d_{\mathrm{DIFS}} \leq d_{\mathrm{AIFS}_j}$ gewählt wird, wobei d_{AIFS_j} die Dauer des AIFS der TC j bezeichnet [TH08]. Damit Übertragungen einer hohen TC dennoch gegenüber den Übertragungen mittels DCF bevorzugt werden, setzt IEEE 802.11e die oberen Grenzen für das Contention Window dynamisch in Abhängigkeit von der Priorität der Daten[4].

Auch mit der Unterstützung verschiedener TCs bietet IEEE 802.11 keine zu *Mode-Based Scheduling* vergleichbaren Konzepte für die effiziente Realisierung sporadischer Nachrichten. Der wettbewerbsbasierte Zugriff bei DCF ermöglicht keine deterministischen Garantien. Dies gilt ebenso für die mit EDCF eingeführten TCs, da verschiedenen Knoten gleichzeitig für unterschiedliche Datenströme die gleichen TCs verwenden können. Die einzige Möglichkeit, deterministische Garantien zu realisieren, bietet die PCF[5], hierbei obliegt es jedoch dem PC, die Knoten zu pollen, sodass auch hierüber keine effiziente Abbildung der Anforderungen sporadischer Nachrichten (im Vergleich zu *Mode-Based Scheduling with Fast Mode-Signaling*) möglich ist.

4.1.2 Byteflight

Byteflight [BPG00, CV04, TTT02, GBP00] ist ein zwischen 1996 und 2000 durch die BMW AG zusammen mit Motorola, der Siemens AG und ELMOS AG entwickelter, zeitgetriggerter Feldbus. Byteflight unterstützt eine Übertragungsrate von 10 MBit/s und wurde speziell für die Anforderungen sicherheitskritischer Anwendungen im Automobil entworfen. Bis zum Jahr 2007 kam Byteflight innerhalb der Serienproduktion von BMW in den Modellreihen 5, 6 und 7 für die Ansteuerung der Airbags zum Einsatz [Hei12]. Um die Robustheit gegenüber elektromagnetischen Störungen sicherzustellen, nutzt Byteflight optische Übertragungsverfahren auf Basis von Lichtwellenleitern (Plastic optical fiber – POF). Byteflight verwendet eine Sterntopologie mit einem aktiven Sternkoppler. Neben dem Weiterleiten von Signalen, überwacht dieser die Kommunikation und trennt fehlerhafte Knoten (z. B. *babbling idiots*) vom Bus.

Byteflight unterteilt die Zeit in sich wiederholende Zyklen mit einer festen Länge von 250 µs. Jeder Zyklus beginnt mit einem Synchronisationsimpuls (*synchronization pulse – sync*), welcher durch einen speziellen Knoten – den *Sync Master* – versendet wird und der Synchronisation der Knoten dient. Der *Sync*

[4] Damit erhöht sich die Wahrscheinlichkeit, dass Daten mit einer hohen Priorität schnell übertragen werden [TH08].

[5] Unter der Annahme, dass die Übertragung des Beacons des PC nicht verzögert wird, z. B. weil kein paralleler DCF-Betrieb erfolgt.

Master kann die Form des Synchronisationsimpulses variieren, um einen Alarmzustand zu signalisieren und dadurch z.B. bestimmte Sicherheitsfunktionen freischalten[6].

Der Medienzugriff wird bei Byteflight mittels FTDMA (*Flexible Time Division Multiple Access*) realisiert. Hierbei wird jedem Nachrichtentyp ein eindeutiger Identifier (8 Bit) zugeordnet. Jede Nachricht kann eine Nutzlast von bis zu 12 Bytes transportieren, die über eine 15-Bit CRC abgesichert wird. Im Fall einer fehlerhaften Übertragung wird auf eine Fehlersignalisierung oder erneute Übertragung auf der Protokollebene von Byteflight verzichtet. Um Kollisionen zu vermeiden, dürfen die Nachrichten eines Typs nur von einem Knoten versendet werden. Die Nachrichten werden in aufsteigender nummerischer Reihenfolge ihrer Identifier über das Medium übertragen. Somit entspricht ein niedrigerer Identifier einer höheren Nachrichtenpriorität. Dies wird realisiert, indem alle Knoten einen eigenen Slotzähler verwalten, der mit jedem verstrichenen Minislot um eins inkrementiert wird. Ein Minislot ist bei Byteflight definiert als eine feste Zeitspanne $T_{wx\Delta}$, in der der Bus nicht belegt ist[7]. Sobald der Slotzähler eines Knotens identisch mit dem Identifier einer ihm vorliegenden Nachricht ist, beginnt dieser mit deren Übertragung. Während einer laufenden Nachrichtenübertragung werden die Slotzähler nicht inkrementiert. Vor dem Start stellen die Knoten sicher, dass die Übertragung noch innerhalb der Grenzen des Zyklus abschlossen werden kann, ansonsten wird darauf verzichtet. Der Slotzähler wird beim Empfang des Synchronisationsimpulses jeweils wieder auf 0 zurückgesetzt.

Abbildung 4.2 illustriert den Ablauf bei der Übertragung von Rahmen innerhalb eines Byteflight-Zyklus. Die beiden Zeitspannen[8] T^{rx}_{wx0}, T^{tx}_{wx0} dienen dazu, die unterschiedlichen Signalverzögerungen bei der Übertragung eines Rahmens zwischen den Knoten zu kompensieren. Hierzu wenden die Knoten nach jeder Übertragung eine der beiden Verzögerungen – T^{rx}_{wx0} als Empfänger und T^{tx}_{wx0} als Sender – an, bevor diese regulär mit der Aktualisierung ihrer Slotzähler fortfahren. Hierdurch soll erreicht werden, dass alle Knoten ihre Slotzähler möglichst synchron inkrementieren (vgl. Abbildung 4.2 sowie [CV04]). In der Abbildung ist der Zyklus aus Sicht des *Sync Master* dargestellt, welcher den Synchronisationsimpuls und den Rahmen mit Identifier 1 (grün) sendet und den Rahmen mit dem Identifier 3 (orange) empfängt.

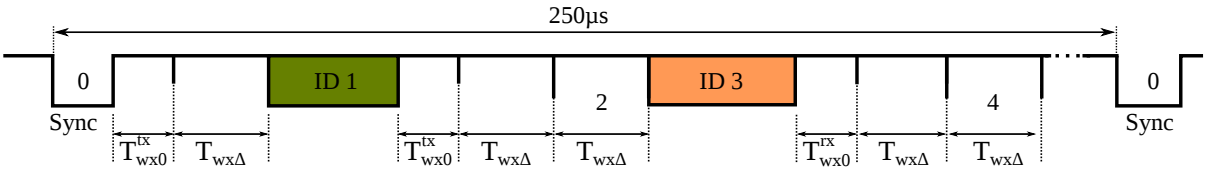


Abbildung 4.2: Timing und Nachrichtenübertragung innerhalb eines Byteflight Zyklus [CV04].

Die Darstellung der Zeiträume in Abbildung 4.2 ist nicht maßstabsgetreu. In der Praxis ist die Länge eines Minislots deutlich kürzer als die Übertragungsdauer eines Rahmens. Auf diese Weise wird der Overhead minimiert, wenn beispielsweise in einem Zyklus keine Nachricht eines bestimmten Identifiers übertragen wird (siehe z. B. die ungenutzten Minislots der Identifier 2 und 4). Denn anders als bei CAN reduziert jeder nicht genutzte, aber zugewiesene Identifier die für die Übertragung zur Verfügung stehende Bandbreite. Mittels FTDMA realisiert Byteflight eine dynamische Vergabe der verfügbaren Bandbreite auf Basis von Prioritäten. Diese Priorisierung wird durch Wartezeiten realisiert.

Byteflight sieht vor, dass den wichtigen periodischen Nachrichten die niedrigsten Identifier zugeordnet werden. So wird sichergestellt, dass diese innerhalb des aktuellen Zyklus übertragen werden können. Die höheren Identifier werden an sporadische Nachrichten vergeben, sodass diese sich die noch verbleibende

[6] Dies bewirkt jedoch keine Änderung des Zyklus oder der zugeteilten Identifier [GBP00].

[7] Die Länge des Minislots kann konfiguriert werden. Die minimale Länge wird jedoch durch die maximale Signallaufzeit innerhalb des Byteflight-Busses beschränkt [BPG00].

[8] Hierbei handelt es sich um einen knotenspezifischen Konfigurationsparameter, der (hauptsächlich) von der Signallaufzeit zwischen dem jeweiligen Knoten und dem Sternkoppler geprägt ist.

Bandbreite des Zyklus teilen. Ob für Nachrichten mit höheren Identifiern eine Übertragung in dem jeweiligen Zyklus noch garantiert ist, hängt von der konkreten Ausgestaltung des Ablaufplans ab. Die Abbildung 4.3 illustriert dies anhand eines Beispiels basierend auf [CV04].

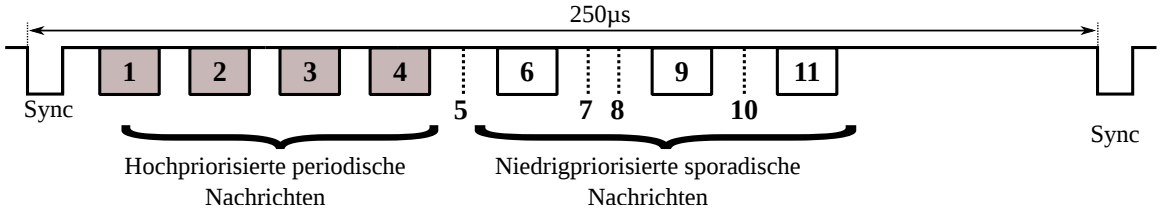


Abbildung 4.3: Mögliche Zuordnung von Identifier bei Byteflight [CV04].

Im Gegensatz zu *Mode-Based Scheduling* konkurrieren bei Byteflight alle Knoten um die verfügbare Bandbreite innerhalb des Zyklus. Dies hat in Kombination mit FTDMA zur Folge, dass die Übertragungszeitpunkte nicht exakt festgelegt sind, sondern davon abhängen, ob und wie viele Rahmen mit einem niedrigeren Identifier in dem jeweiligen Zyklus gesendet werden. Im Allgemeinen führt ein größerer Identifier, in diesem Zusammenhang, zu einem größeren maximalen Jitter des zugeordneten Rahmens. Dies vermeidet *Mode-Based Scheduling* durch einen eingeschränkten Wettbewerb auf Ebene einzelner Slots. Hierdurch werden deterministische Garantien für jede Nachricht mit der höchsten Priorität innerhalb des jeweiligen Slots ermöglicht und gleichzeitig sichergestellt, dass die Übertragungen zu exakt definierten Zeitpunkten erfolgen (vgl. hierzu auch Kapitel 4.2).

4.1.3 Busy tone- und Binary-Countdown-Protokolle

Busy tone-Protokolle übertragen vor dem Senden des eigentlichen Rahmens zunächst ein Jamming-Signal, oft als Black Burst (BB) bezeichnet[9], nachdem das Medium zuvor als frei erkannt wurde [SK96]. Die Mediumarbitrierung wird realisiert, indem die Knoten Black Bursts unterschiedlicher Länge übertragen. Sobald ein Knoten die Übertragung seines BB abgeschlossen hat, prüft er, ob das Medium frei ist. Falls das Medium noch belegt ist, überträgt ein anderer Knoten gerade einen längeren BB und verfügt daher über eine höhere Priorität. Ist das Medium hingegen frei, hat der Knoten die Arbitrierung gewonnen und kann seinen Rahmen senden. Beispiele für entsprechende Protokolle sind z. B. *DB-DCF* [PDO02] oder *Real-Time Chain* [BPC+07].

Mit Hilfe der Arbitrierung der *Busy tone*-Protokolle lässt sich *Mode-Based Scheduling with Fast Mode-Signaling* durch die Abbildung der *Modus-Präferenzen* auf unterschiedlich lange BBs realisieren[10]. Eine höhere *Modus-Präferenz* entspricht dann einem längeren BB. Somit wird auch hier die Priorisierung über unterschiedlich lange Wartezeiten realisiert, nur dass in diesem Fall eine längere Wartezeit eine höhere Priorität repräsentiert. *Busy tone*-Protokolle benötigen die aktive Belegung des Mediums zur Signalisierung der Prioritäten, da *Busy tone*-Protokolle häufig im Kontext von anderen Protokollen, wie z. B. IEEE 802.11, eingesetzt werden, bei denen die Kommunikation (im Gegensatz zu *Mode-Based Scheduling*) ereignisgetriggert erfolgt. Durch das Senden der BBs während der Arbitrierung wird sichergestellt, dass in dieser Zeit kein anderer Knoten eine Übertragung startet. Bei *Mode-Based Scheduling with Fast Mode-Signaling* ist eine Signalisierung der Medienbelegung während des *Fast Mode-Signaling* aufgrund der zeitgetriggerten Kommunikation (Unterteilung der Zeit in Slots, geplante Übertragungszeitpunkte innerhalb der Slots) nicht notwendig, wie unsere Realisierung des *Fast Mode-*

[9] Hierbei handelt es sich um ein dominantes Signal, welches es erlaubt, ein Medium als belegt zu erkennen. Außer ihrer Übertragungsdauer enthalten BB keine weiteren Informationen.

[10] Da die Knoten aktiv kommunizieren, um den Mode mit der höchsten Priorität zu ermitteln, handelt es sich um eine Form des aktiven *Fast Mode-Signaling*.

Signaling in diesem Kapitel zeigt. Daher untersuchen wir die Variante des aktiven *Fast Mode-Signaling* auf Basis der Arbitrierung der *Busy tone*-Protokolle im Folgenden nicht näher.

Binary-Countdown-Protokolle [PAT07] übertragen die Konzepte von CAN, insbesondere die CAN-Arbitrierung, auf drahtlose Netzwerke. Die Prioritäten (von Knoten oder Rahmen) werden durch eindeutige Bitsequenzen (Identifier) repräsentiert, und eine Arbitrierung stellt sicher, dass sich der Knoten (oder Rahmen) mit der höchsten Priorität durchsetzt. Hierzu wird der Identifier bitweise übertragen und dabei die einzelnen Bits mit dominanten und rezessiven Signalen codiert. Sendet ein Knoten ein rezessives Signal, empfängt jedoch ein dominantes, so scheidet dieser aus der Arbitrierung aus. Somit gewinnt der Knoten oder Rahmen den Wettbewerb um das Medium, dessen Identifier den höchsten nummerischen Wert aufweist. Vertreter dieser Protokolle sind z. B. *CSMA/IC* [YYH03], *WiDom* [PAT07] oder ACTP [CGR12]. Kapitel 5.4 skizziert eine mögliche Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* mittels ACTP und diskutiert Vor- und Nachteile.

4.2 Realisierung von Fast Mode-Signaling durch Übertragungsverzögerungen

Wie die ausgewählten Beispiele illustrieren, gibt es zahlreiche Protokolle[11], die Verzögerungen für die Signalisierung von Prioritäten nutzen, allerdings unterscheiden diese sich deutlich von *Mode-Based Scheduling with Fast Mode-Signaling* in ihren Merkmalen und Charakteristika. Keines dieser Protokolle bietet eine Unterstützung für einen beschränkten kontrollierten Wettbewerb, wie von *Mode-Based Scheduling with Fast Mode-Signaling* beschreiben, oder vergleichbare Konzepte für die Umsetzung sporadischer Nachrichten mit hohen Echtzeitanforderungen (vgl. Kapitel 2).

Daher wollen wir in diesem Kapitel zunächst die notwendigen Konzepte (allgemein und unabhängig von einer konkreten Technologie oder einem existierenden Protokoll) entwerfen, die mindestens benötigt werden, um *Mode-Based Scheduling with Fast Mode-Signaling* zu realisieren. Hierfür betrachten wir eine Realisierung von *Fast Mode-Signaling* mittels Verzögerungen (Backoffslots), da diese Variante der Umsetzung kaum Anforderungen an die verwendete Basistechnologie stellt. Daher bildet diese Form des passiven *Fast Mode-Signaling* zusammen mit den weiteren hier vorgestellten Konzepten (Backoffslots, Mikroslots) eine gute Grundlage für die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in existierende TDMA-basierte Kommunikationsprotokolle (vgl. Kapitel 5 und 6).

Wir untersuchen zunächst im Detail die Realisierung von *Fast Mode-Signaling* mittels Verzögerungen (Backoffslots) sowie mögliche Varianten Modes (bzw. *Modus-Präferenzen*) auf Backoffslots abzubilden. Dann beschäftigen wir uns mit dem internen Aufbau der Backoffslots sowie den von *Mode-Based Scheduling* verwendeten Mikroslots zur Unterteilung des Mediums.

Wir verwenden im Folgenden wieder das in Kapitel 2.3 eingeführte Kommunikationsmodell. Das heißt, wir betrachten ein Single-Hop-Netzwerk, bestehend aus den paarweise verbundenen Knoten $V = \{v_1, \ldots, v_s\}$ (Definition 2.2). Des Weiteren sei $M = \{m_1, \ldots, m_r\}$ die nicht leere, endliche Menge der Modes (Definition 2.3). Die Zuordnung der *Modus-Präferenzen* zu den einzelnen Modes erfolgt, bezogen auf einen Mikroslot, anhand der partiellen Funktion $mp : S \times M \to \mathbb{N}_0$ (Definition 2.5). Die Slotzuordnung wird über die partielle Slot-Assignment Funktion $SA : S \times M \to V$ definiert; die Rahmenzuordnung basiert auf der partiellen Funktion *FA* (Definitionen 2.4 und 2.6).

Grundannahme ist ein zeitgetriggertes, auf TDMA basierendes, Kommunikationsmedium. Die Zeit ist unterteilt in eine unendliche Sequenz von (fortlaufend nummerierten) Zeitintervallen identischer Länge $\Theta = \{S_1, S_2, \ldots\}$, die als Makroslots bezeichnet werden. Jeder Makroslot ist wiederum unterteilt in eine endliche Menge $S = \{s_1, \ldots, s_n\}$ fortlaufend nummerierter Mikroslots. Deren Länge ist zur Laufzeit fest,

[11] Eine Abgrenzung zwischen *Mode-Based Scheduling with Fast Mode-Signaling* und Protokollen aus dem Bereich drahtloser Sensornetzwerke (WirelessHart, ISA 100.11a) sowie FlexRay, die eine Unterteilung in Zeitslots verwenden, erfolgt in den Kapiteln 5.6 und 6.1.

kann aber für jeden Mikroslot individuell konfiguriert werden (vgl. Definition 2.1). Zusätzlich zu den bereits definierten Makro- und Mikroslots führen wir Backoffslots ein.

Definition 4.1

Ein Backoffslot bezeichnet ein Zeitintervall der Länge d_{back}, welches lang genug ist, damit die Knoten eines Netzwerkes auch bei maximaler Synchronisationsungenauigkeit und Signalverzögerung die Belegung des Mediums sicher erkennen können, falls ein Knoten innerhalb des Backoffslots seine Übertragung startet.

Wie wir im Folgenden sehen werden, ist es sinnvoll, den Backoffslot so klein wie möglich zu wählen, da dieser den Overhead bei der vorgestellten Realisierung von *Fast Mode-Signaling* prägt. Die minimale Länge eines Backoffslots ist von der verwendeten Technologie, dem Medium, der Topologie und der maximalen Synchronisationsgenauigkeit abhängig. Ebenfalls berücksichtigt werden muss der Zeitpunkt (BTSP), zu dem ein Knoten innerhalb des Backoffslots seine Übertragung startet. Diesen Fragestellungen sowie dem internen Aufbau der Backoffslots widmet sich das Kapitel 4.4.2.

Unsere Realisierung von *Fast Mode-Signaling* verwendet Übertragungsverzögerungen zur Priorisierung und nutzt für die Abbildung der Prioritäten die Backoffslots. Hierzu interpretieren wir die *Modus-Präferenzen* zunächst als Anzahl von Backoffslots, die ein Medium (nach Beginn des Mikroslots) frei sein muss, bevor ein Rahmen des Modes mit der jeweiligen *Modus-Präferenz* gesendet werden darf. Wurde vor dem Erreichen des zugeordneten Backoffslots bereits mit der Übertragung eines anderen Rahmens (mit einer höheren *Modus-Präferenz*) begonnen, wird der Rahmen zurückgestellt.

Definition 4.2

Sei $v \in V$ ein Knoten des Netzwerkes und $s \in S$ ein modusbasierter Mikroslot, für den der Knoten v für den Mode m eine Slotzuordnung besitzt, d.h. $SA(s,m) = v$. Des Weiteren sei $f \in F$ ein entsprechender Rahmen des Modes m, welcher durch den Knoten v in dem Mikroslot s übertragen werden soll, d.h. es gilt $FA(S,s,m,v) = f$. Die Backoffslots eines modusbasierten Mikroslots werden fortlaufend nummeriert; der erste Backoffslot eines modusbasierten Mikroslots trägt die Nummer 0. Dann beginnt der Knoten v die Übertragung des Rahmens f in dem Backoffslot mit der Nummer $mp(s,m)$ des Mikroslots s, sofern in keinem früheren Backoffslot dieses Mikroslots bereits eine Übertragung gestartet wurde.

Die Umsetzung des beschriebenen Verhaltens lässt sich über einfache Zähler realisieren. Beim Start eines Mikroslots, welcher für die modusbasierte Kommunikation vorgesehen ist – kurz modusbasierter Mikroslot –, setzen zunächst alle Knoten ihren Backoffslot-Counter auf 0. Bleibt das Medium für einen Backoffslot unbelegt, erhöhen alle Knoten synchron[12] ihren Backoffslot-Counter um eins. Bei jeder Modifikation des Backoffslot-Counters prüfen die Knoten, ob für den aktuellen Mikroslot ein Slot- und Frame-Assignment existiert, für dessen Rahmen die *Modus-Präferenz* identisch mit dem Wert des Backoffslot-Counter ist. Existiert ein solcher Rahmen und wurde in keinem vorherigen Backoffslot dieses Mikroslots bereits eine Übertragung gestartet, so beginnt der Knoten mit der Übertragung dieses Rahmens in dem aktuellen Backoffslot. Der Backoffslot-Counter wird solange erhöht, bis die maximale Anzahl von Backoffslots für den modusbasierten Mikroslot erreicht oder eine Übertragung gestartet wurde.

Die in Definition 4.2 beschriebene Abbildung der *Modus-Präferenzen* auf die Wartezeiten ist verträglich mit der durch die *Modus-Präferenzen* definierten Ordnungsrelation. Das beschriebene Medienzugriffsverfahren basiert auf einem TDMA-Verfahren und wendet innerhalb der modusbasierten Mikroslots eine Variation des FTDMA-Verfahrens an. Die vorgestellte Abbildung der *Modus-Präferenz* auf Backoffslots

[12] Abweichungen ergeben sich lediglich durch Abweichungen der lokalen Uhren der Knoten zueinander.

stellt dabei sicher, dass es zu keinen Kollisionen in den Mikroslots kommt[13] und der Mikroslot dem (eingeplanten) Rahmen mit der höchsten *Modus-Präferenz* exklusiv zur Verfügung steht.

Betrachten wir zunächst ein einfaches Beispiel: Gegeben sei ein Makroslot, bestehend aus 5 Mikroslots. Der erste Mikroslots ist exklusiv für die Synchronisation reserviert. Insgesamt besteht unser Netzwerk aus den Knoten $V = \{v_1, v_2, v_3, v_4, v_5\}$ und den Modes $M = \{Emergency, Safety, Regular, Stream\}$. Die entsprechenden *Modus-Präferenzen* bzw. Slotzuordnungen dieses Szenarios sind in den Tabellen 4.1a bzw. 4.1b dargestellt.

SA	s_1	s_2	s_3	s_4
Emergency	v_3		v_3	
Safety	v_2	v_3	v_4	v_2
Regular		v_4		v_5
Stream		v_1	v_2	v_1

(a) *Slot-Assignment*-Funktion

MP	s_1	s_2	s_3	s_4
Emergency	0		0	
Safety	1	1	1	1
Regular		2		2
Stream		3	4	3

(b) *Modus-Präferenz*-Funktion

Tabelle 4.1: Darstellung der Slotzuordnungen und *Modus-Präferenzen*.

Die Tabelle 4.2 zeigt die für den Makroslot S_5 eingeplanten Rahmen[14] (*Frame-Assignment*). Grau hinterlegt sind jeweils die Rahmen mit der höchsten *Modus-Präferenz*, die in dem jeweiligen Mikroslot übertragen werden (müssen).

FA		$s_{5,1}$	$s_{5,2}$	$s_{5,3}$	$s_{5,4}$
Emergency,	v_3	–		$f_{3,2}$	
Safety,	v_2	$f_{2,1}$			–
Safety,	v_3		$f_{3,1}$		
Safety,	v_4			$f_{4,1}$	
Regular,	v_4		–		
Regular,	v_5				–
Stream,	v_1		$f_{1,1}$		$f_{1,2}$
Stream,	v_2			–	

Tabelle 4.2: *Frame-Assignment*-Funktion für den Makroslot S_5.

Die Abbildung 4.4 zeigt den Makroslot S_5 für das *Frame-Assignment* aus Tabelle 4.2 sowie dessen Mikro- und Backoffslots. Ebenso dargestellt ist die Zuordnung der Backoffslots zu den jeweiligen Modes und Knoten gemäß Definition 4.2. Die gestrichelten Backoffslots dienen nur zur Verdeutlichung des Prinzips (zu diesem Zeitpunkt wurden die Backoffslot-Counter der Knoten bereits deaktiviert, da schon eine Übertragung läuft). Wie anhand der Abbildung direkt ersichtlich, beeinflusst die Länge des Backoffslots unmittelbar den Overhead der vorgestellten Realisierung von *Fast Mode-Signaling*. Je kleiner die Backoffslots gewählt werden (können), desto geringer fällt dieser aus. Die Abbildung 4.4 illustriert lediglich die konzeptionelle Funktionsweise von *Fast Mode-Signaling* mittels Backoffslots und zeigt nicht den exakten Zeitpunkt, zu dem die Übertragungen innerhalb der Backoffslots gestartet werden (vgl. Kapitel 4.4.2).

[13] Voraussetzung hierfür ist, dass die Vorgaben für *Mode-Based Scheduling* eingehalten werden (vgl. Kapitel 2). Das heißt, ein Mode darf maximal einem Knoten pro Mikroslot zugeordnet werden, und alle Modes eines Mikroslot müssen unterschiedliche *Modus-Präferenzen* aufweisen.

[14] Ein Strich als Eintrag gibt an, dass kein Rahmen für den Versand bereitsteht. Jeder eingeplante Rahmen ist in der Tabelle mit einem Eintrag $f_{i,j}$ dargestellt. Hierbei identifiziert der Index i den Knoten v_i als Sender; der Index j nummeriert alle eingeplanten Rahmen des Knotens fortlaufend.

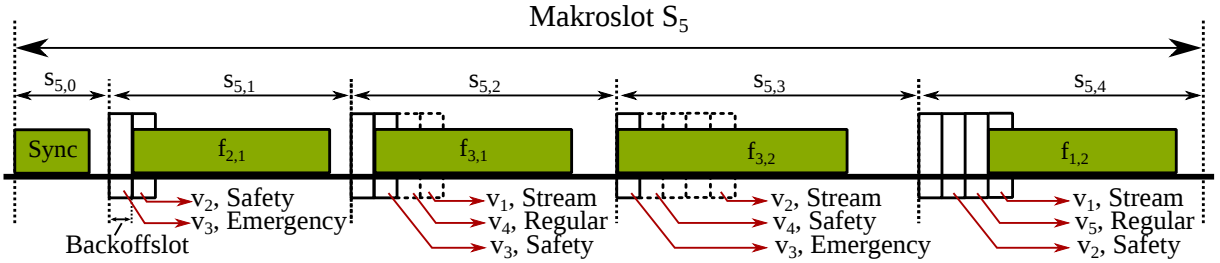


Abbildung 4.4: Darstellung der Medienbelegung in Makroslot S_5.

Der vorgestellte Lösungsansatz ist am ehesten mit Byteflight zu vergleichen, da dieses im Gegensatz zu 802.11e, hinsichtlich der Prioritäten, nicht auf eine bestimmte Anzahl von Traffic-Klassen beschränkt ist und durch die eindeutige Zuordnung von Prioritäten Kollisionen verhindert, sodass (mit gewissen Einschränkungen) deterministische Garantien möglich sind.

Im Gegensatz zu Byteflight, welches nur für die ersten *n* Identifier innerhalb eines Zyklus eine Übertragung garantieren kann, ermöglicht die vorgestellte Lösung deterministische Garantien für alle Nachrichten mit den größten *Modus-Präferenzen* innerhalb der jeweiligen Mikroslots. Der beschränkte Wettbewerb sowie die individuelle Zuordnung der Prioritäten pro Mikroslot erlaubt zudem eine feingranularere und flexiblere Abbildung der Kommunikationsanforderungen (vgl. Kapitel 2.4.1). Ein Nachteil der globalen Anwendung von FTDMA als Medienzugriffsverfahren besteht darin, dass der Zeitpunkt der Übertragung einer Nachricht nicht exakt festgelegt und abhängig von früheren Übertragungen sowie deren Länge ist. Die Folge ist ein zunehmender maximaler Jitter, je niedriger die Priorität der Nachricht ist. Durch die hier vorgestellte Unterteilung des Makroslots in Mikroslots sowie die feste Zuordnung von Backoffslots erfolgen die Übertragungen von Nachrichten dagegen zu exakt festgelegten Zeitpunkten[15] und unabhängig von Existenz sowie Länge vorheriger Übertragungen.

Ein weiterer Vorteil gegenüber einem globalen FTDMA-basierten Zugriffsverfahren betrifft nachträgliche Anpassungen und Änderungen an bestehenden Ablaufplänen. Bei Byteflight kann das Hinzufügen neuer Nachrichtentypen dazu führen, dass die Identifier vieler Nachrichtentypen angepasst werden müssen, wenn es sich um eine wichtige Nachricht mit hoher Priorität handelt. Bei *Mode-Based Scheduling with Fast Mode-Signaling* ist die Zuordnung von *Modus-Präferenz* auf den jeweiligen Mikroslot begrenzt, Anpassungen sind daher mit deutlich geringerem und tendenziell eher lokal begrenztem Aufwand (d.h. auf den Mikroslot begrenzt), verbunden.

Nachteilig an dieser Form der Umsetzung von *Fast Mode-Signaling* (im Vergleich zu der Realisierung mit CAN) ist der Umstand, dass nicht genutzte *Modus-Präferenzen* (z. B. weil kein Rahmen für diese Mikroslotinstanz eingeplant ist) Bandbreite belegen. So ist im Mikroslot $s_{5,1}$ kein Rahmen im Mode *emergency* eingeplant und somit bleibt der erste Backoffslot ungenutzt (vgl. Abbildung 4.4). Hierbei handelt es sich jedoch um ein konzeptuelles Problem, welches in dieser Form der Umsetzung nicht lösbar ist. Dementsprechend wichtig ist es, die Länge des Backoffslots zu minimieren, um diesen Overhead so gering wie möglich zu halten.

4.3 Optimale Abbildung von Modus-Präferenzen auf Backoffslots

Neben der Länge des Backoffslots gilt es auch, die Anzahl der (nicht zugeordneten) Backoffslots pro Mikroslot zu minimieren. So ist eine direkte Nutzung der *Modus-Präferenzen* zur Festlegung der Anzahl von Backoffslots, um die der Übertragungsstart verzögert werden soll, nicht notwendigerweise optimal,

[15] Ob ein eingeplanter Rahmen innerhalb eines Mikroslots gesendet wird, ist abhängig davon, ob ein Frame-Assignment für den gleichen Mikroslot mit einer höheren *Modus-Präferenz* existiert.

da dies dazu führen kann, dass ein Backoffslot mit keinem Mode assoziiert wird. Gründe hierfür können z. B. nicht fortlaufende Werte der zugeordneten *Modus-Präferenzen* eines Mikroslots sein, oder weil dem Mode mit der höchsten *Modus-Präferenz* in einem Mikroslot nicht die *Modus-Präferenz* 0 zugeteilt wurde. Diesen Freiheitsgrad bietet *Mode-Based Scheduling*, da es sich um ein Modellierungskonzept handelt. Bei der hier gezeigten Verwendung der *Modus-Präferenz* für das *Fast Mode-Signaling* führt dies jedoch (unter Umständen) zu unnötigem Overhead. In den Mikroslots $s_{5,2}$ und $s_{5,4}$ folgt aus der Anwendung der Definition 4.2 aufgrund der Wahl der *Modus-Präferenzen*, dass kein Mode mit dem ersten Backoffslot assoziiert wird. Daher bleibt dieser stets ungenutzt und belegt unnötig Bandbreite. Die nicht fortlaufend gewählten Werte der *Modus-Präferenzen* führen in Mikroslot $s_{5,3}$ dazu, dass dem dritten und vierten Backoffslot kein Mode zugeordnet wird (vgl. Tabelle 4.1b, Abbildung 4.4) und diese daher stets ungenutzt bleiben.

Um diesen Problemen zu begegnen, gibt es zwei Ansätze. Der offensichtliche Weg ist die Anpassung der Modus-Präferenz-Funktion *mp*, sodass die zugeordneten *Modus-Präferenzen* pro Mikroslot fortlaufend vergeben sind, beginnend mit dem Wert 0 für die höchste *Modus-Präferenz* eines Mikroslots. Wir präferieren jedoch die zweite Variante, die wir auch bei der Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* angewendet haben: Zusätzlich zu der *Modus-Präferenz*-Funktion *mp* wird eine *verträgliche* Abbildung definiert, welche die Zuordnung der Modes auf die Backoffslots in optimaler Weise durchführt. Hierdurch kann die Anzahl der Backoffslots, die pro Mikroslot benötigt werden, auf ein Minimum reduziert werden. Dies wirkt sich wiederum positiv auf die Länge der Mikroslots und somit auch auf die Bandbreitennutzung aus.

4.3.1 Verwendung von verträglichen Abbildungen

Die Gründe für dieses Vorgehen liegen in der Sichtweise der Modes bzw. *Modus-Präferenzen*. Diese stellen ein abstraktes technologieunabhängiges Modellierungskonzept und Planungsverfahren dar. Eine Vermischung mit technologiespezifischen Realisierungsaspekten sollte deshalb vermieden werden. Durch die Trennung lässt sich sicherstellen, dass eine Planung und Problemlösung zunächst (möglichst) unabhängig von der konkreten Implementierung erfolgt und nicht an eine spezifische Kommunikationstechnologie gebunden ist. Erst im letzten Schritt kommt dann die Adaption an die konkrete Kommunikationstechnologie/Realisierung zum Tragen – in diesem Fall die Abbildung der Modes (bzw. *Modus-Präferenzen*) auf Backoffslots. In gewisser Weise entspricht dieses Vorgehen den Prinzipien bei der modellgetriebenen Entwicklung [BBG05, Obj03]. Auch dort erfolgt zunächst der Entwurf einer technologieunabhängigen Lösung eines Problems in Form eines plattformunabhängigen Modells (PIM), welches dann in ein plattformspezifisches Modell (PSM) transformiert wird (unter Einbeziehung von Wissen über die Eigenschaften der spezifischen Zielplattform).

In Bezug auf unser abstraktes Kommunikationsmodell ist eine *verträgliche Abbildung* (vgl. Kapitel 3.2.1) wie folgt definiert:

Definition 4.3

Sei $M = \{m_1, \ldots, m_r\}$ die Menge der Modes und $S = \{s_1, \ldots, s_n\}$ die Menge der Mikroslots. Des Weiteren sei $mp : S \times M \to \mathbb{N}_0$ eine Modus-Präferenz-Funktion. Dann heißt eine Funktion $map : S \times M \to \mathbb{N}_0$ verträglich mit mp, genau dann, wenn

$$\forall s \in S, m_1 \in M, m_2 \in M. (mp(s,m_1) < mp(s,m_2) \Rightarrow map(s,m_1) < map(s,m_2)).$$

Um deutlich zu machen, dass eine Abbildung map verträglich mit einer gegebenen Modus-Präferenz-Funktion mp ist, schreiben wir kurz map_{mp}.

Für eine gemäß Definition 4.3 mit der *Modus-Präferenz*-Funktion verträgliche Abbildung zwischen Modes und Backoffslots ist gewährleistet, dass die durch die *Modus-Präferenzen* eingeführte Präferenzordnung korrekt durch das *Fast Mode-Signaling* mittels Backoffslots (vgl. Definition 4.4) umgesetzt wird. An dieser Stelle sei darauf hingewiesen, dass die *Modus-Präferenz*-Funktion *mp* zu sich selbst verträglich ist. Allerdings ergibt sich hierbei nicht notwendigerweise eine optimale Nutzung der Bandbreite, wie das Beispiel belegt. Anhand der beschriebenen Umsetzung des passiven *Fast Mode-Signaling* mittels Backoffslots lässt sich eine optimale verträgliche Abbildung der Modes (*Modus-Präferenzen*) auf Backoffslots, welche einen minimalen Overhead erzeugt, sehr einfach definieren[16].

Hierfür ergänzen wir zunächst unsere Definition 4.2 für das passive *Fast Mode-Signaling* mittels Backoffslots um den Aspekt der verträglichen Abbildungen.

Definition 4.4

Sei $v \in V$ ein Knoten des Netzwerkes und $s \in S$ ein modusbasierter Mikroslot, für den der Knoten v für den Mode m eine Slotzuordnung besitzt, d.h. $SA(s,m) = v$. Des Weiteren sei $f \in F$ ein Rahmen des Modes m, welcher durch den Knoten v in dem Mikroslot s übertragen werden soll, d.h. es gilt $FA(S,s,m,v) = f$. Neben der Modus-Präferenz-Funktion mp ist ebenfalls eine mit mp verträgliche Abbildung $map_{mp} : S \times M \to \mathbb{N}_0$ definiert, welche den einzelnen Modes den Backoffslot zuordnet, in dem die Übertragung des Rahmens gestartet wird. Die Backoffslots eines modusbasierten Mikroslots sind fortlaufend nummeriert; der erste Backoffslot eines modusbasierten Mikroslots trägt die Nummer 0. Dann beginnt der Knoten v die Übertragung des Rahmens f in dem Backoffslot mit der Nummer $map_{mp}(s,m)$, sofern in keinem früheren Backoffslot dieses Mikroslots bereits eine Übertragung gestartet wurde.

Die Definition 4.4 räumt die Freiheit ein, beliebige verträgliche Abbildungen für die Zuordnung zwischen Modes und Backoffslots zu wählen. So deckt die Definition auch weiterhin die direkte Verwendung der *Modus-Präferenz*-Funktion ab.

4.3.2 Definition einer optimalen verträglichen Abbildung

Um eine optimale Abbildung der Modes auf die Backoffslots zu erreichen, muss sichergestellt werden, dass allen Backoffslots eines Mikroslots – beginnend mit dem ersten – jeweils ein Mode zugewiesen wird. Auf diese Weise werden Backoffslots vermieden, in denen keine Übertragung starten kann und die daher unnötig Bandbreite reservieren. Eine solche optimale Abbildung der Modes auf Backoffslots ist in Definition 4.5 gegeben.

Definition 4.5

Sei V die Menge der Knoten eines Netzwerkes, S die Menge der Mikroslots und M die Menge der Modes. Des Weiteren sei $mp : S \times M \to \mathbb{N}_0$ eine Modus-Präferenz-Funktion und $SA : S \times M \to V$ eine Slot-Assignment-Funktion. Dann ordnet die partielle Funktion $slt_{mp} : S \times M \to \mathbb{N}_0$ jedem Mode $m \in M$ eines modusbasierten Mikroslots $s \in S$ einen Backoffslot zu, in dem die Übertragung eines Rahmens dieses Modes gestartet werden darf. Die partielle Funktion $slt_{mp} : S \times M \to \mathbb{N}_0$ ist wie folgt definiert:

$$\forall s \in S, \forall m \in M \text{ mit } SA(s,m) \text{ definiert} : slt_{mp}(s,m) \equiv |M_s \backslash M_{s,m}|$$

[16] Anwendungen können eine solche optimale Abbildung direkt aus der Slot-Assignment-Funktion und der *Modus-Präferenz*-Funktion gemäß der Definition 4.4 ableiten (vgl. Kapitel 4.3.2).

mit

$$M_s \equiv \{m \in M | SA(s,m) \text{ ist definiert}\}$$
$$M_{s,m} \equiv \{m' \in M_s | mp(s,m) \leq mp(s,m')\}$$

Die Definition 4.5 sorgt für eine fortlaufende, aufsteigende Zuordnung der Backoffslotnummern zu den Modes eines Mikroslots, unter Berücksichtigung der durch die *Modus-Präferenzen* definierten Präferenzordnung. Da für die Zuordnung der Backoffslotnummern eines Mikroslot nur die Modes berücksichtigt werden, für die ein Slot-Assignment besteht, ist jeder Backoffslot mit genau einem Mode assoziiert. Der Beweis, dass die von uns definierte Abbildung slt_{mp} mit der *Modus-Präferenz*-Funktion mp verträglich ist, folgt unmittelbar aus der Definition (vgl. Lemma B.1, Anhang B.1).

4.3.3 Anwendungsbeispiel

Wenden wir die Abbildung slt_{mp} auf unser Beispiel aus Kapitel 4.2 an, so ergibt sich die in Tabelle 4.3b dargestellte Zuordnung zwischen Modes und Backoffslots. Zum direkten Vergleich zeigt Tabelle 4.3 eine Gegenüberstellung der *Modus-Präferenzen* und der Zuordnung der Backoffslots. Die Unterschiede zwischen der optimierten Abbildung slt_{mp} im Vergleich zu der *Modus-Präferenz*-Funktion sind in der Tabelle 4.3b farblich hinterlegt.

MP	s_1	s_2	s_3	s_4
Emergency	0		0	
Safety	1	1	1	1
Regular		2		2
Stream		3	4	3

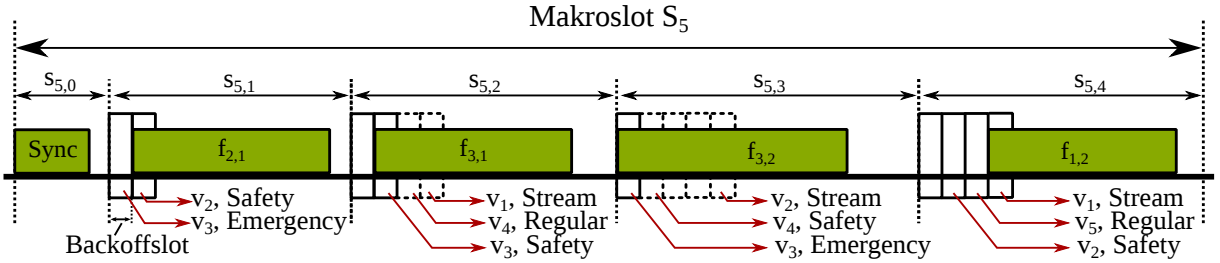
(a) *Modus-Präferenz*-Funktion

$\mathbf{slt}_{mp}$	s_1	s_2	s_3	s_4
Emergency	0		0	
Safety	1	0	1	0
Regular		1		1
Stream		2	2	2

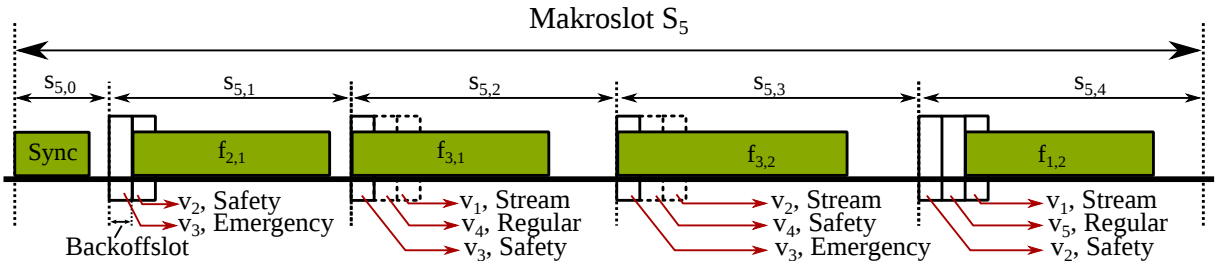
(b) slt_{mp}-Funktion

Tabelle 4.3: Gegenüberstellung der *Modus-Präferenz*-Funktion mit der optimalen Abbildung slt_{mp} aus Definition 4.5.

Die *Slot-Assignment*-Funktion bleibt ebenso wie die *Frame-Assignment*-Funktion unverändert (Tabellen 4.1a und 4.2), weswegen wir an dieser Stelle auf eine erneute Darstellung verzichten. Die Abbildung 4.5 illustriert die Auswirkung der unterschiedlichen Zuordnungen der Modes zu den Backoffslots anhand der eingeplanten Übertragungen in Makroslot S_5 (Tabelle 4.2). Hierfür zeigt Abbildung 4.5a die Situation bei der direkten Verwendung der *Modus-Präferenzen* als Backoffslotnummern, während in der Darstellung 4.5b die optimierte Abbildung slt_{mp} zum Einsatz kommt. Im direkten Vergleich führt dies zur Entfernung der nicht nutzbaren Backoffslots und somit eines Teils des Overheads für die Umsetzung des *Fast Mode-Signaling*, ohne das der Designer bei der Wahl der *Modus-Präferenzen* hierauf Rücksicht nehmen muss. Somit können die entsprechenden Mikroslots um die eingesparten Backoffslots und somit auch der gesamte Makroslot verkürzt werden (hierauf wurde in der Darstellung verzichtet).



(a) Medienbelegung bei der Nutzung der *Modus-Präferenz*-Funktion für die Zuordnung von Backoffslots.



(b) Medienbelegung bei der Nutzung der slt_{mp}-Funktion für die Zuordnung von Backoffslots.

Abbildung 4.5: Vergleich der Medienbelegung bei unterschiedlicher Zuordnung von Backoffslots zu den verschiedenen Modes.

4.4 Aufbau der Mikroslots und Backoffslots

In diesem Kapitel widmen wir uns dem internen Aufbau der Mikro- und Backoffslots. Dazu leiten wir anhand technischer Größen (Synchronisationsungenauigkeit, Signallaufzeiten etc.) Anforderungen an Aufbau und Dimensionierung der Mikro- und Backoffslots her. Diese aus den theoretischen Überlegungen resultierenden Constraints können bei der Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* mit einer konkreten Technologie eingesetzt werden, um die Mikro- und Backoffslots so zu dimensionieren, dass eine zuverlässige Funktionalität gewährleistet ist[17].

Eine wesentliche Einflussgröße ist hierbei die maximale Synchronisationsungenauigkeit zwischen den Knoten. Diese wird durch regelmäßige (Re-)Synchonisationen begrenzt, um die Abweichungen, welche aus dem Clock Skew der Knoten zueinander resultieren, zu kompensieren. Verantwortlich für den Clock Skew sind unter anderem Temperatureinflüsse, Vibrationen, aber auch Fertigungstoleranzen oder Alterung der Quarze [Joh92, WG92]. Doch auch bei einer regelmäßigen (Re-)Synchronisation ist es nicht möglich, eine perfekte Synchronität der Knoten zu erreichen. Dabei hängt die maximale Synchronisationsungenauigkeit sowohl von dem gewählten Synchronisationsverfahren als auch von den maximalen Signalverzögerungen bei der Übertragung sowie dem Resynchronisationsintervall und dem maximal zulässigen Clock Skew ab.

Diese maximale Synchronisationsungenauigkeit der Knoten zueinander gilt es bei der Konzipierung der Mikro- und Backoffslots zu berücksichtigen. Es muss sichergestellt sein, dass alle Knoten – auch bei einer maximalen Abweichung – den Übertragungsstart dem gleichen Mikroslot und bei modusbasierter Kommunikation auch dem gleichen Backoffslot zuordnen. Nur so ist die Kollisionsfreiheit sowie die korrekte Identifizierung des jeweiligen Modes – anhand des Backoffslots, in dem die Übertragung gestartet wurde – gewährleistet.

[17] Kapitel 5 nutzt die hier vorgestellten Konzepte und Constraints, um ein modusbasiertes Kommunikationsprotokoll auf Basis von IEEE 802.15.4 zu realisieren. Auch die vorgestellte Integration in FlexRay (Kapitel 6) baut auf den hier beschriebenen Konzepten auf.

Kapitel 4.4.1 gibt zunächst eine allgemeine Übersicht über die Einflussfaktoren sowie die grundlegenden Annahmen. Basierend hierauf erfolgt dann die Ableitung der Anforderungen für den Aufbau der Backoff- (Kapitel 4.4.2) und Mikroslots (Kapitel 4.4.3).

4.4.1 Grundlagen

Für die Realisierung eines TDMA-basierten Zugriffsverfahrens genügt die Verwendung einer Ticksynchronisation. Eine Uhrensynchronisation ist in diesem Anwendungskontext nicht notwendig. Die Ticksynchronisation (auch Heartbeat Synchronisation [BBJM07]) basiert auf der Erzeugung eines gemeinsamen Referenzzeitpunktes (Tick) bei den zu synchronisierenden Knoten. Die Uhrensynchronisation unterscheidet sich von der Ticksynchronisation lediglich dadurch, dass zusätzlich noch der mit dem Referenzzeitpunkt assoziierte Zeitstempel ausgetauscht wird. Bei der Ticksynchronisation erfolgt die Bestimmung von Zeitpunkten innerhalb der Makroslots (z. B. der Start von Mikro- und Backoffslots) in Bezug auf den ermittelten Referenzzeitpunkt unter Verwendung der lokalen Uhren der jeweiligen Knoten. Aufgrund des Clock Skew zwischen den Uhren der Knoten muss die Synchronisation regelmäßig wiederholt werden, um ein Auseinanderdriften der Uhren der Knoten zu verhindern. Für unsere weiteren Betrachtungen gehen wir daher von einer Ticksynchronisation aus, diese gelten jedoch entsprechend auch für eine Uhrensynchronisation.

Ausgangspunkt ist die Fragestellung, wie groß der maximale Tickoffset innerhalb eines Netzwerkes sein kann (Worst-Case Betrachtung). Dieser tritt unmittelbar vor der erneuten Synchronisation auf und setzt sich aus zwei Komponenten zusammen (vgl. [GK11]). Der maximale Baseoffset gibt die maximale Ungenauigkeit unmittelbar nach der (Re-)Synchronisation an, die sich aufgrund von Signalverzögerungen sowie den Eigenarten des gewählten Verfahrens einstellen kann. Hinzu kommen die aus der Clock Skew resultierenden Abweichungen zwischen den Resynchronisationen. Wir nehmen an, dass sich sowohl für den maximalen Baseoffset als auch den Clock Skew obere Grenzen bestimmen lassen. Dann ergibt sich für den maximalen Tickoffset:

Definition 4.6 (Maximaler Tickoffset)

Gegeben sei eine Menge von Knoten V eines Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren, und ein (Tick-)Synchronisationsprotokoll P mit dem periodischen Resynchronisationsintervall $d_{rsyncInt}$. Darüber hinaus sei $r_{maxClockSkew}$ der maximale paarweise Clock Skew der Knoten aus V. Dann ist der maximale Tickoffset $d^P_{maxTickoffset}$ unmittelbar vor einer erneuten Synchronisation durch das Protokoll P definiert als

$$d^P_{maxTickoffset} \equiv d^P_{maxBaseOffset} + 2 \cdot r_{maxClockSkew} \cdot d_{rsyncInt}.$$

Hierbei bezeichnet $d^P_{maxBaseOffset}$ den maximalen Baseoffset, d.h. den maximalen Offset aller Knoten unmittelbar nach der Resynchronisation.

Definition 4.6 ist als Verallgemeinerung von Definition 3.7 (Kapitel 3.2.2) zu verstehen. Basierend auf dieser allgemeinen Abschätzung des maximalen Tickoffsets betrachten wir im Folgenden den Aufbau der Mikro- und Backoffslots für unser TDMA-basiertes Zugriffsprotokoll mit Unterstützung für *Mode-Based Scheduling with Fast Mode-Signaling*.

Während wir uns im vorherigen Kapitel speziell mit CAN als Übertragungstechnologie beschäftigt haben, soll hier eine Lösung für *Mode-Based Scheduling with Fast Mode-Signaling* entwickeln, die nicht auf die Eigenschaften einer speziellen Technologie angewiesen ist. Aus diesem Grund wählen wir auch für die Definition der maximalen Signalverzögerung zwischen den Knoten eines Netzwerkes eine allgemeinere Formulierung als in Kapitel 3.3 und definieren diese als Funktion des Propagation-Delay sowie den spezifischen Verzögerungen für die Signalverarbeitung.

Definition 4.7 (Signalverzögerung)

Gegeben sei eine Menge von Knoten V eines Single-Hop-Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren. Wir definieren die maximale Signalverzögerung zwischen zwei Knoten als Funktion[18] $sig\_delay_{max} : V \times V \to \mathbb{R}$. *Für zwei Knoten* $v_1 \in V$ *(Sender) und* $v_2 \in V$ *(Empfänger) gilt:*

$$sig\_delay_{max}(v_1, v_2) \equiv d^{v_1,max}_{txProcDelay} + d^{v_1,v_2,max}_{PropagationDelay} + d^{v_2,max}_{rxProcDelay}$$

Hierbei bezeichnet

$d^{x,max}_{txProcDelay}$ *die maximale interne Verarbeitungsverzögerung des Sendeknotens* $x \in V$

$d^{x,y,max}_{PropagationDelay}$ *die maximale Signallaufzeit (Propagation-Delay) aufgrund der physikalischen Entfernung der Knoten* $x \in V$ *und* $y \in V$.

$d^{x,max}_{rxProcDelay}$ *die maximale interne Verarbeitungsverzögerung des Empfangsknotens* $x \in V$

Zur Vereinfachung definieren wir zusätzlich die maximale Signalverzögerung für Teilmengen von Knoten $V_1 \subseteq V$ *und* $V_2 \subseteq V$ *unseres Netzwerkes*

$$sig\_delay^{V_1 \to V_2}_{max} \equiv \max_{v_1 \in V_1, v_2 \in V_2, v_1 \neq v_2} sig\_delay_{max}(v_1, v_2)$$

sowie die folgende Kurzschreibweise in Bezug auf alle Knoten eines Netzwerkes

$$sig\_delay^{V}_{max} \equiv sig\_delay^{V \to V}_{max}.$$

Analog zu Definition 4.7 legen wir die minimale Signalverzögerung als Funktion $sig\_delay_{min} : V \times V \to \mathbb{R}$ sowie die Schreibweisen $sig\_delay^{V_1 \to V_2}_{min}$ und $sig\_delay^{V}_{min}$ fest.

4.4.2 Die interne Struktur der Backoffslots

Zentrale Anforderung für Aufbau und Dimensionierung der Backoffslots ist es sicherzustellen, dass auch bei maximalem Tickoffset alle Knoten den Übertragungsstart zuverlässig demselben Backoffslot zuordnen können. Ohne diese Anforderung ist die korrekte Funktionsweise von *Fast Mode-Signaling* sowie die Kollisionsfreiheit nicht gewährleistet.

Hierzu definieren wir zunächst – in Analogie zum TSP bei TTCAN – den Backoff Transmission Start Point (BTSP).

Definition 4.8 (Backoff Transmission Startpunkt – BTSP)

Der Backoff Transmission Startpunkt (BTSP) legt den Startzeitpunkt einer modusbasierten Rahmenübertragung innerhalb eines Backoffslots fest. Dieser Startzeitpunkt wird von dem jeweiligen Sender anhand seiner lokalen Uhr bestimmt. Die Zeitdauer d_{BTSP} *legt die Zeitspanne zwischen dem Beginn eines Backoffslots und dessen BTSP fest. Hierbei handelt es sich um einen globalen Konfigurationsparameter, der für alle Backoffslots eines Makroslots gilt*[19]*. Die Abbildung 4.6 zeigt einen entsprechend aufgebauten Backoffslot der Länge* d_{back}*, in welchem ein Knoten eine Übertragung startet.*

[18] Hierbei steht *sig_delay* als Kurzform für Signaldelay.

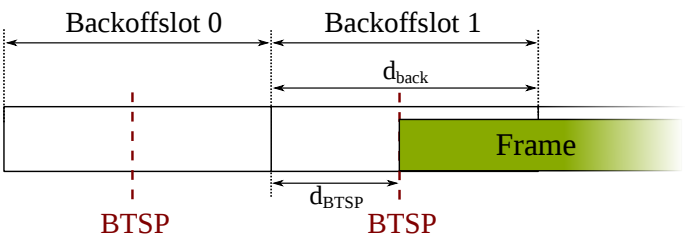


Abbildung 4.6: Aufbau eines Backoffslots.

Um sicherzustellen, dass alle Knoten den Übertragungsstart dem korrekten Backoffslot zuordnen, ist die minimale Länge von d_{BTSP} durch den maximalen Tickoffset begrenzt. Die Abbildung 4.7 illustriert dies anhand eines Senders (S) und eines Empfängers (E), die den maximalen Tickoffset zueinander aufweisen. Hierbei beträgt die Signalverzögerung zwischen Sender und Empfänger in diesem Beispiel $sig\_delay(S,E)$.

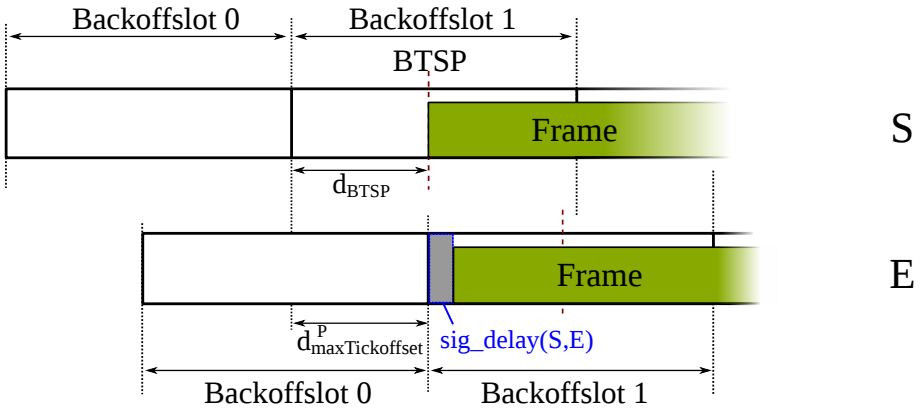


Abbildung 4.7: Rahmenübertragung mit maximalem Tickoffset zwischen Sender (S) und Empfänger (E).

Da wir unser Modell der Backoff- und Mikroslots möglichst unabhängig von einer konkreten Kommunikationstechnologie gestalten wollen, muss bei der Wahl des BTSP auch berücksichtigt werden, dass der jeweilige Knoten zu diesem Zeitpunkt auch technisch in der Lage sein muss, eine Übertragung zu starten. So müssen z. B. Transceiver für Funktechnologien in der Regel zunächst vom Empfangs- in den Sendemodus wechseln, bevor eine Übertragung möglich ist. Da diese während der Umschaltphase taub sind, kann ein Umschalten erst zum Beginn des Backoffslots erfolgen, in dem der Knoten seine eigene Übertragung startet, da ansonsten die Möglichkeit bestünde, einen verspäteten Übertragungsstart im vorherigen Slot (z. B. aufgrund des Tickoffsets, vgl. Abbildung 4.8) nicht zu detektieren. Diese Totzeit (sofern vorhanden) muss bei der Wahl des BTSP ebenfalls berücksichtigt werden. Constraint 4.1 fasst diese Anforderungen bzgl. der Konfiguration des BTSP zusammen.

[19] Die Verwendung eines einzigen globalen Konfigurationsparameters, ohne Berücksichtigung, welche Knoten als modusbasierte Sender innerhalb der Backoffslots agieren können, reduziert sowohl den Konfigurationsaufwand als auch potentielle Fehlerquellen (vgl. Constraint 4.1).

Constraint 4.1 (Konfiguration des BTSP)

Gegeben sei eine Menge von Knoten V eines Single-Hop-Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren und ein (Tick-)Synchronisationsprotokoll P. Des Weiteren sei $V_{MB} \subseteq V$ die Menge aller modusbasierten Knoten, d.h. aller Knoten mit mindestens einem Slot-Assignment in einem modusbasierten Mikroslot. Der maximale Tickoffset der Knoten V zueinander beträgt $d^P_{maxTickoffset}$. Zudem bezeichnet $d^v_{RX \to TX}$ die Totzeit des Transceivers des Knotens $v \in V$, die für das Umschalten vom Empfangs- in den Sendemodus anfällt. Dann muss für die Konfiguration des BTSP gelten, dass

$$d_{BTSP} \geq d^{MIN}_{BTSP}$$

mit

$$d^{MIN}_{BTSP} \equiv \max(d^P_{maxTickoffset}, \max_{v \in V_{MB}} d^v_{RX \to TX})$$

Auch bei der Nutzung von Kommunikationstechnologien bzw. Transceivern, die entweder Vollduplex-fähig oder beim Umschalten vom Empfangs- in den Sendemodus keine Totzeit aufweisen, kann Constraint 4.1 zur Bestimmung des BTSP herangezogen werden. Es genügt, $\max_{v \in V_{MB}} d^v_{RX \to TX} = 0$ zu setzen.

Bei der Festlegung der Länge eines Backoffslots gilt es zu gewährleisten, dass die Knoten eine Belegung des Mediums bis zum Ende des jeweiligen Backoffslots sicher detektieren können. Um eine Übertragung zu erkennen, kommen häufig die gleichen Mechanismen wie beim *Carrier Sense* zum Einsatz. Diese Mechanismen – im drahtlosen Bereich häufig mit CCA (Clear Channel Assessment) bezeichnet – benötigen jedoch eine bestimmte Zeitspanne[20], um die Belegung eines Mediums zu erkennen und ggf. an den Knoten zu melden[21]. Weitere Einflussfaktoren bezogen auf die minimal zulässige Länge eines Backoffslots sind neben dem maximalen Tickoffset auch die Wahl des BTSP sowie die maximale Signalverzögerung innerhalb des Netzes. Abbildung 4.8 illustriert diese Zusammenhänge anhand eines Beispiels, bestehend aus zwei Knoten. Wir betrachten wieder einen Sender (S) und einen Empfänger (E) mit einem maximalen Tickoffset von $d^P_{maxTickoffset}$. Da wir die minimal zulässige Länge des Backoffslots ermitteln wollen, eilt bei diesem Beispiel die Uhr des Empfängers der des Senders voraus. In der Abbildung bezeichnet

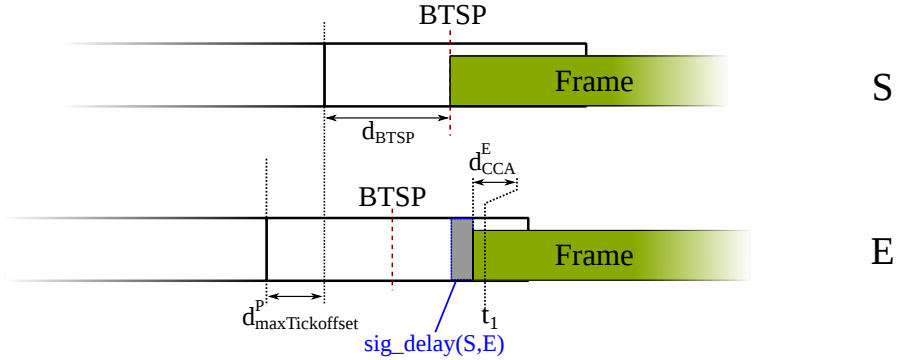


Abbildung 4.8: Erkennung der Medienbelegung bei Verwendung des CCA-Mechanismus.

[20] Die Zeitspanne, die der CCA-Mechanismus benötigt, hängt von der Implementierung, den gewählten Konfigurationsparametern, der Technologie, aber auch von der Topologie und Ausdehnung des Netzes ab (z. B. spielt bei Funknetzen die Stärke des empfangenen Signals eine Rolle).

[21] Häufig besteht ein Knoten aus einem separaten Sendebaustein oder Transceiver und einem an diesen angeschlossenen Controller.

$sig\_delay(S,E)$ die Signalverzögerung zwischen S und E; d^E_{CCA} ist die Dauer, die der Knoten E bei dieser Übertragung benötigt, um die Belegung des Mediums zu erkennen. Zum Zeitpunkt t_1 erkennt der Knoten E, dass in diesem Backoffslot eine Übertragung gestartet wurde.

Da die CCA- sowie die Signalverzögerung für jede Übertragung unterschiedlich ausfallen können, verwenden wir Worst-Case-Abschätzung für die Herleitung von Constraint 4.2 zur Bestimmung der minimalen Länge eines Backoffslots.

Constraint 4.2 (Konfiguration des Backoffslots)

Gegeben sei eine Menge von Knoten V eines Single-Hop-Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren und ein (Tick-)Synchronisationsprotokoll P. Des Weiteren sei $V_{MB} \subseteq V$ die Menge aller modusbasierten Knoten, d.h. aller Knoten mit mindestens einem Slot-Assignment in einem modusbasierten Mikroslot. Zusätzlich bezeichnet d^v_{maxCCA} die obere Zeitschranke, die der CCA-Mechanismus des Transceivers des Knotens $v \in V$ für die Erkennung der Medienbelegung und Benachrichtigung des Knotens benötigt. Dann muss die Länge d_{back} des Backoffslots so gewählt werden, dass

$$d_{back} \geq d^{MIN}_{back}$$

mit

$$d^{MIN}_{back} \equiv d^P_{maxTickoffset} + d_{BTSP} + sig\_delay^{V_{MB}\rightarrow V}_{max} + \max_{v\in V} d^v_{maxCCA}$$

aufweisen. Des Weiteren gilt, dass alle Backoffslots eines Makroslots die gleiche Länge besitzen, d.h. d_{back} ist wie d_{BTSP} ein globaler Konfigurationsparameter.

Die Constraints 4.1 und 4.2 ermöglichen es, die Backoffslots anhand einfacher Kenngrößen zu definieren. Wichtig ist, dass in dem Ausdruck in Constraint 4.2 die maximale CCA-Verzögerung aller Knoten berücksichtigt wird, damit alle Knoten die empfangenen Rahmen korrekt interpretieren können. In beiden Fällen geben die Constraints jeweils nur eine untere Schranke für die Konfigurationsparameter an. Wählt der Entwickler für die beiden Parameter größere Werte, so führt dies zu einer Erhöhung der Robustheit und zusätzlichem Sicherheitsspielraum.

Verwendet man in Constraint 4.2 anstelle von $sig\_delay^{V_{MB}\rightarrow V}_{max}$ direkt $sig\_delay^{V}_{max}$, so kann bei einer eventuellen Anpassung des Ablaufplanes jeder Knoten als Sender agieren, ohne dass Anpassungen an der Konfiguration erforderlich sind. In der Praxis bietet es sich ohnehin an, an dieser Stelle die technisch maximale Signalverzögerung zu verwenden, anstatt diese für eine konkrete Topologie zu bestimmen. Entsprechend gilt dies natürlich auch in Bezug auf die Dimensionierung des BTSP (Constraint 4.1).

4.4.2.1 Reduktion der minimalen Länge der Backoffslots

Optimierungspotential hinsichtlich der Backoffslots besteht, wenn sich die minimale Signalverzögerung $sig\_delay^{V_{MB}\rightarrow V}_{min}$ innerhalb eines spezifischen Netzwerkes exakt ermitteln lässt. Ist diese bekannt, kann ein Sender seine Übertragung $sig\_delay^{V_{MB}\rightarrow V}_{min}$ vor dem BTSP starten. Dies setzt natürlich voraus, dass die untere Schranke von Constraint 4.1 nicht durch die Umschaltzeit, sondern durch den maximalen Tickoffset geprägt ist, ansonsten stellt sich keine Zeitersparnis ein. Unter idealen Umständen kann somit eine Reduktion des Overheads um $sig\_delay^{V_{MB}\rightarrow V}_{min}$ pro Backoffslot realisiert werden.

Weitere (geringfügige) Verbesserungen lassen sich erzielen, wenn die BTSPs sowie Längen der einzelnen Backoffslots unterschiedlich gewählt werden dürfen, in Abhängigkeit von den konkreten Sendern sowie deren maximalen Signalverzögerungen.

Die vorgeschlagenen Optimierungen sind jedoch mit massiven Einschränkungen hinsichtlich der Anpassbarkeit des Ablaufplans behaftet, erfordern einen sehr hohen Konfigurationsaufwand und sind problematisch bei Änderungen der Topologie (z. B. in drahtlosen Netzen). Aufgrund des geringen Einsparpotentials und der geschilderten Schwierigkeiten verzichten wir auf eine detaillierte Betrachtung.

4.4.3 Die interne Struktur der Mikroslots

Viele der Aspekte, die bei der Dimensionierung der Backoffslots berücksichtigt werden, gelten analog auch für die Mikroslots. So muss ebenfalls sichergestellt werden, dass auch bei maximal erlaubtem Tickoffset $d^P_{maxTickoffset}$ Übertragungsstart und -ende bei allen Knoten innerhalb der Grenzen des Mikroslots liegen. Nur so ist sichergestellt, dass angrenzende Slots nicht gestört werden. Die hieraus resultierenden Anforderungen an Aufbau und Konfiguration der Mikroslots sind nicht spezifisch für die *Mode-Based Scheduling with Fast Mode-Signaling*, sondern gelten auch bei exklusiver Nutzung oder einem klassischen Wettbewerb innerhalb eines Mikroslots. Daher betrachten wir zunächst diese klassischen Anwendungsfälle und widmen uns erst im Anschluss den spezifischen Erfordernissen der modusbasierten Kommunikation.

4.4.3.1 Allgemeiner Aufbau und Anforderungen

Analog zu den BTSP definieren wir mit dem Mikroslot Transmission Start Point (MTSP) den nominellen Zeitpunkt, zu dem innerhalb eines Mikroslots eine Übertragung gestartet wird.

Definition 4.9 (Mikroslot Transmission Startpunkt – MTSP)

Der Mikroslot Transmission Startpunkt (MTSP) legt den Startzeitpunkt einer Rahmenübertragung innerhalb eines Mikroslots fest. Dieser Startzeitpunkt wird von dem jeweiligen Sender anhand seiner lokalen Uhr bestimmt. Die Zeitdauer $d_{MTSP,s}$, ist als Zeitspanne zwischen dem Beginn eines Mikroslots $s \in S$ und dessen MTSP definiert.

Die Vorgaben, die für die Dimensionierung des MTSP (Constraint 4.3) gelten, müssen sowohl bei der modusbasierten Kommunikation als auch bei einer klassischen Nutzung der Mikroslots (exklusive Nutzung oder Wettbewerb) eingehalten werden. Nur so kann die Integrität angrenzender Slots sichergestellt werden. Auch beim MTSP berücksichtigen wir, neben dem maximalen Tickoffset, die Umschaltzeiten eines nicht Vollduplex-fähigen Transceivers.

Constraint 4.3 (Konfiguration des MTSP)

Gegeben sei eine Menge von Knoten V eines Single-Hop-Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren und ein (Tick-)Synchronisationsprotokoll P. Der maximale Tickoffset der Knoten V zueinander ist beschränkt durch $d^P_{maxTickoffset}$. Im Folgenden sei $s \in S$ ein Mikroslot des Makroslots S und $V_s \subseteq V$ die Knoten, welche in dem Mikroslot senden dürfen. Des Weiteren bezeichnet $d^v_{RX \to TX}$ die Totzeit des Transceivers des Knotens $v \in V$, die für das Umschalten vom Empfangs- in den Sendemodus anfällt.
Dann muss für die Konfiguration des MTSP des Mikroslots s gelten, dass

$$d_{MTSP,s} \geq d^{MIN}_{MTSP,s}$$

mit

$$d^{MIN}_{MTSP,s} \equiv \max(d^P_{maxTickoffset}, \max_{v \in V_s} d^v_{RX \to TX}).$$

Das Constraint 4.3 entspricht den Anforderungen bzgl. der Konfiguration des BTSP (vgl. Definition 4.1), da die gleichen Voraussetzungen erfüllt werden müssen (vgl. Erläuterungen in Kapitel 4.4.2).

Die minimale Länge eines Mikroslots $s \in S$ ist von der maximalen Übertragungsdauer[22] $d^v_{maxTransmission,s}$ innerhalb dieses Mikroslots abhängig. Jeder Sender muss vor der Übertragung prüfen, ob diese die Grenzen des Mikroslots verletzt. Ist dies der Fall, erfolgt keine Übertragung, um die Kommunikation in angrenzenden Mikroslots nicht zu stören. Zusätzlich wird gefordert, dass der Mikroslot hinreichend lang ist, sodass der Sender vor Beginn des nächsten Slots noch genügend Zeit hat, um wieder in den Empfangsmodus umzuschalten. Constraint 4.4 fasst diese Anforderungen zusammen und formuliert eine untere Schranke für die minimale Länge eines Mikroslots $s \in S$.

Constraint 4.4 (Konfiguration eines Mikroslots)

Gegeben sei eine Menge von Knoten V eines Single-Hop-Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren, und ein (Tick-)Synchronisationsprotokoll P. Der maximale Tickoffset der Knoten V zueinander ist beschränkt durch $d^P_{maxTickoffset}$. Des Weiteren bezeichnet $d^v_{TX\to RX}$ die Totzeit des Transceivers des Knotens $v \in V$, die für das Umschalten vom Sende- in den Empfangsmodus anfällt. Im Folgenden sei $s \in S$ ein Mikroslot des Makroslots S und $V_s \subseteq V$ die Knoten, welche in dem Mikroslot senden dürfen. Hierbei bezeichnet $d^v_{maxTransmission,s}$ die maximale Länge der Übertragung eines Knotens $v \in V_s$ für den Mikroslot s.
Dann muss die Länge $d_{Mikro,s}$ des Mikroslots $s \in S$ so gewählt werden, dass

$$d_{Mikro,s} \geq \max(d^P_{maxTickoffset} + d_{MTSP,s} + \max_{v\in V_s}(d^v_{maxTransmission,s} + sig\_delay^{\{v\}\to V}_{max}), \\ d_{MTSP,s} + \max_{v\in V_s}(d^v_{maxTransmission,s} + d^v_{TX\to RX})) \tag{4.1}$$

Der Ausdruck lässt sich vereinfachen, indem man annimmt, dass alle Knoten einen Rahmen mit der maximal zulässigen Übertragungsdauer $d_{maxTransmission,s}$ innerhalb des Mikroslots $s \in S$ senden. Gleichzeitig benutzen wir eine Abschätzung für die maximale Signalverzögerung sowie der Totzeiten beim Umschalten, um auf diese Weise alle Knoten als potentielle Sender zu berücksichtigen[23]. Unter diesen Annahmen muss die Länge $d_{Mikro,s}$ des Mikroslots s so gewählt werden, dass

$$d_{Mikro,s} \geq d_{MTSP,s} + d_{maxTransmission,s} + \max(d^P_{maxTickoffset} + sig\_delay^V_{max}, \\ \max_{v\in V}(d^v_{TX\to RX})) \tag{4.2}$$

Wir erläutern die Herleitung des Constraint 4.4 anhand des Ausdrucks (4.1). Der Ausdruck (4.1) bildet das Maximum über zwei Teilausdrücke. Der erste beschreibt die minimale Länge des Mikroslots aus Sicht des Empfängers und stellt sicher, dass dieser, auch wenn seine Uhr der des Senders um $d^P_{maxTickoffset}$ vorauseilt, den gesendeten Rahmen noch innerhalb des Mikroslots vollständig empfängt. Der zweite Teilausdruck beschreibt die Situation aus Sicht des Senders und stellt sicher, dass dieser nach der Übertragung des Rahmens noch rechtzeitig vor dem Beginn des nächsten Mikroslots wieder in den Empfangsmodus umschalten kann. Die Abbildung 4.9 illustriert beide Fälle.

Die Vereinfachung in Constraint 4.4 Ausdrucks (4.2) ermöglicht es, alle Knoten als potentiellen Sender zu berücksichtigen und erleichtert daher Modifikationen eines bestehenden Ablaufplans. Auch hier bietet es sich an, anstelle der maximalen Signalverzögerungen für das konkrete Netz eine obere technische Grenze zu verwenden, um so unabhängig von der konkreten Topologie zu sein. Ist die minimale Signalverzögerung innerhalb des Netzwerkes bekannt, kann diese wie bei den Backoffslots genutzt werden,

[22] Die Übertragungsdauer muss auch Verzögerungen für den Medienzugriff oder evtl. Acknowledgements berücksichtigen.

[23] Die Wahl des MTSP muss in diesem Fall ebenfalls alle Knoten als Sender berücksichtigen. Des Weiteren sollte $d_{maxTransmission,s}$ so gewählt werden, dass auch zukünftige Anforderungen abgedeckt sind.

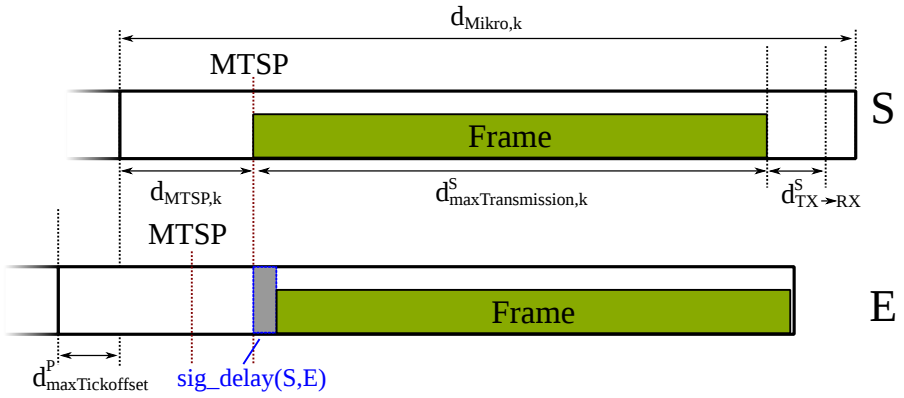


Abbildung 4.9: Einflussgrößen bei der Konfiguration eines Mikroslots.

um die minimale Länge der Mikroslots (unter Umständen) zu reduzieren. Aufgrund des in der Praxis geringen Einsparpotentials im Vergleich zu den Aufwänden und Einschränkungen, verzichten wir auch hier auf eine Betrachtung und erwähnen diese Modifikation lediglich aus Gründen der Vollständigkeit.

4.4.3.2 Der Aufbau von Mikroslots bei modusbasierter Kommunikation

Modusbasierte Mikroslots enthalten Backoffslots für die Realisierung des *Fast Mode-Signaling*. Die Anzahl der Backoffslots ist abhängig von der Anzahl der Slot-Assignments für diesen Mikroslot sowie der gewählten Abbildung der Modes auf die Backoffslots. Bei den modusbasierten Mikroslots gibt der MTSP lediglich den frühest zulässigen Zeitpunkt für einen Übertragungsstart an. Deshalb werden die Backoffslots so angeordnet, dass der BTSP des ersten Backoffslots mit dem MTSP des Mikroslots deckungsgleich ist. Zwischen den einzelnen Backoffslots existiert kein Abstand, sodass diese unmittelbar aufeinander folgen. Die Backoffslots eines Mikroslots werden fortlaufend nummeriert; der erste Backoffslot jedes Mikroslots trägt die Nummer 0. Die Abbildung 4.10 illustriert diesen Aufbau eines modusbasierten Mikroslots; die Backoffslots sind in der Abbildung in blau dargestellt.

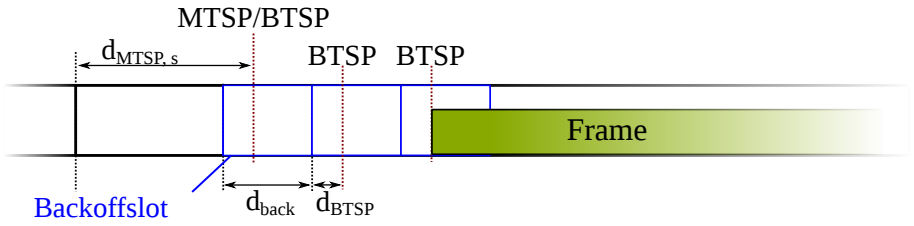


Abbildung 4.10: Aufbau und Konfiguration eines modusbasierten Mikroslots.

Für die Wahl des MTSP gelten weiterhin die in Constraints 4.3 formulierten Anforderungen, welche die Integrität angrenzender Slots sicherstellen. Die in Kapitel 4.4.2 definierten Constraints für die Dimensionierung der Backoffslots gewährleisten dann die korrekte Funktionsweise des *Fast Mode-Signaling*.

Durch die beschriebene Anordnung der Backoffslots sowie der formulierten Rolle des MTSP in modusbasierten Mikroslots müssen die Konfigurationsparameter d_{BTSP} und $d_{MTSP,s}$ entsprechend aufeinander abgestimmt werden. Dies ist Gegenstand von Constraint 4.5. Aufgrund der Vorgaben für die Wahl des BTSP (Constraint 4.1) ist für den MTSP stets sichergestellt, dass Constraint 4.3 erfüllt ist.

Constraint 4.5

Gegeben sei eine Menge von Knoten V eines Single-Hop-Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren sowie ein (Tick-)Synchronisationsprotokoll P. Ist $s \in S$ ein modusbasierter Mikroslot und d_{BTSP} der gewählte BTSP für die Backoffslots, so muss für den MTSP von s gelten, dass

$$d_{MTSP,s} \geq d_{BTSP}.$$

Die minimal zulässige Größe eines modusbasierten Mikroslots hängt von der Größe und der Anzahl der benötigten Backoffslots ab. Dieser Zusammenhang wird von Constraint 4.6 hergestellt. Die Herleitung erfolgt wieder unter Berücksichtigung der Anforderung von Sender und Empfänger in Bezug auf die benötigte Länge, wie in Kapitel 4.4.3.1 bereits erläutert (vgl. Abbildung 4.9).

Constraint 4.6 (Minimale Länge von modusbasierten Mikroslots)

Gegeben sei eine Menge von Knoten V eines Single-Hop-Netzwerkes, die über ein gemeinsames Medium miteinander kommunizieren sowie ein (Tick-)Synchronisationsprotokoll P. Der maximale Tickoffset der Knoten V zueinander ist beschränkt durch $d^P_{maxTickoffset}$. Des Weiteren bezeichnet $d^v_{TX\to RX}$ die Totzeit des Transceivers des Knotens $v \in V$, die für das Umschalten vom Sende- in den Empfangsmodus anfällt.
Im Folgenden sei $s \in S$ ein modusbasierter Mikroslot des Makroslots S und $V_s \subseteq V$ die Knoten mit einem Slot-Assignment für diesen Mikroslot. Hierbei bezeichnet $d^v_{maxTransmission,s}$ die maximale Länge der Übertragung eines Knotens $v \in V_s$ für den Mikroslot s. Des Weiteren sei neben der Modus-Präferenz-Funktion mp eine mit mp verträgliche partielle Funktion $map_{mp} : S \times M \to \mathbb{N}_0$ gegeben, welche den Modes die entsprechenden Backoffslots für das Fast Mode-Signaling zuordnet. $M_s \subseteq M$ ist die Menge der Modes, für die ein Slot-Assignment für den Slot $s \in S$ existiert (vgl. Definition 4.5). Dann muss die Länge $d_{Mikro,s}$ des Mikroslots s so gewählt werden, dass

$$d_{Mikro,s} \geq \max(d^P_{maxTickoffset} + d_{MTSP,s} + d^{transRec}_s, \\ d_{MTSP,s} + d^{transSend}_s)$$

mit

$$d^{transRec}_s = \max_{m \in M_s}(d^{SA(s,m)}_{maxTransmission,s} + sig\_delay^{\{SA(s,m)\}\to V}_{max} + k_{s,m})$$
$$d^{transSend}_s = \max_{m \in M_s}(d^{SA(s,m)}_{maxTransmission,s} + d^{SA(s,m)}_{TX\to RX} + k_{s,m})$$
$$k_{s,m} = d_{back} \cdot map_{mp}(s,m)$$

Dieser Ausdruck lässt sich vereinfachen, indem man annimmt, dass alle Knoten einen Rahmen mit der maximal zulässigen Übertragungsdauer $d_{maxTransmission,s}$ für den Mikroslot s senden. Gleichzeitig benutzen wir eine Abschätzung für die maximale Signalverzögerung sowie der Totzeiten beim Umschalten, um auf diese Weise alle Knoten als potentielle Sender modusbasierter Nachrichten zu berücksichtigen[24]. Unter diesen Annahmen muss die Länge $d_{Mikro,s}$ des Mikroslots s so gewählt werden, dass

$$\begin{aligned} d_{Mikro,s} &\geq k_{max,s} + \max(d^P_{maxTickoffset} + d_{MTSP,s} + d_{maxTransmission,s} + sig\_delay^V_{max}, \\ &\qquad d_{MTSP,s} + d_{maxTransmission,s} + \max_{v \in V}(d^v_{TX\to RX})) \\ &= k_{max,s} + d_{MTSP,s} + d_{maxTransmission,s} + \max(d^P_{maxTickoffset} + sig\_delay^V_{max}, \max_{v \in V}(d^v_{TX\to RX})) \end{aligned} \tag{4.3}$$

mit

$$k_{max,s} = d_{back} \cdot \max_{m \in M_s}(map_{mp}(s,m)).$$

4.5 Zusammenfassung

Dieses Kapitel beschreibt einen allgemeinen, technologieunabhängigen Ansatz für die Realisierung von *Fast Mode-Signaling* auf der Basis von Backoffslots. Dieser lässt sich sehr einfach in beliebige TDMA-basierte Medienzugriffsverfahren integrieren, da keine speziellen Eigenschaften der Kommunikationstechnologie vorausgesetzt werden.

Bei der Entwicklung haben wir verschiedene Strategien für die Abbildung der Modes (*Modus-Präferenzen*) auf Wartezeiten (Backoffslots) untersucht und schließlich eine optimale Abbildung definiert, um den resultierenden Overhead zu minimieren. Anhand unseres abstrakten Kommunikationsmodells eines TDMA-basierten Medienzugriffsverfahrens erfolgte dann die Entwicklung der notwendigen Konzepte für eine Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* auf Basis der Backoffslots. Hierzu haben wir die allgemeine Struktur und den Aufbau von Backoff- und Mikroslots unter Berücksichtigung möglicher relevanter Kenngrößen (Synchronisationsungenauigkeit, Signalverzögerung, Umschaltzeiten der Transceiver etc.) entworfen. Sowohl der Aufbau als auch die Dimensionierung der Backoff- und Mikroslots hängen von verschiedenen Parametern ab, welche durch die jeweilige Technologie beeinflusst werden. Die Constraints sind entsprechend allgemein formuliert, so dass die Eigenschaften und Charakteristika unterschiedlicher Kommunikationstechnologien über die Wahl der Parameter der Constraints abgebildet werden können.

So ergibt sich ein abstraktes Konzept, welches die Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* für ein breites Spektrum von unterschiedlichen drahtlosen und -gebundenen Kommunikationstechnologien ermöglicht. Die entwickelten Constraints ermöglichen es dem Designer, Aufbau und Konfiguration der Backoff- und Mikroslots so zu wählen, dass ein zuverlässiger und fehlerfreier Betrieb gewährleistet ist. Um das von uns entwickelte Konzept in der Praxis zu testen, haben wir es für die Entwicklung eines drahtlosen Kommunikationsprotokolls auf Basis von IEEE 802.15.4 [IEE03] erfolgreich eingesetzt (Kapitel 5). Auch unsere Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in FlexRay (Kapitel 6) wendet die in diesem Kapitel vorgestellten Konzepte an.

[24] Die Wahl des MTSP muss in diesem Fall ebenfalls alle Knoten als Sender berücksichtigen.

5 Umsetzung von Mode-Based Scheduling with Fast Mode-Signaling in drahtlosen Netzen

In diesem Kapitel nutzen wir die vorgestellten Konzepte zur Realisierung von *Fast Mode-Signaling* mittels Backoffslots, um *Mode-Based Scheduling with Fast Mode-Signaling* im Kontext drahtloser Sensornetze [CES04, ASSC02] zu evaluieren. Gleichzeitig ermöglicht dieser Schritt die Bewertung der Tauglichkeit des entwickelten abstrakten Lösungskonzeptes aus dem vorherigen Kapitel als Grundlage konkreter Implementierungen.

Die Unterstützung drahtloser Kommunikationstechnologien eröffnet zudem weitere Einsatzgebiete für *Mode-Based Scheduling with Fast Mode-Signaling* im Bereich der funkgestützten Steuerung und Überwachung von Produktionsanlagen [PC11, Mar12]. Drahtlose Lösungen sind insbesondere in variablen Fertigungsanlagen, an denen im Rahmen der Industrie 4.0 Initiative geforscht wird, gefragt, da diese eine höhere Flexibilität hinsichtlich Erweiterbarkeit und Modifizierbarkeit bieten als klassische, in diesem Bereich etablierte drahtgebundene Ansätze (wie z. B. Profibus [IEC07, Fel09], Modbus [IEC07] oder auch CANopen [CAN05]). Die nahtlose Integration mobiler Knoten, beispielsweise in Form mobiler Roboter wie etwa Wartungs- oder Transportsysteme, ist ein weiterer Pluspunkt. Zusätzlich steigt die Zahl der Knoten in Produktionsanlagen aufgrund des vermehrten Einsatzes von *Smart Devices*, welche neue Anwendungsmöglichkeiten im Bereich des Controllings, der Alarmierung und der Diagnostik erschließen [CNM10]. Eine klassische drahtgebundene Vernetzung wäre diesbezüglich sehr aufwändig und unflexibel. Smart Devices integrieren sich in das Netzwerk, ermöglichen mittels Sensoren die direkte Überwachung von Prozessen und Prozessgrößen und zeichnen sich durch eine gewisse Eigenintelligenz sowie umfangreiche Kontroll- und Diagnoseschnittstellen aus [The10, CNM10]. Im industriellen Umfeld existieren für dieses Szenario im Wesentlichen zwei etablierte, miteinander konkurrierende, drahtlose Kommunikationsstandards: WirelessHART und ISA 100.11a. Der Medienzugriff wird bei beiden über TDMA realisiert, um deterministische Garantien in Bezug auf die auftretenden Übertragungsverzögerungen gewährleisten zu können (vgl. Kapitel 5.6).

Anstelle einer Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* auf Basis von WirelessHART oder ISA 100.11a haben wir uns dazu entschieden, unseren *Black burst-Integrated Protocol Stack* (BiPS), um ein eigenes drahtloses modusbasiertes MAC-Protokoll zu erweitern. BiPS ist ein speziell auf eingebettete Systeme zugeschnittener Protokollstack, mit dem Ziel, echtzeitfähige, deterministische Kommunikationsprotokolle für drahtlose (Sensor-)Netzwerke bereitzustellen. Das langfristige Ziel besteht darin, einen eigenen Kommunikationsstack für das Produktionsumfeld anbieten zu können, als Alternative zu WirelessHART und ISA 100.11a. Aufbauend auf BiPS bieten wir zusätzlich eine dienstbasierte Middleware [Kra14] an. Technisch basiert BiPS ebenso wie WirelessHART und ISA 100.11a auf IEEE 802.15.4.

Dieses Kapitel ist wie folgt strukturiert: Kapitel 5.1 liefert eine Beschreibung des BiPS-Frameworks. Diese umfasst die von BiPS integrierten Protokolle sowie dessen Architektur, Schnittstellen und Hardwareplattform. In Kapitel 5.2 konzentrierten wir uns auf die Schnittstelle von BiPS für die Integration neuer Protokolle sowie deren Anwendung zur Erweiterung von BiPS um ein modusbasiertes MAC-Protokoll. Hierbei betrachten wir auch die spezifischen Anforderungen in Bezug auf die Konfiguration der Backoffslots sowie modusbasierten Mikroslots aufgrund der verwendeten Hardware. Kapitel 5.3 evaluiert unser modusbasiertes MAC-Protokoll und fasst die Ergebnisse der Evaluation zusammen. Kapitel 5.4 betrachtet alternative Ansätze zur Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling*

in drahtlosen Multi-Hop-Netzen, während Kapitel 5.5 kurz auf die Aspekte der modellbasierten Anwendungsentwicklung im Kontext von BiPS eingeht. Eine Abgrenzung von BiPS und dessen modusbasiertem MAC-Protokoll zu WirelessHART und ISA 100.11a erfolgt in Kapitel 5.6. Abschließend fasst Kapitel 5.7 noch einmal die wichtigsten Ergebnisse zusammen.

Ein Teil der Ergebnisse dieses Kapitels wurde in [11] publiziert.

5.1 Black burst-Integrated Protocol Stack (BiPS)

Der Black burst-Integrated Protocol Stack (BiPS) ist ein Framework für die Entwicklung von Anwendungen im Umfeld drahtloser Sensornetzwerke. Hierfür stellt BiPS unterschiedliche (zum Teil echtzeitfähige, deterministische) Kommunikationsprotokolle in einer effizienten Infrastruktur bereit, die von einer Anwendung über einfache Schnittstellen genutzt werden können.

Um die Anforderungen hinsichtlich der Echtzeitfähigkeit erfüllen zu können, basiert BiPS auf einer *Bare-Metal*-Realisierung. Das heißt, BiPS selbst läuft direkt auf der verwendeten Hardwareplattform (*Imote 2*, vgl. Kapitel 5.1.4) und verzichtet auf ein Betriebssystem. Hierdurch ist eine vollständige Kontrolle über die verwendete Hardware sowie deren Interrupts gewährleistet. Dies ermöglicht es, zeitkritische Protokollfunktionen mit minimalen Verzögerungen zu realisieren.

Neben der reinen Kommunikationsfunktionalität stellt BiPS daher auch (notwendigerweise) Betriebssystemfunktionalitäten zur Verfügung, um die Entwicklung und Integration von Kommunikationsprotokollen auf höheren Abstraktionsebenen (z. B. Routing-Protokolle, dienstbasierte Middleware) sowie komplexer verteilter Anwendungen, welche die von BiPS bereitgestellten Protokolle für ihre Kommunikation einsetzen, zu vereinfachen. Hierfür stellt BiPS zwei unterschiedliche Scheduler zur Verfügung; den BiPS Communication Scheduler (BCS) sowie den BiPS Application Scheduler (BAS). Der BCS kontrolliert die Ausführung der zeitkritischen Protokollfunktionen. Der BAS hingegen steuert die Ausführung von Kommunikationsprotokollen auf höheren Abstraktionsebenen und/oder Anwendungen, unter Berücksichtigung der Anforderungen des BCS. Da die Anwendungen und Kommunikationsprotokolle höherer Abstraktionsebenen in der Regel geringere zeitliche Anforderungen aufweisen, haben die durch den BCS kontrollierten funktionalen Komponenten höhere Ausführungsprioritäten.

Die grundlegende Architektur des BiPS sowie Treiber und Laufzeitumgebung für die *Imote 2*-Plattform [MEMar] wurde im Rahmen der Masterarbeit von Markus Engel 2013 entwickelt [Eng13a] und sind speziell auf die Anforderungen dieser Plattform zugeschnitten (vgl. Kapitel 5.1.4). Zusammen mit der Middleware ProMID [Kra14] stellt BiPS einen vollständigen Kommunikationsstack für das Produktionsumfeld bereit, welcher im Rahmen der SmartFactory KL bereits erfolgreich eingesetzt wurde. Die Weiterentwicklung von BiPS (BCS, BAS, Multiplexer) sowie die Integration neuer Protokolle (ACTP) erfolgte im Rahmen des Projektes *SINNODIUM - Software Innovations For the Digital Company* [SIN] durch Dennis Christmann, als Teil der Technologie-Initiative SmartFactory KL [sma], deren Ziel die Entwicklung innovativer Informations- und Kommunikationstechnik für das Produktionsumfeld ist. Der Schwerpunkt dieser Arbeit bestand in der Entwicklung und Integration eines modusbasierten MAC-Protokolls in BiPS.

5.1.1 Logische Strukturierung des Mediums

Ziel bei der Entwicklung von BiPS ist die Bereitstellung eines Protokollstacks, welcher unterschiedliche drahtlose Protokolle auf dem MAC-Layer integriert und einer Anwendung erlaubt, von den Details der eigentlichen Realisierung und Plattform zu abstrahieren. Über die Wahl bzw. die Kombination der integrierten MAC-Protokolle lassen sich unterschiedlichste Kommunikationsanforderungen abbilden, von einem wahlfreien konkurrierenden Medienzugriff bis hin zu deterministischen Garantien bzgl. der auftretenden Verzögerungen. Um dies leisten zu können, verwendet BiPS TDMA, um eine zeitliche Strukturierung des Mediums zu etablieren. Hierzu unterteilt BiPS die Zeit zunächst in Makroslots fester

Länge. Jeder Makroslot beginnt jeweils mit einer Synchronisationsregion. Der verbleibende Makroslot wird in Slotregionen mit frei wählbarer, aber fester Länge unterteilt. Das hierfür benötigte gemeinsame Zeitverständnis der Knoten etabliert BiPS durch die Verwendung des masterbasierten Black Burst Synchronisation Protokolls (BBS, [GK11, GK08, Eng13a, ECG14]). Dieses erlaubt durch den Einsatz kollisionsresistenter *Black Bursts*[1] auch in Multi-Hop-Netzen eine netzweite Synchronisation und ist dazu in der Lage, eine obere Schranke für die maximale Ungenauigkeit sowie die maximale Konvergenzzeit zu garantieren.

Für jede Slotregion kann individuell das zu verwendende MAC-Protokoll festgelegt werden. Dieses kontrolliert dann innerhalb der Slotregion den Medienzugriff. Eine solche Konfiguration eines Makroslots (Länge, Slotregionen, Zuordnung der Protokolle) wird als Makroslotkonfiguration bezeichnet. Durch die flexible Kombination verschiedener MAC-Protokolle in einem Makroslot ermöglicht BiPS eine optimale Abbildung unterschiedlichster Kommunikationsanforderungen. Durch den Umstand, dass die Protokolle innerhalb ihrer Slotregion den Medienzugriff vollständig kontrollieren, lassen sich beliebige MAC-Protokolle integrieren[2] (vgl. Kapitel 5.2.1). Aktuell unterstützt BiPS neben einem wettbewerbsbasierten Protokoll (CB), welches an CSMA/CA angelehnt ist, auch drei deterministische Protokolle. Hierzu gehören ein klassisches, auf exklusiven Reservierungen basiertes MAC-Protokoll (RB) sowie das *Arbitrating and Cooperative Transfer Protocol (ACTP)* [CGR12]. ACTP nutzt BBs und deren Eigenschaften, um eine netzweite Arbitrierung (auch in Multi-Hop-Netzen) nach dem Vorbild von CAN zu realisieren. Das dritte Protokoll mit deterministischen Garantien ist das modusbasierte MAC-Protokoll (MB), welches in der hier implementierten Variante auf den Einsatz in Single-Hop-Netzwerken beschränkt ist. Zusätzlich können IDLE-Regionen definiert werden, in denen keine Kommunikation stattfindet. In diesen Regionen darf BiPS den Transceiver zur Reduktion des Energiebedarfs deaktivieren – dies erfolgt automatisch (sofern die IDLE-Region hinreichend lang ist) und transparent für die Anwendung.

Die Abbildung 5.1 zeigt exemplarisch den Aufbau eines Makroslots, beginnend mit der Synchronisationsregion. Der verbleibende Makroslot ist in sechs Slotregionen unterteilt, denen jeweils unterschiedliche MAC-Protokolle zugeordnet sind. Hierbei ist es auch zulässig, einem Protokoll mehrere Slotregionen innerhalb eines Makroslots zuzuordnen, z. B. um unterschiedliche Nachrichtentypen zu übertragen.

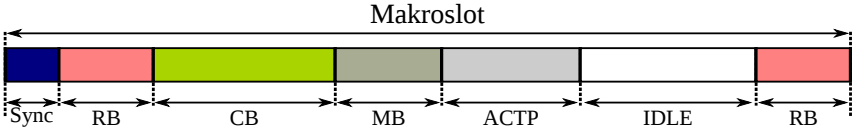


Abbildung 5.1: Aufbau von Makroslots.

Mehrere Makroslots (bzw. deren Makroslotkonfigurationen) können sequentiell aneinandergefügt werden und bilden dann einen Superslot (Superslotkonfiguration). BiPS erlaubt die Definition beliebig vieler Superslots und unterstützt zur Laufzeit den Wechsel zwischen verschiedenen Superslotkonfigurationen.

[1] Bei Black Bursts (BB) handelt es sich konzeptionell um Sendeenergie auf dem Medium, mit einer definierten Start- und Endzeit. Mit Hilfe der BBs lassen sich die Bits einer Übertragung kollisionsresistent kodieren. Eine Eins entspricht der Übertragung eines (dominanten) Signals, bei einer Null entfällt die Übertragung. BBs sind kollisionsresistent, da bei der Überlagerung mehrerer identischer BBs keine Informationen zerstört werden. Wird zeitgleich eine logische Eins (dominanter BB) und eine logische Null (rezessiver BB) gesendet, so setzt sich der dominante BB durch (logisches Oder) [Eng13a].

[2] Es muss lediglich sichergestellt sein, dass die von ihnen veranlassten Übertragungen die Grenzen der Slotregionen nicht verletzen.

5.1.2 Architektur von BiPS

BiPS basiert auf einer klassischen Schichtenarchitektur und abstrahiert soweit wie möglich von den konkreten Eigenschaften der verwendeten Hardwareplattform. Die Abbildung 5.2 zeigt die verschiedenen Schichten von BiPS. Die Schichten 0 und 1 erbringen die klassischen Dienste eines Betriebssystemkerns. Schicht 1 implementiert Treiber für spezifische Hardwarekomponenten wie Transceiver oder UART, die von den höheren Schichten verwendet werden, während Schicht 0 allgemeine Funktionen für den Zugriff auf die Hardware im Low-Level Bereich zur Verfügung stellt. Hierzu gehört die Verwaltung von DMA-Transfers, Hardware-Timer, Zugriff auf Hardware-Pins und der Umgang mit Interrupts[3].

Schicht 2 umfasst die eigentlichen MAC-Protokolle, die innerhalb der Slotregionen zum Einsatz kommen (vgl. Kapitel 5.1.1), sowie BBS zur Synchronisation. Der BCS muss sicherstellen, dass die zeitkritischen Protokollfunktionen rechtzeitig und mit möglichst geringen Verzögerungen ausgeführt werden[4]. Um Unterbrechungen und Verzögerungen zu vermeiden, führt der BCS die zeitkritischen Funktionen der Protokolle im Interruptkontext unter Verwendung eines Hardware-Timers aus. Für die Ansteuerung der Protokolle verwendet der BCS eine einheitliche von BiPS definierte Protokollschnittstelle. Diese ermöglicht die einfache Integration neuer MAC-Protokolle, wie z. B. dem MB Protokoll (Kapitel 5.2).

Die Schichten 3 und 4 sind Kommunikationsprotokollen auf höheren Abstraktionsebenen (z. B. Clustering, Routing oder der dienstbasierten Middleware ProMID) und den eigentlichen Anwendungen (die BiPS für ihre Kommunikation nutzen) vorbehalten. Die Schnittstelle zwischen den höheren Schichten (Schicht 3 und 4) und den in BiPS integrierten MAC-Protokollen wird über den Multiplexer realisiert. Dieser erlaubt eine Entkopplung zwischen den MAC-Protokollen mit ihren harten Echtzeitanforderungen und den eigentlichen Anwendungen mit geringeren Anforderungen bzgl. Echtzeitfähigkeit (Kapitel 5.1.3).

Der BAS ist dafür verantwortlich, Kommunikationsprotokolle oder Anwendungen auf höheren Abstraktionsebenen (Schicht 3 und 4) zu aktivieren, ohne dass diese zeitkritische Funktionalitäten auf den niedrigeren Schichten stören, unterbrechen oder verzögern. Deshalb wurde der BAS als ereignisbasierter Scheduler für kooperative Tasks entworfen. Der BAS unterstützt keine separaten Threads, sondern verwendet stattdessen einfache Callbacks, um die Anwendungen auszuführen. Da er selbst die Ausführung der Callbacks nicht im Sinne eines Kontextwechsels unterbrechen kann, eignet er sich nur für kooperative Anwendungen, die eigenständig die Kontrolle an den BAS zurückgeben. Anwendungen oder Protokolle auf Schicht 3 bzw. 4 können mit Hilfe des BAS selbstdefinierte Callbacks, mit eigenen oder bereits von

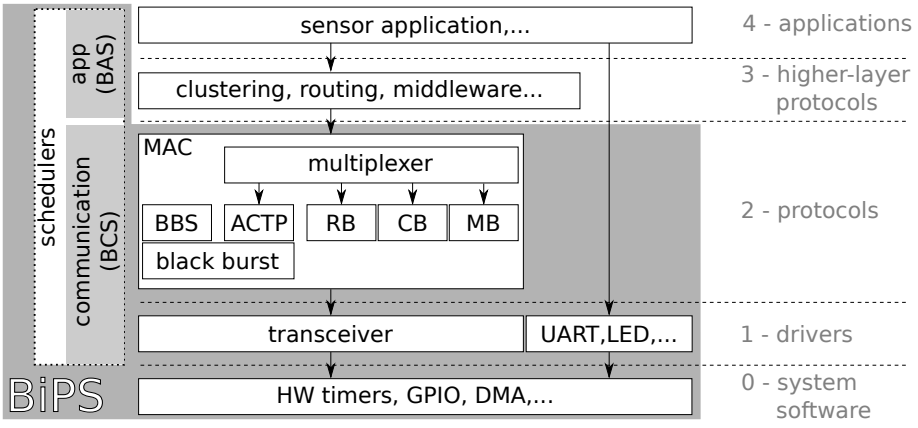


Abbildung 5.2: Aufbau des Black burst-Integrated Protocol Stack (BiPS).

[3]Eine Schnittstelle erlaubt es Treibern oder anderen BiPS-Komponenten, eigene Interrupthandler zu registrieren, um über diese schnell auf Ereignisse reagieren zu können. Der Treiber des Transceivers nutzt z. B. diese Möglichkeit, um ohne Verzögerungen über eine Medienbelegung (auf Basis des CCA-Mechanismus) informiert zu werden.

[4]Hierzu gehört z. B. die rechtzeitige Aktivierung der Protokolle zu Beginn der ihnen zugeordneten Slotregion(en).

BiPS vordefinierten Ereignissen (z. B. Timer für ein zeitgesteuertes Auslösen) assoziieren. Bei deren Auftreten führt der BAS den assoziierten Callback (außerhalb des Interruptkontextes) aus. Somit ist sichergestellt, dass der BCS einen solchen Callback jederzeit für zeitkritische (Protokoll-)Operationen unterbrechen kann.

5.1.3 Der BiPS-Multiplexer

Aufgabe des BiPS-Multiplexers ist die temporale Entkopplung zwischen den harten Echtzeitanforderungen der MAC-Protokolle (Schicht 2) sowie den Anwendungen[5] (Schicht 3 und 4). Hierzu stellt der Multiplexer den Anwendungen eine einheitliche Schnittstelle für den Zugriff auf die in BiPS integrierten MAC-Protokolle zur Verfügung. Die zeitliche Entkopplung wird durch intelligente Übertragungspuffer sowohl in Sende- als auch in Empfangsrichtung realisiert.

Die Übertragungspuffer in Senderichtung werden als *TX transmission opportunities (TX TOs)* bezeichnet und speichern ausgehende Nachrichten solange, bis das entsprechende MAC-Protokoll diese überträgt. Will eine Anwendung eine Nachricht übertragen, genügt es, diese in der entsprechenden TX TO abzulegen. Beim Anlegen einer TX TO kann die Anwendung über verschiedene Parameter (max. Anzahl Übertragungsversuche, Priorität, Größe der Warteschlange etc.) deren Semantik und Verhalten konfigurieren. So kann eine einzige TX TO mit einer beliebigen Anzahl von Slotregionen, die dem gleichen MAC-Protokoll angehören, verknüpft werden. Jeder TX TO wird zudem eine eindeutige Priorität zugeordnet, bei mehreren (derselben Slotregion zugeordneten) übertragungsbereiten TX TOs wird die Nachricht derjenigen TX TO mit der höchsten Priorität für die Übertragung ausgewählt. Sobald eine Slotregion beginnt, transferiert der Multiplexer die entsprechende Nachricht aus der jeweiligen TX TO an das verantwortliche MAC-Protokoll. Daraufhin erstellt das Protokoll aus der Nachricht einen Rahmen und versucht diesen zu übertragen. Anschließend wird der Multiplexer über das Resultat des Übertragungsversuches informiert. Abhängig von der Konfiguration des TX TO wird die Rückmeldung an die Anwendung weitergereicht oder eine Wiederholung der Übertragung veranlasst.

In der Standardeinstellung verwendet jede TX TO Warteschlange eine FIFO-Strategie. Der Multiplexer unterstützt jedoch auch die Spezifikation von Deadlines für Nachrichten. In diesem Fall wird die zu versendende Nachricht mittels Earliest Deadline First (EDF) innerhalb der TX TO Warteschlange gewählt und an das MAC-Protokoll übergeben[6]. Nachrichten mit abgelaufener Deadline werden verworfen und die Anwendung hierüber informiert.

Analog zu den TX TOs stellt der Multiplexer ebenfalls Empfangspuffer (RX TOs) zur Verfügung. Diese können mit einer oder mehreren Slotregionen verknüpft werden und verfügen ihrerseits über eine Warteschlange. Anwendungen können sich bei den RX TOs Callbacks registrieren, um über den Empfang von Rahmen informiert zu werden.

Die Abbildung 5.3 illustriert das Zusammenspiel zwischen Multiplexer, Transmission Opportunities, Slotregionen und Anwendungen. In dem gezeigten Beispiel nutzen zwei Applikationen den Multiplexer, um Rahmen zu versenden. Der Superslot besteht aus zwei Makroslots mit insgesamt sechs Slotregionen sowie drei TX TOs und zwei RX TOs. Über die Funktionen `enqueue` können Nachrichten für eine anschließende Übertragung in den jeweiligen TX TOs abgelegt werden. Der Multiplexer stellt die Nachrichten dem jeweiligen MAC-Protokoll bei dessen Aktivierung durch den Aufruf der Funktion `setData` zur Verfügung. Beim Empfang eines Rahmens wird dieser vom MAC-Protokoll über die `rxCallback`-Funktion an den Multiplexer übergeben. Dieser hinterlegt die Nachrichten in den entsprechenden RX TOs und informiert die Anwendungen über den (zuvor hierfür registrierten) Callback.

[5] Aus der Sicht von BiPS kann eine Anwendung im Folgenden sowohl ein Kommunikationsprotokoll auf höherer Abstraktionsebene (Schicht 3) als auch eine Applikation auf Schicht 4 sein.

[6] Ist einem MAC-Protokoll mehr als eine TX TO zugeordnet, wird zuerst diejenige übertragungsbereite TX TO mit der höchsten Priorität ausgewählt und erst dann mittels EDF die Nachricht selektiert.

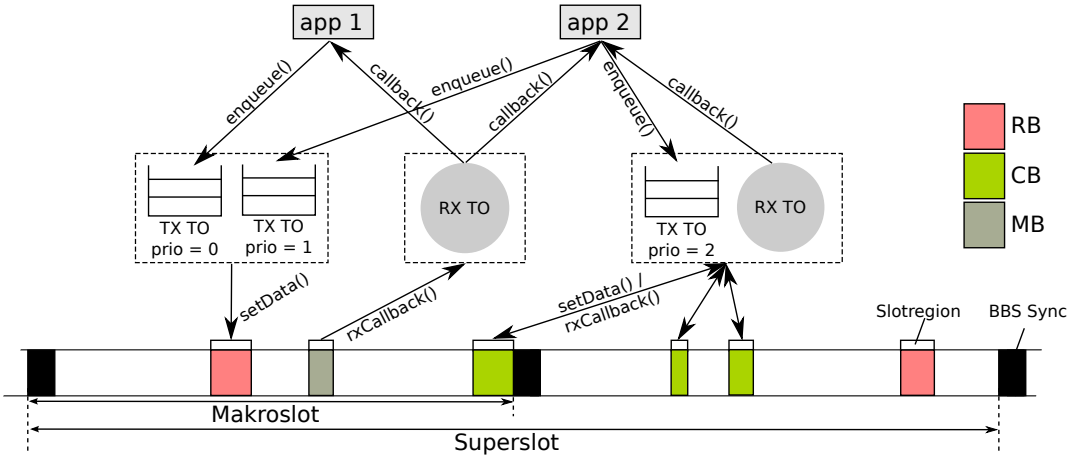


Abbildung 5.3: Darstellung des Zusammenhangs zwischen TX TOs, RX TOs und den Slotregionen eines Makroslots [BCGM14].

Durch den Einsatz der TX TO wird eine temporale Entkopplung zwischen Anwendung und MAC-Protokoll realisiert, da Nachrichten zu jedem beliebigen Zeitpunkt generiert und an den Multiplexer übergeben werden können. Dieser stellt sicher, dass die Nachricht zum nächstmöglichen Zeitpunkt versendet wird. Eine Synchronisation zwischen Anwendung und MAC-Protokoll ist somit in keiner Weise erforderlich. Dies gilt umgekehrt auch für den Einsatz der RX TOs, die Anwendung muss sich nicht selbst um den Empfang kümmern, sondern wird automatisch informiert, sobald ein Rahmen eintrifft, für den diese sich interessiert. Neben der temporalen Entkopplung wird über den Multiplexer und die TOs auch eine lose Kopplung zwischen Anwendung und den eigentlichen MAC-Protokollen sowie der Makroslotkonfiguration erreicht. Hierdurch wird die Austauschbarkeit der MAC-Protokolle[7] und Anpassungen der Makroslotkonfiguration erleichtert.

5.1.4 Die Imote 2-Plattform

BiPS wurde speziell für die *Imote 2*-Plattform [MEMar, Cro07] entwickelt. Hierbei handelt es sich um eine Plattform für Wireless Sensor Networks, die ursprünglich von Intel Research [Int06] stammt, deren Vertrieb jedoch 2011 von Memsic Inc. eingestellt wurde. Um die Anforderungen bei der Realisierung echtzeitfähiger, deterministischer Protokolle erfüllen zu können, verzichtet BiPS auf die Verwendung eines Betriebssystems. Stattdessen kommt eine *Bare-Metal* Umgebung zum Einsatz, welche die vollständige Kontrolle über die Hardwareplattform, deren Ressourcen und Verhalten ermöglicht. Ziel ist es, hierdurch auftretende Verzögerungen zu minimieren oder zu kompensieren, um die Ausführung zeitkritischer, protokollspezifischer Funktionen möglichst exakt zu den vordefinierten Zeitpunkten gewährleisten zu können.

Die Abbildung 5.4 zeigt die Vorder- und Rückseite eines *Imote 2*-Sensorknotens. Kernstück ist der Intel XScale[8] Prozessor PXA271 [PXA13], welcher auf der ARM-v5TE-Architektur [ARM05] basiert. Um Energie zu sparen, unterstützt der PXA271, neben verschiedenen Schlafmodi, auch den Betrieb mit unterschiedlichen Taktraten zwischen 13 MHz und 416 MHz. Der verwendete PXA271 Chip beinhaltet neben dem CPU Kern mit 256 kB SRAM noch 32 MB SDRAM sowie 32 MB Flash Speicher. Über die in der Abbildung gezeigten weißen Konnektoren ist der Zugriff auf einen Teil der vom PXA271

[7] Auch wenn hier keine vollständige Transparenz und Entkopplung möglich ist, da verschiedene MAC-Protokolle für den Versand von Nachrichten spezielle Parameter benötigen, welche die Anwendung bereitstellen muss.

[8] Die XScale-Prozessorlinie wurde an die Marvell Technology Group verkauft, die deren Entwicklung fortführt.

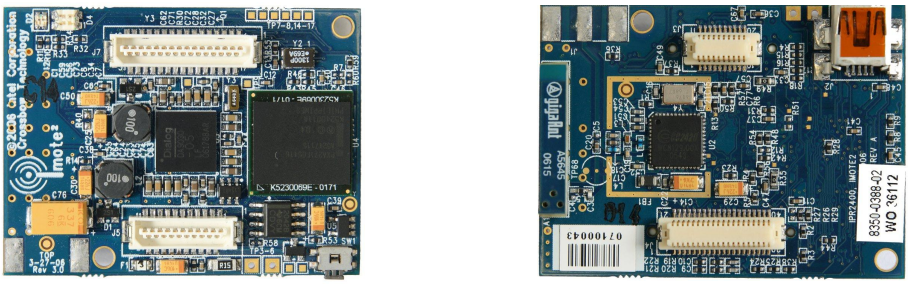


Abbildung 5.4: Ober- und Unterseite des *Imote 2*.

bereitgestellten Schnittstellen möglich. Zu diesen gehören neben I^2C und Synchronous Serial Ports (SPI, [Mot00]) auch mehrere Universal Asynchronous Receiver Transceiver (UARTs) sowie verschiedene GPIO-Pins (vgl. [Int06, Cro07].). Der PXA271 stellt dem Entwickler auch eine Reihe von Hardware-Timern zur Verfügung, die als Interruptquelle konfiguriert werden können und von BiPS für die Ausführung zeitkritischer Funktionen (BCS, Protokollfunktionalitäten) eingesetzt werden.

Für die Kommunikation integriert der *Imote 2* den IEEE 802.15.4-konformen CC2420-Transceiver von Texas Instruments [Tex07] (ehemals ChipCon). Der CC2420-Transceiver ermöglicht Übertragungen im 2.4 GHz Band (16 Kanäle) mit einer Übertragungsrate von 250 kBit/s. Da es sich bei dem CC2420-Transceiver um einen integrierten Sende- und Empfangsbaustein mit einer einzelnen Antenne handelt, muss jeweils zwischen Sende- und Empfangsmodus umgeschaltet werden. Das heißt, während der CC2420-Transceiver einen Rahmen überträgt, kann er nicht gleichzeitig das Medium abhören.Während des Wechsels des Betriebsmodus ist der CC2420-Transceiver taub. Für den Wechsel vom Empfangs- in den Sendemodus benötigt dieser, je nach gewählter Konfiguration, 8 bzw. 12 Symbolperioden ($d_{rx\rightarrow tx}^{CC2420} = 128\,\mu s$ bzw. $d_{rx\rightarrow tx}^{CC2420} = 192\,\mu s$) für die Kalibrierung des internen Frequenzgenerators [Eng13a, Tex07]. Der Wechsel vom Empfangs- in den Sendemodus benötigt 12 Symbolperioden ($d_{tx\rightarrow rx}^{CC2420} = 192\,\mu s$).

Die Anbindung des CC2420-Transceivers an den PXA271 Prozessor erfolgt über einen SPI-Bus und 4 zusätzliche GPIO-Pins, die Zustandssignalisierungen des CC2420-Transceivers an den PXA271 über Signalflanken erlauben. Der SPI-Bus wird in unserer Konfiguration mit einem Takt von 13 MHz betrieben. Um einen Rahmen zu senden, müssen zunächst die Nutzdaten über SPI in den TXFIFO des CC2420-Transceivers übertragen werden. Anschließend kann durch das Senden des `STXON` bzw. `STXONCCA`-Kommandos – ebenfalls über SPI – die Übertragung gestartet werden. Die vier zusätzlichen GPIO-Pins sind mit den Pins `CCA`, `SFD`, `FIFO` und `FIFOP` des CC2420-Transceivers verbunden. Diese Pins (bzw. deren Signalflanken) können genutzt werden, um Interrupts auszulösen, sodass auf eine Zustandsänderung unmittelbar reagiert werden kann.

Das `CCA`-Signal repräsentiert den Zustand des Mediums, wie er durch den CCA-Mechanismus (CCA – Clear Channel Assessment) des CC2420-Transceivers erkannt wurde. Der CC2420-Transceiver implementiert alle drei Varianten des CSMA-CA bzw. des CCA-Mechanismus, die durch den IEEE 802.15.4-Standard beschrieben werden[IEE03, Tex07]). Die von uns genutzte Variante integriert den auf dem Medium detektierten Energiepegel über 8 Symbolperioden ($d_{maxCCA} = 128\,\mu s$) und erkennt das Medium als belegt, sobald ein zuvor konfigurierter Schwellwert überschritten wird[9]. Das `SFD`-Signal signalisiert den Empfang oder Versand des SFD-Feldes innerhalb des Headers des IEEE 802.15.4 Rahmens während der Übertragung und ist für die Synchronisation gedacht, z. B. bei der Nutzung von Beacons. Das Datenblatt des CC2420-Transceivers gibt eine typische Latenz von 3 µs zwischen Versand bzw. Empfang des

[9] Die anderen beiden Varianten des CCA-Mechanismus prüfen zusätzlich, ob das empfangene Signal ein valider IEEE 804.15.4 Rahmen ist. Da dies keine notwendige Voraussetzung für BBs ist (z. B. aufgrund von Überlagerungen) und BBs innerhalb von BBS und ACTP zum Einsatz kommen, nutzt BiPS diese CCA-Modi nicht. Zudem benötigen diese Mechanismen länger für die Erkennung einer Übertragung und erfordern daher längere Backoffslots.

SFD-Feldes und der dazugehörigen SFD-Signalisierung an. Die Signale `FIFO` und `FIFOP` dienen als Indikator für den Füllstand des RXFIFO (Empfangspuffers), des CC2420-Transceivers oder zeigen den vollständigen Empfang eines Rahmens an.

5.2 Integration von Mode-Based Scheduling with Fast Mode-Signaling in BiPS

Für die Integration neuer MAC-Protokolle stellt BiPS eine einheitliche Protokoll-Schnittstelle bereit, über welche die Ansteuerung der Protokolle durch den BCS sowie der Datenaustausch mit dem Multiplexer erfolgt. Unsere Implementierung von *Mode-Based Scheduling with Fast Mode-Signaling* für BiPS basiert auf der in Kapitel 4 beschriebenen Strategie zur Umsetzung des *Fast Mode-Signaling* mittels Backoffslots (passives *Fast Mode-Signaling*). Diese Variante der Realisierung ist auf den Einsatz in Single-Hop-Netzwerken beschränkt, einen Lösungsansatz für Multi-Hop-Netzwerke skizzieren wir in Kapitel 5.4. Bevor wir uns in Kapitel 5.2.2 mit den Details der Umsetzung des modusbasierten MAC-Protokolls für BiPS (MB) beschäftigen, folgt zunächst die Beschreibung der Protokollschnittstelle von BiPS.

5.2.1 Allgemeine Schnittstelle für Protokolle

Um beliebige Protokolle in BiPS integrieren zu können, wurde die Protokollschnittstelle möglichst generisch gestaltet. Die definierten Methoden ermöglichen sowohl die Steuerung der Ausführung des Protokolls durch den BCS als auch den Datenaustausch mit dem Multiplexer. Abbildung 5.5 zeigt die Schnittstelle als UML-Klassendiagramm.

Der BCS ist unter anderem dafür verantwortlich, die zeitkritischen Protokollfunktionen (Methoden der Protokollschnittstelle) möglichst exakt zu den vordefinierten Zeitpunkten auszuführen. Hierzu verwendet dieser einen Hardware-Timer des PXA271. Innerhalb des Timerinterrupts ruft der BCS dann die jeweiligen Protokollfunktionen auf. Hierdurch erfolgt die Ausführung der zeitkritischen Protokollfunktionen vollständig innerhalb des Interruptkontextes und kann nicht durch andere Operationen unterbrochen werden[10]. Durch die Nutzung von Interrupts können Anwendungen, die der BAS ausführt, jederzeit für zeitkritische Protokollfunktionalitäten unterbrochen werden.

Wir beschreiben die einzelnen Methoden der Protokollschnittstelle und deren Aufgaben direkt im Kontext ihrer Verwendung. Das Sequenzdiagramm in Abbildung 5.6 stellt dar, wann und in welcher Reihenfolge der BCS die einzelnen Methoden aufruft und setzt diese in Relation zum Aufbau einer

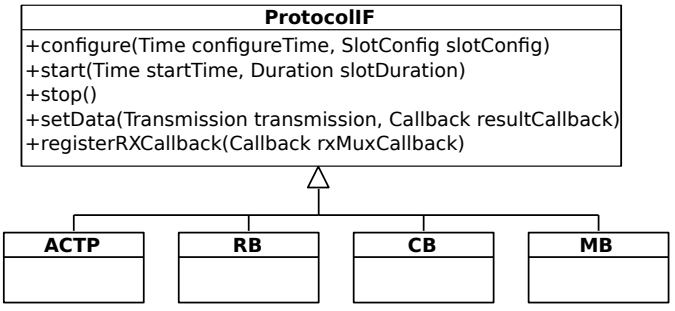


Abbildung 5.5: BiPS Schnittstelle für MAC-Protokolle.

[10] Um das Programmiermodell einfach zu halten, verzichten wir auf geschachtelte Interrupts, obwohl diese prinzipiell durch die Kombinationen von normalen Interrupts mit den sogenannten Fast Interrupts (FIRQ) eingeschränkt möglich wären (siehe [PXA13, Eng13a]).

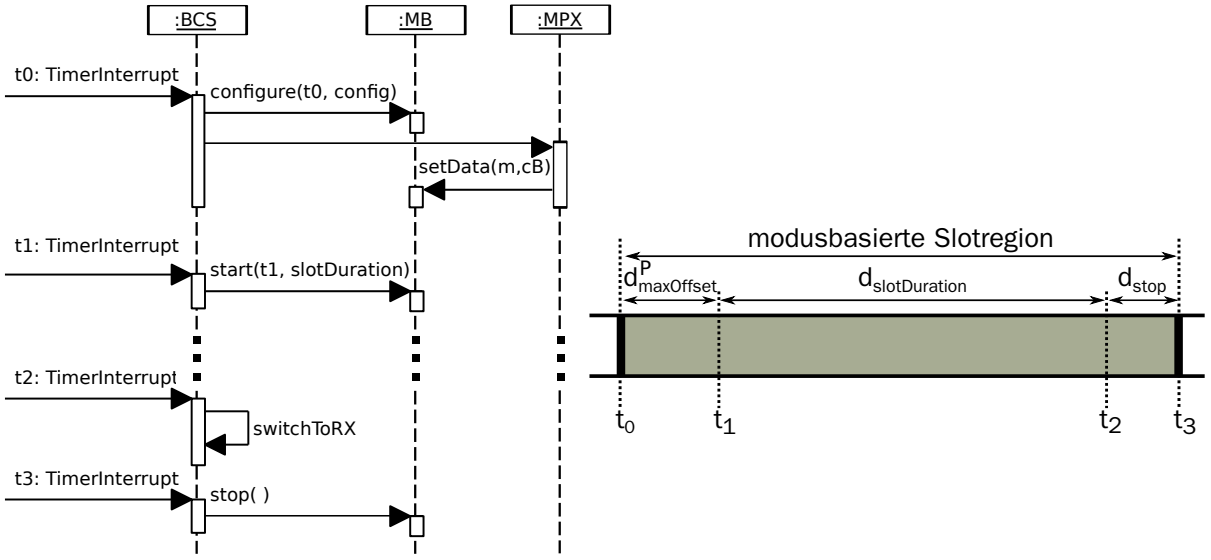


Abbildung 5.6: Aktivierung der MAC-Protokolle durch den BCS.

Slotregion[11]. Unmittelbar vor dem Beginn einer Slotregion ruft der BCS die Methode `configure` des mit dem Slot assoziierten Protokolls auf. Als Parameter wird der Zeitpunkt t_0, zu dem die Slotregion startet, sowie die für das Protokoll hinterlegte protokollspezifische Konfiguration der Region übergeben. Anschließend führt der BCS den Multiplexer (MPX) aus, welcher dem Protokoll (MB) die ggf. zu übertragenden Nachrichten übergibt, samt eines Callback für Rückmeldungen (Aufruf von `setData`).

Zum Zeitpunkt t_1 erfolgt der Aufruf der `start`-Methode. Hierdurch wird das Protokoll beauftragt, den Medienzugriff bis zum Zeitpunkt $t_2 = t_1 + d_{\text{slotDuration}}$ zu organisieren. Der Abstand zwischen t_0 und t_1 beträgt $d^P_{\text{maxOffset}}$, wobei $d^P_{\text{maxOffset}}$ der maximale Tickoffset ist, welcher durch das verwendete Synchronisationsprotokoll P für die aktuelle Konfiguration zugesichert wird. t_1 identifiziert den frühesten Zeitpunkt, zu dem ein Knoten eine Übertragung starten darf, da nun sichergestellt ist, dass für alle Knoten die Slotregion begonnen hat (trotz einer maximalen Abweichung der lokalen Uhren von bis zu $d^P_{\text{maxOffset}}$). Außerdem stellt der BCS sicher, dass zu diesem Zeitpunkt alle Knoten empfangsbereit sind (siehe t_2). Danach ist das Protokoll selbst für die Einplanung zeitkritischer Operationen verantwortlich (z. B. um den Sendevorgang einzuleiten), hierfür steht diesem ein eigener Hardware-Timer zur Verfügung.

Bis zum Zeitpunkt t_2 muss das Protokoll sämtliche Übertragungen abgeschlossen haben, da der BCS den Transceiver für die nächste Slotregion in den Empfangsmodus schaltet oder ihn ggf. deaktiviert. Dabei ist t_2 so gewählt, dass dieser d_{stop} Zeiteinheiten vor dem (gemäß Makroslotkonfiguration) eingeplanten Ende des Mikroslots (t_3) liegt. Hierbei gilt, $t_2 = t_3 - d_{\text{stop}} = t_3 - \max(d^P_{maxOffset}, d^{CC2420}_{tx\to rx})$. Die Wahl von t_2 stellt sicher, dass eine Übertragung auch bei maximaler Uhrenabweichung $d^P_{maxOffset}$ die Grenzen einer benachbarten Slotregion nicht verletzt und der Transceiver des Senders vor Beginn des nächsten Mikroslots wieder empfangsbereit ist. Der Aufruf der Methode `stop` erfolgt durch den BCS zum Zeitpunkt t_3 und erlaubt dem Protokoll lokale Strukturen aufzuräumen und noch reservierten Speicher freizugeben.

Die Methode `registerRXCallback` setzt einen Callback, den das Protokoll aufruft, sobald ein Rahmen empfangen wurde. Innerhalb dieses Callbacks legt der Multiplexer dann den empfangenen Rahmen innerhalb der assoziierten RX TO ab.

[11] Das Beispiel zeigt eine modusbasierte Slotregion, das dargestellte Verhalten ist jedoch für alle Protokolle identisch.

5.2.2 Umsetzung von Mode-Based Scheduling with Fast Mode-Signaling

Die Umsetzung des modusbasierten MAC-Protokolls (MB) für BiPS basiert auf den in Kapitel 4 vorgestellten Konzepten und Strukturen; die Realisierung des *Fast Mode-Signaling* erfolgt passiv mittels Backoffslots. Eine modusbasierte Slotregion, d.h., eine Slotregion bei der das modusbasierte MAC-Protokoll den Medienzugriff kontrolliert, entspricht in unserer Umsetzung einem modusbasierten Mikroslot. Auf diese Weise erreichen wir die größtmöglichste Flexibilität in Bezug auf die Kombination unterschiedlicher Slotsregionen (MAC-Protokolle) in einem Makroslot.

Nachdem wir im vorherigen Abschnitt die Abläufe bei der Ausführung eines Protokolls durch den BCS sowie die allgemeine Strukturierung einer Slotregion betrachtet haben, liegt die in Abbildung 5.7 dargestellte Struktur für eine modusbasierte Slotregion nahe. Hierbei muss natürlich bei der Wahl der Länge des Backoffslots bzw. des MTSP sichergestellt werden, dass der Backoffslot nicht vor dem Start der Slotregion beginnt und der MTSP nicht vor dem Zeitpunkt t_1 liegt. Ebenfalls muss gewährleistet sein, dass Übertragungen bis zum Zeitpunkt t_2 abgeschlossen sind.

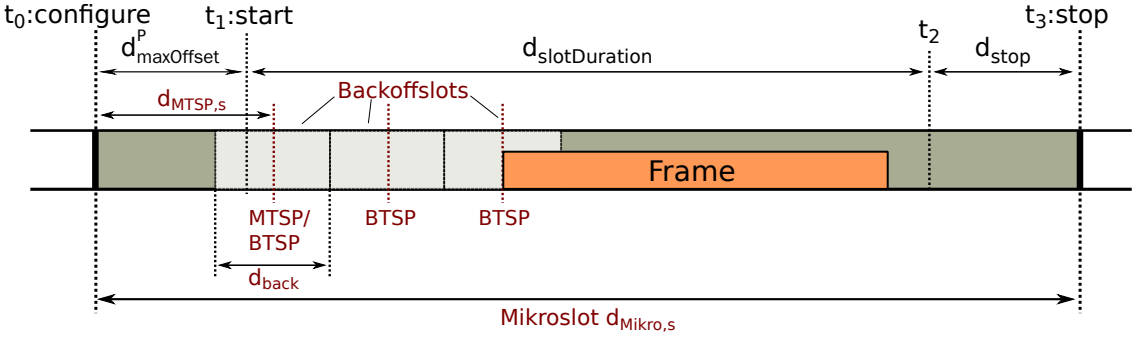


Abbildung 5.7: Aufbau einer modusbasierten Slotregion.

Um die konkreten Anforderungen an die Dimensionierung einer modusbasierten Slotregion formulieren zu können und die Zusammenhänge zwischen modusbasierten Mikroslots und Slotregionen zu beleuchten, konzentrieren wir uns zunächst auf die Mikroslots und deren Struktur. Das heißt, wir bestimmen für unsere konkrete Hardwareplattform (*Imote 2* mit CC2420-Transceiver) im Zusammenspiel mit BiPS möglichst exakt, wie MTSP, BTSP, Backoffslots und Mikroslots dimensioniert werden müssen und ignorieren hierbei zunächst den Umstand, dass es Slotregionen gibt. Hierzu verwenden wir die in Kapitel 4 abgeleiteten Constraints unter Berücksichtigung der hardware- und implementierungsspezifischen Verzögerungen. Anschließend nutzen wir diese Ergebnisse, um zu zeigen, dass der vorgeschlagene Aufbau einer modusbasierten Slotregion gemäß Abbildung 5.7 für das modusbasierte MAC-Protokoll geeignet ist.

Als Informationsquellen hinsichtlich der auftretenden (hardwarespezifischen) Verzögerungen dienen uns die Datenblätter des CC2420-Transceivers sowie des PX271, ebenso wie die Arbeit von Markus Engel zur Evaluation und Optimierung von BBS auf der *Imote 2*-Plattform [Eng13a, ECG14]. Zusätzlich werden wir auch die Auswirkungen von Implementierungsentscheidungen und Optimierungen von BiPS berücksichtigen, die wir an den entsprechenden Stellen erläutern. Um möglichst exakte Aussagen hinsichtlich der Dimensionierung zu erhalten, gehen wir von einem homogenen Netzwerk aus, in dem alle Knoten $v \in V$ auf der *Imote 2*-Plattform basieren, den CC2420-Transceiver verwenden und über eine identische Konfiguration verfügen[12]. Wie für die Realisierung von *Fast Mode-Signaling* mittels Backoffslots (vgl. Kapitel 4) gefordert, existiert ein Ticksynchronisationsprotokoll P, welches eine obere Schranke für den maximalen Tickoffset von $d^P_{maxOffset}$ garantiert.

[12] Darüber hinaus nehmen wir an, dass es pro modusbasierter Slotregion mindestens ein Slot-Assignment gibt.

5.2.2.1 Dimensionierung und Struktur der Backoffslots

Der Aufbau des Backoffslots wird durch zwei Größen festgelegt: die Wahl des BTSP (Backoffslot Transmission Startpunkt, Definition 4.8) und die Länge des Backoffslots selbst.

Festlegung des BTSP

Der BTSP bezeichnet den Startzeitpunkt einer Übertragung innerhalb des Backoffslots. Durch die Wahl des BTSP muss sichergestellt werden, dass alle anderen Knoten den Übertragungsbeginn – auch bei Auftreten der maximal zulässigen Uhrenabweichung (Tickoffset) – dem gleichen Backoffslot zuordnen können. Diesem Umstand wird durch die Berücksichtigung von Constraint 4.1 bei der Dimensionierung des BTSP Rechnung getragen. Gemäß diesem Constraint muss bei der Wahl des BTSP berücksichtigt werden, dass

$$d_{BTSP} \geq d_{BTSP}^{MIN} \text{ mit } d_{BTSP}^{MIN} \equiv \max(d_{maxOffset}^{P}, \max_{v \in V_{MB}} d_{RX \to TX}^{v}) \tag{5.1}$$

ist, um die korrekte Funktionsweise von *Fast Mode-Signaling* sicherzustellen. Hierbei bezeichnet $d_{RX \to TX}^{v}$ die Dauer für den Wechsel des Knotens $v \in V$ vom Empfangs- in den Sendemodus.

Bei der von uns verwendeten Hardware setzt sich die maximale Zeitdauer $\max_{v \in V_{MB}} d_{RX \to TX}^{v}$ für den Wechsel des Betriebsmodus aus dem Senden der entsprechenden Anweisung per SPI vom *Imote 2* an den CC2420-Transceiver sowie der Umschaltverzögerung zusammen. Bei der Übertragung über den SPI-Bus muss, neben der reinen Übertragungsverzögerung für den Befehl über SPI, auch eine Zugriffsverzögerung auf den internen Bus des PXA271 berücksichtigt werden. Wir übernehmen die Ergebnisse aus [Eng13a] und verwenden 5 µs als obere Schranke für die Zugriffsverzögerung. Die Übertragungsverzögerung ergibt sich aus der Anzahl der Bytes, aus denen der Befehl besteht, und der Übertragungsrate des SPI von 13 MBit/s. Nach dem Empfang des Umschaltbefehls führt der CC2420-Transceiver eine Kalibrierung durch. Die Kalibrierungsdauer $d_{rx \to tx}^{CC2420}$ ist von der Konfiguration abhängig, die beim Start gewählt wird, und beträgt maximal 128 µs bzw. 192 µs. Während dieser Zeit ist der CC2420-Transceiver taub und kann das Medium nicht abhören. Da der CC2420-Transceiver aber während der Kalibrierung weiterhin Daten per SPI empfangen und verarbeiten kann, lässt sich die Zeitspanne $d_{RX \to TX}^{v}$ dadurch effizienter nutzen, dass zunächst die Kalibrierung durch Senden des `STXCAL`-Befehls gestartet und währenddessen die Nutzdaten in den TXFIFO des CC2420-Transceivers übertragen werden. Unsere Implementierung startet die Kalibrierung zu Beginn des Backoffslots und überträgt, direkt im Anschluss an den `STXCAL`-Befehl, die Nutzdaten in den TXFIFO[13]. Beim Erreichen des BTSP genügt es dann, die eigentliche Übertragung durch Senden des `STXON`-Befehls über SPI zu starten. Die Zeitdauer d_{STXCAL} für die Übertragung des `STXCAL`-Befehls ergibt sich aus

$$d_{STXCAL} \approx 5\,\mu s + 0,7\,\mu s = 5,7\,\mu s.$$

Für die Übertragung der Nutzdaten – bei einer maximalen Nutzlast von 127 Bytes – ergibt sich eine Verzögerung[14] von

$$d_{PAYLOAD} \approx 5\,\mu s + 79,4\,\mu s = 84,4\,\mu s.$$

Dementsprechend folgt dann, für den Wechsel vom Empfangs- in den Sendemodus (inkl. Übermittelung eines Datenrahmens):

$$\begin{aligned} \max_{v \in V_{MB}} d_{RX \to TX}^{v} &= d_{INT} + d_{STXCAL} + \max(d_{PAYLOAD}, d_{rx \to tx}^{CC2420}) \\ &\approx 20,7\,\mu s + d_{rx \to tx}^{CC2420} \end{aligned} \tag{5.2}$$

[13] Wir nutzen an dieser Stelle, dass die Backoffslots gemäß den Constraints aus Kapitel 4.4.2 so konfiguriert werden müssen, dass auch bei maximalen Tickoffsets jeder Knoten eine im vorherigen Backoffslot begonnene Übertragung bis zum Beginn dieses Backoffslots bereits erkannt haben muss.

[14] Insgesamt müssen für die Übertragung von 127 Bytes Nutzdaten 129 Bytes über SPI übertragen werden.

Um bei Beginn des Backoffslots die Befehlssequenz möglichst exakt zum vordefinierten Zeitpunkt zu starten, verwenden wir einen Hardware-Timer. Die Verzögerung für die Behandlung eines Interrupts (Timer-Interrupt, beendeter DMA-Transfer) wird mit d_{INT} bezeichnet. Für unsere weiteren Berechnungen gehen wir für d_{INT} von einer maximalen Verzögerung von 15 µs aus (vgl. [Eng13a]). Dass die Umschaltverzögerung $d_{rx\to tx}^{CC2420}$ des CC2420-Transceiver größer ist als die Dauer für die (parallele) Übertragung der Nutzdaten, rechtfertigt unsere Implementierungsentscheidung, die Nutzdaten erst zu Beginn des Backoffslots, in dem gesendet wird, zum CC2420-Transceiver zu transferieren.

Somit fordern wir für die Wahl des Konfigurationsparameters d_{BTSP} zur Festlegung des BTSP, dass

$$d_{BTSP} \geq d_{BTSP}^{MIN} \text{ mit } d_{BTSP}^{MIN} \approx \max(d_{maxOffset}^{P}, 20,7\,\mu s + d_{rx\to tx}^{CC2420}). \tag{5.3}$$

Minimale Länge eines Backoffslots

Die Länge der Backoffslots muss so gewählt werden, dass alle Knoten eine zum BTSP gestartete Übertragung dem korrekten Backoffslot zuordnen können, trotz evtl. auftretender maximaler Uhrenabweichung. Für die Dauer eines Backoffslots d_{back} gilt gemäß Constraint 4.2, dass hierfür die Bedingung

$$d_{back} \geq d_{back}^{MIN} \text{ mit } d_{back}^{MIN} \equiv d_{maxOffset}^{P} + d_{BTSP} + sig\_delay_{max}^{V_{MB}\to V} + \max_{v\in V} d_{maxCCA}^{v} \tag{5.4}$$

erfüllt sein muss.

Die maximale Signalverzögerung $sig\_delay_{max}^{V_{MB}\to V}$ gemäß Definition 4.7 kann für unsere Implementierung und die *Imote 2*-Plattform wie folgt abgeschätzt werden (Herleitung siehe Anhang C.1):

$$\begin{aligned} sig\_delay_{max}^{V_{MB}\to V} &= \max_{v_1\in V_{MB}, v_2\in V, v_1\neq v_2} sig\_delay_{max}(v_1, v_2) \\ &= \max_{v_1\in V_{MB}, v_2\in V, v_1\neq v_2} (d_{txProcDelay}^{v_1} + d_{PropagationDelay}^{v_1,v_2} + d_{rxProcDelay}^{v_2}) \\ &\leq 36,4\,\mu s \end{aligned} \tag{5.5}$$

Verwenden wir gemäß [Tex07, Eng13a], dass die maximale Verzögerung des CCA-Mechanismus $\max_{v\in V} d_{maxCCA}^{v} = 128\,\mu s$ beträgt, so erhalten wir für die minimale Länge eines Backoffslots

$$d_{back} \geq d_{back}^{MIN} \text{ mit } d_{back}^{MIN} \approx d_{maxOffset}^{P} + d_{BTSP} + 164,4\,\mu s. \tag{5.6}$$

5.2.2.2 Dimensionierung und Struktur der Mikroslots

Auch in Bezug auf die Dimensionierung der Mikroslots für *Mode-Based Scheduling with Fast Mode-Signaling* betrachten wir – analog zum Vorgehen bei den Backoffslots – zunächst die Anforderungen an den MTSP (Mikroslot Transmission Startpunkt) und danach deren minimal zulässige Länge.

Definition des MTSP

Der MTSP (Definition 4.9) ist der früheste Zeitpunkt, zu dem eine Übertragung innerhalb des Mikroslots gestartet werden darf. Wie beim BTSP muss auch hier der maximale Tickoffset bei der Konfiguration berücksichtigt werden. Der Abstand $d_{MTSP,s}$ zwischen dem Beginn des modusbasierten Mikroslot s und dessen MTSP muss den Vorgaben von Constraint 4.3 und 4.5 genügen:

$$d_{MTSP,s} \geq d_{MTSP,s}^{MIN} \text{ mit } d_{MTSP,s}^{MIN} \equiv \max(d_{maxOffset}^{P}, \max_{v\in V_s} d_{RX\to TX}^{v}) \tag{5.7}$$

$$d_{MTSP,s} \geq d_{BTSP} \tag{5.8}$$

Bei der Implementierung von BiPS verwaltet der Multiplexer sämtliche Nachrichten und übergibt diese den Protokollen zu Beginn ihrer Slotregion. Erst nach der Ausführung des Multiplexers kann das Protokoll entscheiden, ob ein Wechsel in den Sendemodus erforderlich ist. Dieser Aspekt wird natürlich

von Constraint 4.3 nicht berücksichtigt, daher wurde Ausdruck (5.7) entsprechend ergänzt. Für die Ausführung des Multiplexers rechnen wir mit einer maximalen Verzögerung[15] von $d_{Multiplexer} = 70\,\mu s$ (Aufruf der Methode `setData`, vgl. Abschnitt 5.1.3). Unter Berücksichtigung dieses Aspekts sowie der Zusammenfassung der Ausdrücke (5.7) und (5.8) und Nutzung von (5.2) sowie der Prämisse, dass alle Knoten identisch aufgebaut sind, erhalten wir:

$$\begin{aligned} d_{MTSP,s} &\geq d^{MIN,MB}_{MTSP,s} \\ d^{MIN,MB}_{MTSP,s} &\equiv \max(d^{P}_{maxOffset}, d_{Multiplexer} + \max_{v \in V_s}(d^{v}_{RX \to TX}), d_{Multiplexer} + d_{BTSP}) \\ &\approx \max(d^{P}_{maxOffset}, 90,7\,\mu s + d^{CC2420}_{rx \to tx}, 70\,\mu s + d_{BTSP}) \end{aligned} \tag{5.9}$$

Minimale Länge eines modusbasierten Mikroslots

Zur Bestimmung der minimalen Länge eines modusbasierten Mikroslots s muss die Anzahl von Backoffslots bekannt sein, die für die Abbildung der zugeordneten Modes (*Modus-Präferenzen*) benötigt werden. Des Weiteren gehen die konfigurierte Länge des Backoffslots d_{back}, die Konfiguration des MTSP sowie die maximale Länge für die Übertragung eines Rahmens $d_{maxTransmission}$ bei IEEE 802.15.4 ein. Um die minimale Länge des Mikroslots s für die *Imote 2*-Plattform bzw. BiPS zu bestimmen, verwenden wir Constraint 4.6 Ausdruck (4.3), wobei map_{mp} die verwendete mit mp verträgliche Abbildung der *Modus-Präferenzen* auf Backoffslots ist. Es gilt,

$$\begin{aligned} d_{Mikro,s} &\geq d^{MIN}_{Mikro,s} \\ d^{MIN}_{Mikro,s} &\equiv k_{max,s} + d_{MTSP,s} + d_{maxTransmission} + \max(d^{P}_{maxOffset} + sig\_delay^{V}_{max}, \max_{v \in V}(d^{v}_{TX \to RX})) \end{aligned} \tag{5.10}$$

Für die einzelnen Teilausdrücke aus Ausdruck (5.10) verwenden wir die Abschätzungen aus Anhang C.2 und erhalten daher für $d^{MIN}_{Mikro,s}$ für unsere Plattform

$$d^{MIN}_{Mikro,s} \approx k_{max,s} + d_{MTSP,s} + 4,256\,\text{ms} + \max(d^{P}_{maxOffset} + 120,8\,\mu s, d^{CC2420}_{tx \to rx}) \tag{5.11}$$

mit $k_{max,s} = d_{back} \cdot \max_{m \in M_s}(map_{mp}(s,m))$. Mittels $k_{max,s}$ wird die Anzahl der benötigten Backoffslots des Mikroslots s für die Abbildung der Modes (*Modus-Präferenzen*) berücksichtigt[16].

5.2.2.3 Zusammenhang zwischen modusbasierten Mikroslots und modusbasierten Slotregionen

Die vorgeschlagene Struktur aus Abbildung 5.7 (noch einmal dargestellt in Abbildung 5.8) ist zulässig, wenn wir für die Dimensionierung der Backoffslots und des MTSP die Vorgaben der beiden vorangegangenen Kapitel berücksichtigen. In diesem Fall ist die minimale Länge der modusbasierten Slotregion identisch mit der minimalen Länge eines modusbasierten Mikroslots gemäß Ausdruck (5.11). Um dies einzusehen, muss lediglich geprüft werden, ob die Vorgaben des BCS bzgl. der Zeitpunkte t_1 bzw. t_2 bei der verwendeten Strukturierung eingehalten werden, so wie dies Abbildung 5.8 suggeriert.

Zeitpunkt t_1 ist der früheste Zeitpunkt zu dem das MAC-Protokoll der Slotregion eine Übertragung starten darf. Dies wird bei der Wahl des MTSP bereits durch Ausdruck (5.9) berücksichtigt, da

$$d_{MTSP,s} \geq d^{MIN,MB}_{MTSP,s} = \max(d^{P}_{maxOffset}, d_{Multiplexer} + \max(\max_{v \in V_s}(d^{v}_{RX \to TX}), d_{BTSP})) \geq d^{P}_{maxOffset} = t_1 - t_0$$

[15] Bei den hier verwendeten $d_{Multiplexer} = 70\,\mu s$ handelt es sich um keine garantierte obere Schranke, da die Komplexität der auszuführenden Operationen des Multiplexers stark von der Struktur der angelegten Puffer abhängen. In den bisherigen Anwendungen hat sich diese Schranke jedoch bewährt.

[16] $M_s \equiv \{m \in M | SA(s,m)\ \textit{ist definiert}\}$

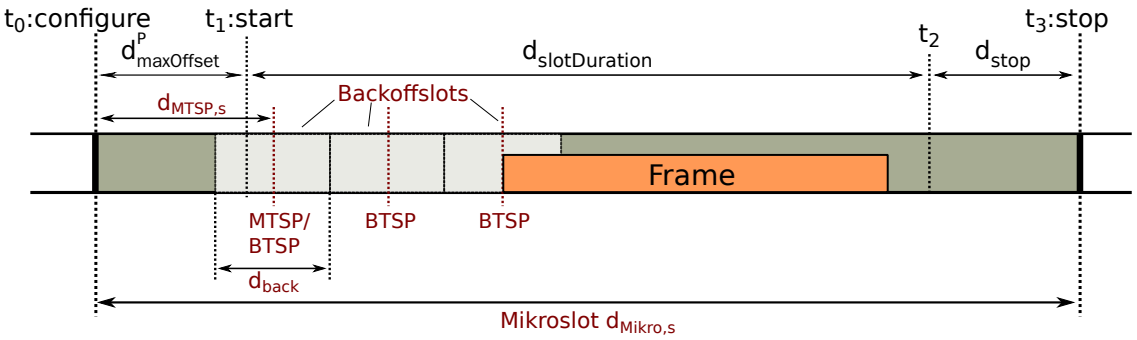


Abbildung 5.8: Aufbau einer modusbasierten Slotregion.

Des Weiteren muss zum Zeitpunkt t_2 die Übertragung abgeschlossen sein. Auch dies ist gewährleistet, wenn die Länge der modusbasierten Slotregion entsprechend Ausdruck (5.10) konfiguriert wird. Gemäß den Vorgaben muss die Übertragung in einer Slotregion spätestens d_{stop} Zeiteinheiten vor deren nominellen Ende abgeschlossen sein. Bei einem modusbasierten Mikroslot endet die Übertragung spätestens $\max(d^P_{maxOffset} + sig\_delay^V_{max}, \max_{v \in V}(d^v_{TX \to RX}))$ vor dem Ende des Mikroslots (vgl. Ausdruck (5.10) und (C.6)).

$$d_{\text{stop}} = \max(d_{maxOffset}, d^{CC2420}_{tx \to rx}) \leq \max(d^P_{maxOffset} + sig\_delay^V_{max}, \max_{v \in V}(d^v_{TX \to RX}))$$

Also ist auch diese Anforderung erfüllt und der vorgestellte Aufbau der modusbasierten Slotregion gefährdet weder die Integrität angrenzender Slotregionen noch die Funktionsweise von *Mode-Based Scheduling* und auch die Ansteuerung des Protokolls durch den BCS muss nicht angepasst werden.

5.2.2.4 Details zur Realisierung des modusbasierten MAC-Protokolls in BiPS

Die Erweiterung von BiPS um ein modusbasiertes MAC-Protokoll (MB) auf der Basis von Backoffslots lässt sich aufgrund der bereitgestellten Protokollschnittstelle (Kapitel 5.2.1) und vorhandenen Basisfunktionen (BCS, Multiplexer, TX TOs etc.) relativ einfach umsetzen. Die Konfiguration einer modusbasierten Slotregion (innerhalb der Makroslotskonfiguration) umfasst neben der Länge der Slotregion, der Länge und Anzahl der Backoffslots, auch die Position des MTSP und der BTSP. Da jede (modusbasierte) Slotregion über einen eigenen Satz frei wählbarer Konfigurationsparameter verfügt, kann diese optimal an die Anforderungen angepasst werden. Für eine modusbasierte Slotregion müssen die Parameter den Vorgaben der Ausdrücke (5.1)-(5.11) genügen, um eine zuverlässige Funktion zu gewährleisten. Fehler zur Laufzeit werden dadurch vermieden, dass das MB die Parameter beim Aufruf der `configure`-Methode (vgl. Abbildung 5.5) prüft und bei Fehlern auf die Kommunikation verzichtet, um Störungen zu verhindern.

Für jede Slotregion können individuell sowohl die Modes M^s als auch das zu verwendende Mapping $map^s_{mp} : M^s \to \mathbb{N}_0$ der Modes auf die Backoffslots festgelegt werden. Dies entspricht der Idee unabhängiger Slotregionen, die BiPS vertritt, erlaubt aber natürlich auch – wie klassisch von *Mode-Based Scheduling* vorgesehen – die Definition von Modes, welche in allen modusbasierten Slotregionen gelten. Soll eine Nachricht mittels des modusbasierten Protokolls gesendet werden, muss dem Multiplexer, neben der Nachricht selbst, lediglich der für die Übertragung zu verwendende Mode als Parameter sowie die TX TO übergeben werden. Alle weiteren Details der Kommunikation werden vor den Anwendungen der höheren Abstraktionsschichten verborgen. Hierzu gehört z. B. die Wahl des Mappings (basierend auf der Zuordnung der TX TO zu der/den Slotregion/en) sowie die eigentliche Abbildung der Modes auf Backoffslot.

Die Implementierung des MBs in BiPS verwendet einen Hardware-Timer (Protokolltimer), um zeitkritische Protokollfunktionen und Abläufe mit hoher Genauigkeit auszuführen. Im Folgenden skizzieren wir

den Ablauf und die Verwendung des Protokolltimers: Übermittelt der Multiplexer eine zu versendende Nachricht, so wird zunächst anhand des Mappings der mit dem Mode assoziierte Backoffslot bestimmt. Anschließend wird der Protokolltimer so konfiguriert, dass ein Interrupt bei Beginn dieses Backoffslots ausgelöst wird. Das MB registriert sich – unabhängig von einer vorliegenden Nachricht – am Anfang der Slotregion als Empfänger des CCA-Interrupts, um den Beginn einer Übertragung zu erkennen.

Bleibt das Medium bis zum Auslösen des Protokolltimers frei, hat der Knoten den Wettbewerb um die Slotregion gewonnen und sendet seinen Rahmen. Hierzu wird beim Ablaufen des Protokolltimers die Kalibrierung des CC2420-Transceivers gestartet und unmittelbar nach dem Absetzen des Kalibrierungsbefehls die Nachricht per SPI in den TXFIFO übertragen. Dann wird der Protokolltimer auf den BTSP des Backoffslots gesetzt. Beim Erreichen des BTSP (signalisiert durch das Auslösen des Protokolltimers) wird der `STXON`-Befehl an den CC2420-Transceiver übermittelt, welcher die eigentliche Übertragung startet. Nach der Übertragung wird der Multiplexer informiert, und der CC2420-Transceiver wechselt wieder vom Sende- in den Empfangsmodus.

Wird vor dem Beginn des Backoffslots, dem die Nachricht zugeordnet wurde, eine Übertragung mittels CCA-Interrupt detektiert, so hat ein anderer Knoten die Übertragung einer Nachricht mit einer höheren *Modus-Präferenz* gestartet. In diesem Fall wird der Multiplexer über den fehlgeschlagenen Übertragungsversuch informiert und der Protokolltimer gelöscht. Anhand des Auslösezeitpunktes des CCA-Interrupts wird der dazugehörige Backoffslot und mit Hilfe des Mappings der Mode des empfangenen Rahmens ermittelt. Der Rahmen sowie dessen Mode werden dem Multiplexer übergeben, welcher diesen in den RX TO(s) speichert und die Anwendung informiert. Wenn der Knoten in der Slotregion nicht als Sender auftritt oder für diese Instanz der Slotregion keine Nachricht vorliegt, entfallen lediglich die Abläufe für die Übertragung (Setzen des Protokolltimers, Benachrichtigung des Multiplexers).

5.3 Evaluation der Umsetzung Mode-Based Scheduling with Fast Mode-Signaling für BiPS

Für die Evaluation des modusbasierten Protokolls in BiPS haben wir ein Szenario bestehend aus vier *Imote 2*-Knoten aufgebaut. Ziel ist die Überprüfung der korrekten Funktionen sowie des Zusammenspiels zwischen den Mechanismen von BiPS und dem modusbasierten Protokoll, unter Verwendung verschiedener Konfigurationen und Synchronisationsverfahren. Hierbei werden wir die auftretenden Verzögerungen messen und diese Messergebnisse mit den theoretischen Betrachtungen aus Kapitel 5.2.2 vergleichen, um hieraus auf die Zuverlässigkeit und Korrektheit der Constraints zu schließen.

5.3.1 Beschreibung des Evaluationsszenarios

Das von uns gewählte Szenario besteht aus vier Knoten in Single-Hop-Reichweite. Hiervon tauschen jeweils drei Knoten unter Verwendung des modusbasierten Protokolls Nachrichten aus. Der vierte Knoten übernimmt bei der Synchronisation die Rolle des Masters und arbeitet zusätzlich als Sniffer. In dieser Funktion zeichnet er die gesamte Kommunikation auf und überprüft gleichzeitig, ob die empfangene Sequenz von Rahmen dem vorgegebenen Ablaufplan entspricht. Abweichungen werden als Fehler registriert und in Form einer Statistik am Ende des Testlaufs ausgegeben. Diese dient als Indikator für die Zuverlässigkeit und funktionale Korrektheit der Implementierung.

Um repräsentative Ergebnisse zu erlangen, wählen wir eine in der Praxis typische Konfiguration, bestehend aus einem Makroslot mit einer Länge von einer Sekunde, aus. Der Makroslot enthält eine modusbasierte Slotregion, welche 500 ms nach Beginn des Makroslots startet. Die Abbildung 5.9 zeigt exemplarisch die Struktur des Makroslots sowie die verwendete Topologie des Single-Hop-Netzwerkes.

In unserem Beispiel verwenden wir die drei Modes $M = \{Emergency, Safety, Regular\}$. Für jeden der Modes existiert ein Slot-Assignment für die modusbasierte Slotregion. Der Knoten 1 sendet nur Nach-

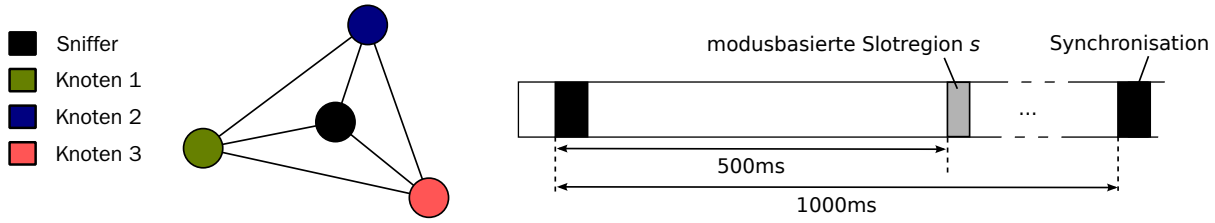


Abbildung 5.9: Darstellung der Topologie und Aufbau des Makroslots in dem Evaluationsszenario.

richten mit dem Mode *Regular*, Knoten 2 nur Nachrichten des Modes *Safety* und Knoten 3 ausschließlich Nachrichten des Modes *Emergency*. Um die korrekte Funktion des *Fast Mode-Signaling* evaluieren zu können, verwenden alle Knoten strikt periodische Kommunikationsmuster. Dies ermöglicht dem Sniffer, den geplanten Ablauf mit den empfangenen Rahmen zu vergleichen und automatisch Abweichungen zu erkennen. Das Kommunikationsmuster (Frame-Assignment) ist wie folgt aufgebaut: Knoten 1 versucht in jedem Makroslot einen Rahmen mit dem Mode *Regular* zu übertragen, während Knoten 2 nur in jedem vierten Makroslot versucht einen Rahmen mit dem Mode *Safety* zu versenden. Knoten 3 beschränkt sich auf die Übertragung eines Rahmens mit dem Mode *Emergency* in jedem achten Makroslot.

Wie bereits anhand der Wahl der Benennung der Modes offensichtlich ist, hat der Mode *Emergency* die höchste *Modus-Präferenz*, gefolgt von dem Mode *Safety*. Dem Mode *Regular* wurde die niedrigste *Modus-Präferenz* zugeordnet. Durch das Kommunikationsmuster konkurrieren in jedem vierten Makroslot zwei und in jedem achten Makroslot drei Knoten miteinander. Zusätzlich wird über diesen Ablaufplan die Funktion der TX TOs sowie des Multiplexers geprüft, da diese dafür verantwortlich sind, die jeweiligen Nachrichten rechtzeitig an das modusbasierte Protokoll zu übergeben. Die Tabelle 5.1 zeigt die definierten Modes, deren *Modus-Präferenzen* sowie die verwendete Abbildung der Modes auf Backoffslots (vgl. Definition 4.5). Wie direkt ersichtlich, ist die verwendete Abbildung map^s_{mp} verträglich mit der *Modus-Präferenz*-Funktion und optimal hinsichtlich der Zuordnung der Backoffslots.

Mode	mp	map^s_{mp}
Emergency	1	0
Safety	5	1
Regular	10	2

Tabelle 5.1: Definition der Modes, *Modus-Präferenzen* und deren Abbildung auf Backoffslots.

Die von uns gewählte Konfiguration der modusbasierten Slotregion spiegelt die Anforderungen des Szenarios wieder: Die Slotregion enthält drei Backoffslots, um die definierten Modes sowie deren *Modus-Präferenzen* abbilden zu können. Um den Einfluss des maximalen Tickoffsets auf den Overhead zu zeigen, verwenden wir drei unterschiedliche Synchronisationsmechanismen. Anhand der Vorgaben bzgl. der oberen Schranke für den maximalen Tickoffset bestimmen wir, gemäß den Vorgaben aus (5.1)-(5.11), zulässige Konfigurationsparameter für die Slotregion. Hierauf basierend evaluieren wir sowohl die Funktionalität als auch die Robustheit der Implementierung. Gleichzeitig dient dieses Vorgehen auch der Überprüfung, ob die von uns ermittelten Constraints für die Bestimmung der Konfigurationsparameter in der Praxis eine zuverlässige Funktionalität gewährleisten. Daher haben wir uns bei der Wahl der Parameter an den unteren Schranken orientiert, welche von den Constraints vorgegeben wurden.

Die Konfigurationsparameter sind in Abhängigkeit von den verschiedenen Synchronisationsverfahren in Tabelle 5.2 aufgeführt (weiß hinterlegt). Diese enthält zusätzlich auch die ermittelten unteren Grenzen für die Wahl von BTSP und MTSP sowie die hierauf beruhenden minimalen Längen der Backoffslots bzw. der gesamten modusbasierten Slotregion (hellgrau hinterlegt). Da die Auflösung der Hardware-Timer

	Sync.-Verfahren P	**BBS**	**Beaconed**	**Beaconed***	
Konfiguration	Sync.-Intervall	1000 ms	1000 ms	1000 ms	
	max. Clockskew	80 ppm	80 ppm	–	
	$d_{propMax}$	1 µs	1 µs	–	
	Netzwerkdiameter	3	1	1	
	$d^{P}_{\text{maxOffset}}$	338 µs	100 µs	15 µs	
	Umschaltzeiten:				
	$d^{CC2420}_{rx\to tx}$	128 µs	128 µs	128 µs	
	$d^{CC2420}_{tx\to rx}$	192 µs	192 µs	192 µs	
	Aufbau der Slotregion $s \in S$ (Mikroslot)				
	$d_{Mikro,s}$	6805 µs	5525 µs	5326 µs	
	$d_{MTSP,s}$	408 µs	220 µs	220 µs	
	Anzahl Backoffslots	3	3	3	
	d_{back}	841 µs	414 µs	329 µs	
	d_{BTSP}	338 µs	149 µs	149 µs	
(5.2)	$\max_{v\in V_{MB}} d^{v}_{RX\to TX}$	148,7 µs	148,7 µs	148,7 µs	**Constraints nach 5.2.2**
(5.3)	d^{MIN}_{BTSP}	338 µs	148,7 µs	148,7 µs	
(5.6)	d^{MIN}_{back}	840,4 µs	413,1 µs	328,1 µs	
(5.9)	$d^{MIN,MB}_{MTSP,s}$	408 µs	218,7 µs	218,7 µs	
(5.11)	$d^{MIN}_{Mikro,s}$	6803,6 µs	5521,7 µs	5322,9 µs	
(5.6)	d^{MIN}_{back}	840,4 µs	413,4 µs	328,4 µs	
(5.9)	$d^{MIN,MB}_{MTSP,s}$	408 µs	219 µs	219 µs	
(5.11)	$d^{MIN}_{Mikro,s}$	6804,8 µs	5524,8 µs	5326 µs	

Tabelle 5.2: Konfigurationsparameter für die modusbasierte Slotregion $s \in S$ des entworfenen Szenarios.

auf der *Imote 2*-Plattform auf volle Mikrosekunden beschränkt ist, haben wir bei der Wahl der Parameter – sofern notwendig – aufgerundet. Aufgrund des Einflusses, den die Wahl des BTSP sowie des MTSP auf die anderen Parameter haben, enthält die Tabelle, in dunkelgrau hinterlegt, auch die aus der Wahl dieser Parameter resultierenden Veränderungen der angegebenen unteren Schranken.

Wir betrachten bei unserer Evaluation drei Synchronisationsverfahren: das masterbasierte **BBS**-Verfahren und zwei Beacon-basierte Verfahren. Das masterbasierte BBS ist das von BiPS standardmäßig eingesetzte (und dem Stack namensgebende) Synchronisationsverfahren. Hierbei haben wir die Standardkonfiguration für BBS verwendet, welche BiPS vorsieht. Diese unterstützt einen maximalen Netzwerkdurchmesser von 3 Hops und garantiert einen maximalen Tickoffset von 338 µs (bei dem gewählten Synchronisationsintervall) und geht von einem maximalen Clock Skew von 80 ppm aus. Da die Schranke für den maximalen Tickoffset in dieser Konstellation sehr hoch ist und die Länge des Backoffslots sowie des MTSP und BTSP maßgeblich prägt[17], betrachten wir zusätzlich auch noch zwei alternative Synchronisationsverfahren für Single-Hop Netzwerke.

Das mit **Beaconed** bezeichnete Verfahren verwendet einfache Beacons für die Realisierung einer Ticksynchronisation, die durch einen Master periodisch versendet werden. IEEE 802.15.4 sieht für diese Form der Synchronisation ein SFD-Feld im Header jedes Rahmens vor. Beim Senden oder Empfangen des SFD-Feldes wird sofort ein SFD-Ereignis ausgelöst. Dieses erzeugt beim CC2420-Transceiver eine steigende Flanke an dessen (mit dem PXA271 verbundenen) SFD-Pin und löst einen Interrupt aus, welcher als Referenzzeitpunkt für die (Re-)Synchronisation genutzt wird. Diese Funktionalität wurde im Rahmen

[17] Zumal wir die Fähigkeit von BBS, mehrere Hops zu synchronisieren, für unser MB Protokoll sowie unser Szenario ohnehin nicht benötigen.

von [Eng13a] bereits in BiPS integriert und garantiert für unsere Konfiguration eine obere Schranke für den maximalen Tickoffset von 100 µs. Der Tickoffset resultiert aus dem maximalen Baseoffset beim Empfang des SFD-Ereignisses sowie dem Auslösen des Interrupts (20 µs). Die restlichen 80 µs sind auf den maximalen Clock Skew von 80 ppm bei dem gewählten Resynchronisationintervall von einer Sekunde zurückzuführen.

Das mit **Beaconed*** bezeichnete Verfahren ist technisch identisch mit der zuvor beschriebenen Beacon-basierten Ticksynchronisation. Allerdings wurde die obere Schranke für den maximalen Tickoffset nicht anhand theoretischer Betrachtung bestimmt, sondern mittels Messungen experimentell für den konkreten Aufbau ermittelt. Um den Tickoffset sowie den Baseoffset zu ermitteln, erzeugen die Knoten jeweils unmittelbar vor und nach erfolgter Synchronisation eine steigende Flanke an einem Pin des PXA271. Diese Zeitpunkte werden erfasst[18] und, basierend auf den beobachteten Werten, obere Schranken für den maximalen Tickoffset $d_{\text{maxOffset}}^{\text{Beaconed}^*}$ abgeschätzt. Ziel dieses Vorgehens ist es zu prüfen, wie zuverlässig die von uns ermittelten Ausdrücke und Constraints zur Bestimmung der Konfigurationsparameter sind, indem wir uns möglichst nahe an deren Minimum *herantasten* (welches maßgeblich durch den maximalen Tickoffset dominiert wird).

Die Abbildung 5.10 zeigt die Messergebnisse für die Beacon-basierte Synchronisation als Boxplot. In dieser Darstellung wird der Median durch eine dicke Linie innerhalb der Box repräsentiert. In dem durch die Box abgegrenzten Bereich liegen 50 Prozent der erfassten Messwerte. Die sogenannten Antennen geben jeweils den gemessenen minimalen und maximalen Offset an. Der beobachtete mittlere Baseoffset liegt bei 1.9 µs, der mittlere Tickoffset beträgt 7,2 µs. Während der Messung (bestehend auf 1000 Messwerten) wurde ein maximaler Tickoffset von 12,04 µs gemessen (der jedoch auf wenige Ausreißer zurückzuführen war). Aus diesem Grund haben wir bei der Beaconed*-Konfiguration den maximalen Tickoffset auf 15 µs angesetzt, sodass dieser mit hinreichend hoher Wahrscheinlichkeit nicht verletzt wird.

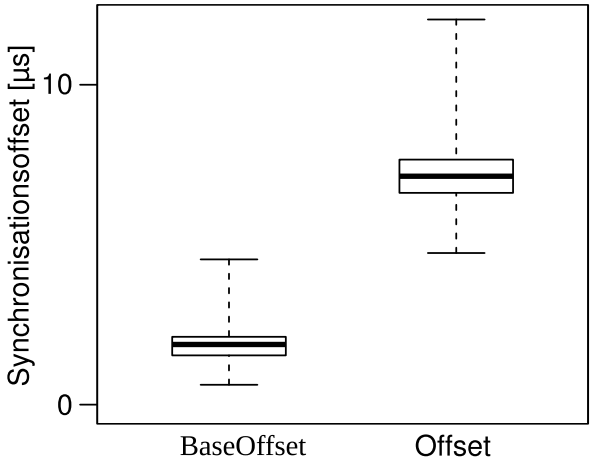


Abbildung 5.10: Messungen des Baseoffset und Tickoffset für die Beacon-basierte Ticksynchronisation.

[18] Hierfür haben wir den entsprechenden Pin jedes Knotens mit einem Logic-Analyser verbunden. Wir verwenden für die Messungen wieder unseren Logic-Analyser auf Basis eines FPGAs mit einer Abtastrate von 50 MHz aus Kapitel 3.3.

5.3.2 Ergebnisse der Evaluation

Um zu prüfen, ob unsere Implementierung korrekt arbeitet, haben wir für eine Abfolge von 5000 Makroslots die Übertragungen aller drei Knoten mit der eingeplanten Abfolge von Rahmen verglichen. Die Experimente wurden mit allen drei Konfigurationsvarianten aus Tabelle 5.2 wiederholt, mit demselben Ergebnis: Während der Messreihen traten keine Fehler auf, und auch bei mehreren Wiederholungen waren die Ergebnisse identisch. Die Messungen fanden in einer kontrollierten Umgebung statt, bei der Einflüsse durch andere Sender weitestgehend ausgeschlossen waren.

Da die Logausgaben für alle drei Konfigurationsvarianten nahezu identisch sind, beschränken wir uns auf die Darstellung der Logausgaben des Sniffers sowie ausgewählter Knoten, bei Verwendung von BBS als Synchronisationsverfahren. Das Listing 5.1 zeigt exemplarisch einen Auszug der Logmeldungen des Sniffers für eine Messreihe. Die ersten beiden Zeilen spezifizieren die Rolle des Knotens sowie den von BiPS berechneten maximalen Tickoffset für die Konfiguration. Die letzte Zeile fasst die Ergebnisse der Auswertung der Messreihe zusammen. Bei den insgesamt 5000 überwachten Makroslot, traten 0 Fehler (d.h., Abweichung von der geplanten Abfolge) auf. Im letzten Superslot (Nummer[19] 5010), der von der Auswertung berücksichtigt wurde, hat der Sniffer einen Rahmen mit dem Mode *Regular* empfangen (Zeile 8). Das ungekürzte Log führt für jede modusbasierte Slotsregion, wie in Zeile 4 bis 8 dargestellt, die empfangenen Rahmen und deren Modes auf.

Listing 5.1: Ergebnis einer Messreihe mit 5000 Superslots.

```
1 Modebased Scheduling Evaluation: Node MASTER/SNIFFER
  Offset 338
3 [...]
  MB-Slot 5006: FRAME received: Mode=Safety
5 MB-Slot 5007: FRAME received: Mode=Regular
  MB-Slot 5008: FRAME received: Mode=Regular
7 MB-Slot 5009: FRAME received: Mode=Regular
  MB-Slot 5010: FRAME received: Mode=Regular
9 Rounds 5000, Errors 0
```

Die Listings 5.2 und 5.3 illustrieren den Ablaufplan und belegen die korrekte Funktion des *Fast Mode-Signaling*. Hierzu haben wir jeweils nur ausgewählte Knoten angeschaltet und dann geprüft, ob der Sniffer wie erwartet die entsprechenden Nachrichten der jeweiligen Modes in den korrekten Intervallen empfängt. Listing 5.2 zeigt das verwendete Kommunikationsmuster der einzelnen Knoten, während Listing 5.3 für verschiedene Konstellationen von Knoten zeigt, dass sich stets die Nachricht mit der höchsten *Modus-Präferenz* durchsetzt.

[19] Der Sniffer nummeriert die Superslots durchlaufend, beginnt mit der Überwachung des Ablaufplans aber erst, nachdem alle drei Knoten einmal einen Rahmen mit dem jeweiligen Mode übertragen haben. Dies stellt sicher, dass alle Knoten synchronisiert und bereit sind, bevor die Auswertung der Messreihe beginnt.

Listing 5.2: Funktionsweise von *Mode-Based Scheduling with Fast Mode-Signaling* bei unterschiedlichen Knotenkonstellationen (1).

```
1  1 Knoten: Mode Regular
   Modebased Scheduling Evaluation: Node MASTER/SNIFFER
3  [..]
   MB-Slot 10: FRAME received: Mode=Regular
5  MB-Slot 11: FRAME received: Mode=Regular
   MB-Slot 12: FRAME received: Mode=Regular
7
   1 Knoten: Mode Safety
9  Modebased Scheduling Evaluation: Node MASTER/SNIFFER
   [..]
11 MB-Slot 10: FRAME received: Mode=Safety
   MB-Slot 14: FRAME received: Mode=Safety
13 MB-Slot 18: FRAME received: Mode=Safety

15 1 Knoten: Mode Emergency
   Modebased Scheduling Evaluation: Node MASTER/SNIFFER
17 [..]
   MB-Slot 10: FRAME received: Mode=Emergency
19 MB-Slot 18: FRAME received: Mode=Emergency
   MB-Slot 26: FRAME received: Mode=Emergency
```

Listing 5.3: Funktionsweise von *Mode-Based Scheduling with Fast Mode-Signaling* bei unterschiedlichen Knotenkonstellationen (2).

```
   2 Knoten: Modes Regular, Safety
2  Modebased Scheduling Evaluation: Node MASTER/SNIFFER
   [..]
4  MB-Slot 5: FRAME received: Mode=Regular
   MB-Slot 6: FRAME received: Mode=Safety
6  MB-Slot 7: FRAME received: Mode=Regular
   MB-Slot 8: FRAME received: Mode=Regular
8  MB-Slot 9: FRAME received: Mode=Regular
   MB-Slot 10: FRAME received: Mode=Safety
10 MB-Slot 11: FRAME received: Mode=Regular

12 3 Knoten: Modes Regular, Safety, Emergency
   Modebased Scheduling Evaluation: Node MASTER/SNIFFER
14 [..]
   MB-Slot 5: FRAME received: Mode=Regular
16 MB-Slot 6: FRAME received: Mode=Safety
   MB-Slot 7: FRAME received: Mode=Regular
18 MB-Slot 8: FRAME received: Mode=Regular
   MB-Slot 9: FRAME received: Mode=Regular
20 MB-Slot 10: FRAME received: Mode=Emergency
   MB-Slot 11: FRAME received: Mode=Regular
22 MB-Slot 12: FRAME received: Mode=Regular
   MB-Slot 13: FRAME received: Mode=Regular
24 MB-Slot 14: FRAME received: Mode=Safety
   MB-Slot 15: FRAME received: Mode=Regular
26 MB-Slot 16: FRAME received: Mode=Regular
   MB-Slot 17: FRAME received: Mode=Regular
28 MB-Slot 18: FRAME received: Mode=Emergency
   MB-Slot 19: FRAME received: Mode=Regular
```

Listing 5.4 zeigt den internen Ablauf bei der Übertragung einer Nachricht anhand der Logausgaben des Knotens 2. Dieser versendet Nachrichten des Modes *Safety*. Dargestellt sind zwei Abläufe, einmal sendet der Knoten die Nachricht mit der höchsten *Modus-Präferenz* und kann sich durchsetzen (Zeile 1-5) und einmal sendet der Knoten 3 einen Rahmen mit einer höheren *Modus-Präferenz* und Knoten 2 muss seinen Übertragungsversuch einstellen (Zeile 7-12). In beiden Fällen wird zunächst zu Beginn der Slotregion die Methode `configure` des Protokolls aufgerufen. Hierauf folgt der Aufruf von `setData` und die Übergabe des zu übertragenden Rahmens durch den Multiplexer. Die Übertragung wird aufgrund der *Modus-Präferenz* für den zweiten Backoffslot (mit der Nummer 1) eingeplant – Zeilen 3 und 9. Beim Start des zugeordneten Backoffslots wird der Transceiver kalibriert und die Nachricht in den TXFIFO geladen. In Zeile 4 bestätigt ein Callback die erfolgreiche Übermittlung der Nachricht in den TXFIFO; beim Erreichen des BTSP erfolgt der Versand des Rahmens (Zeile 5). Ab Zeile 7 ist der Ablauf dargestellt, der sich ergibt, wenn innerhalb der modusbasierten Slotregion ein anderer Knoten mit einer höheren *Modus-Präferenz* einen Rahmen sendet. Zwar plant Knoten 2 wieder eine Übertragung ein (Zeilen 8 und 9), jedoch wird bereits in einem früheren Backoffslot (hier Backoffslot 0) eine Medienbelegung erkannt (Zeile 10). Hierdurch wird die eingeplante Übertragung abgebrochen. Nach dem Empfang dieses Rahmens wird er an den Multiplexer weitergereicht (Zeile 11). Die eigene für die Slotregion eingeplante Nachricht wird nicht übertragen. Dies wird ebenfalls dem Multiplexer mitgeteilt. Aus Effizienzgründen erfolgt die Benachrichtigung des Multiplexers innerhalb des Aufrufs der Methode `stop` (Zeile 12).

Listing 5.4: Logausgaben des Knotens 2, bei der Übertragung von Nachrichten mit dem Mode *Safety*.

```
1  MB: configure (MB-Slot 59)
   MB_setData: set frame with mode SAFETY
3  MB_EndofBackoffSlot: Scheduling transmission start for backoffslot 1
   MB_CC2420_TXReadyCallback: Fifo loaded
5  MB_TIMER_BTSP: Transmission performed

7  MB: configure (MB-Slot 63)
   MB_setData: set frame with mode SAFETY
9  MB_EndofBackoffSlot: Scheduling transmission start for backoffslot 1
   MB_CC2420_setRXCCAFECallback: Medium occupied at 86113739-> slotregion already in use
11 MB_CC2420_RxCallback: Frame received at 86114325!
   MB_stop: Transmission aborted due frame with higher mode preference
```

5.3.3 Diskussion der Ergebnisse

Wie die Messreihen gezeigt haben, ermöglicht unsere Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in BiPS für alle drei untersuchten Konfigurationen eine zuverlässige Kommunikation, und dies, obwohl die Konfigurationsparameter so gewählt wurden, dass diese sich sehr nahe an den zulässigen Minimalwerten bewegen, welche die Constraints vorgeben. Für die Praxis ist dies ein guter Indikator dafür, dass die Constraints tatsächlich die Erstellung korrekter und zuverlässiger Konfigurationen ermöglichen. Andererseits ist dies aufgrund der Herleitung auch nicht allzu überraschend, da die Constraints auf Worst-Case Betrachtungen beruhen. So wird der maximale Tickoffset bei der Dimensionierung der Backoffslot sowie des BTSP und MTSP berücksichtigt. Dieser basiert selbst wiederum auf einer Worst-Case Abschätzung für das jeweilige Verfahren. Zur Laufzeit ist die mittlere Abweichung der Knoten im Allgemeinen deutlich geringer und der Worst-Case selbst, wenn überhaupt, nur selten beobachtbar. Dies gilt ebenso für die berücksichtigte maximale Signalverzögerung oder Umschaltzeiten sowie die maximale CCA-Verzögerung des CC2420-Transceivers. Auch hier wurden jeweils Worst-Case Abschätzungen verwendet. Die Wahrscheinlichkeit, dass der Worst-Case für jede dieser Einflussgrößen gleichzeitig in einem Szenario auftritt, ist sehr gering. Die Größen, welche wir nur abschätzen konnten, wie die Verzögerung für die Verarbeitung der Interrupts oder den Zugriff auf den SPI-Bus sowie die Verarbeitungszeiten des

Multiplexers, wurden anhand von Messungen ermittelt und mit einem zusätzlichen *Sicherheitspuffer* versehen. Dass dieser Ansatz gut funktioniert, belegen die Messreihen für die Konfiguration Beaconed*. Hier wurde anstelle einer theoretischen oberen Schranke für den maximalen Tickoffset ein auf Messungen basierender Wert verwendet, der als Ausgangsbasis für die Ermittlung der Konfigurationsparameter diente. Auch bei dieser Konfiguration traten bei den Messreihen keine Fehler auf.

Hinsichtlich der Beurteilung der Zuverlässigkeit des modusbasierten Protokolls in BiPS auf Basis der in den Experimenten ermittelten Fehlerrate muss an dieser Stelle betont werden, dass die Messungen unter Laborbedingungen in einer kontrollierten Umgebung stattfanden. Das bedeutet insbesondere, dass Einflüsse durch andere Sender weitestgehend ausgeschlossen waren. Zudem war die räumliche Ausdehnung des Netzwerkes klein, sodass der Empfang mit hohen RSSI-Werten (Received Signal Strength Indication) erfolgte. Dies führt in der Folge zu einer sehr schnellen Erkennung der Medienbelegung, da der verwendete CCA-Mechanismus die Empfangsstärken über 8 Symbolperioden (128 µs) integriert und bei Überschreiten des Schwellenwertes das Medium als belegt erkennt. Um die Auswirkungen des Versuchsaufbaus besser beurteilten zu können, haben wir die Verzögerung des gesamten Signalweges gemessen: Der Knoten 3 generiert beim Erreichen seines BTSP (beim Start der Übertragung) eine steigende Flanke an einem externen Pin, das Gleiche machen die anderen Knoten beim Erkennen der Medienbelegung. Die Zeitpunkte wurden wieder mit einem Logic-Analyser aufgezeichnet und ausgewertet. Abbildung 5.11 zeigt diese Verzögerung bei den verschiedenen Empfangsknoten. Auffällig ist der Unterschied zwischen der mittleren Erkennungsverzögerung des Knotens 2 und den beiden anderen Knoten. Diese kann sowohl mit der Position des Knotens (Ausrichtung der Antenne, Abstand, Reflektionen) als auch unterschiedlichen Charakteristika hinsichtlich der Alterung der verbauten Komponenten zusammenhängen. Insgesamt bleibt die maximale Verzögerung bei der Erkennung jedoch unter 80 µs und liegt im Mittel, abhängig vom Knoten, deutlich darunter. Unsere Constraints berücksichtigten (und erlauben) eine maximale Erkennungsverzögerung von $149,4\,\mu s$, welche sich aus der Verzögerung für das Absetzen des Befehls zum Senden, der Signallaufzeit sowie der Verarbeitungsverzögerung für den CCA-Interrupt und den CCA-Mechanismus (maximal 128 µs) selbst zusammensetzt[20] (vgl. Ausdrücke (5.4) sowie (C.2)-(C.5)).

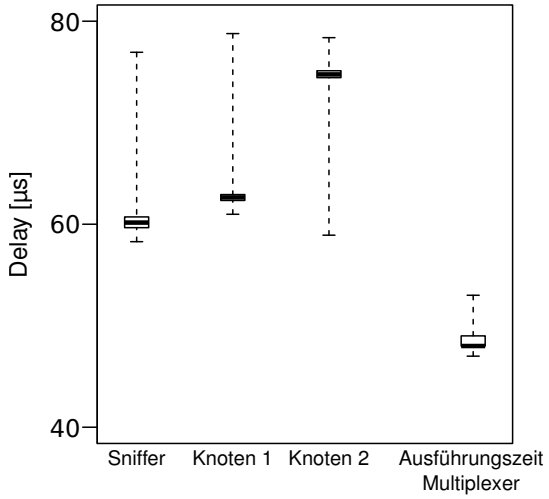


Abbildung 5.11: Verzögerung bei der Erkennung der Mediumbelegung sowie der Übergabe von Daten vom Multiplexer zum modusbasierten Protokoll.

[20] $d_{STXON} + d_{maxPropagationDelay} + d_{CCA-Interrupt} + d_{maxCCA} = 149,4\,\mu s$

Eine weitere Größe, welche wir lediglich abgeschätzt haben, ist die Ausführungszeit des Multiplexers ($d_{Multiplexer} = 70\,\mu s$), die in die Konfiguration des MTSP eingeht (vgl. Ausdruck (5.9)). Auch hier haben wir in unserem Testszenario Messungen vorgenommen. Dazu wurde die Ausführungszeit des Multiplexers lokal auf Knoten 3 gemessen. Die Ergebnisse sind ebenfalls in der Abbildung 5.11 dargestellt. Die mittlere gemessene Ausführungszeit betrug 48 µs (Maximum 53 µs, Auflösung 1 µs), sodass die verwendeten 70 µs auch komplexere Konfigurationen der TX TOs abdecken sollten.

Insgesamt lassen die Constraints aufgrund der Worst-Case Annahmen ausreichend Spielraum, sodass sich aus der Verwendung der unteren Schranken keine Einschränkungen hinsichtlich der Zuverlässigkeit ergeben sollten. Schwerwiegender im Feldeinsatz dürfte sein, dass die Single-Network-Assumption dort unter Umständen verletzt sein kann. Dies kann sich als problematisch erweisen, wenn die CCA-Erkennung durch andere Übertragungen ausgelöst wird bzw. das Medium gerade dann belegt ist, wenn ein Knoten seinen BTSP erreicht und die Übertragung eines eigenen Rahmens startet. In diesem Fall ist nicht sichergestellt, dass die anderen Knoten den Beginn der eigentlichen Übertragung erkennen bzw. diese dem korrekten Backoffslot zuordnen können. Hierbei handelt es sich jedoch um ein inhärentes Problem von drahtlosen Netzwerken, mit denen auch BBS oder ACTP kämpfen. Bei Einsatz des modusbasierten Protokolls muss daher vor Ort sichergestellt werden, dass der verwendete Kanal hinreichend frei von Störungen ist, um einen zuverlässigen Betrieb zu ermöglichen. In Bezug auf weitere Verbesserungen, für eine möglichst robuste Erkennung von (relevanten) Übertragungen und Medienbelegungen, sind noch zusätzliche Feldversuche notwendig.

Abschließend bleibt festzustellen, dass eine Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* mittels Backoffslots auch in drahtlosen Netzwerken möglich ist, wie unsere Integration des modusbasierten Protokolls in BiPS beweist. Jedoch ist der verwendete CC2420-Transceiver aufgrund seiner langen Umschaltzeiten sowie der hohen maximalen CCA-Verzögerungen hinsichtlich des Overheads nicht optimal. Dieser resultiert im Wesentlichen aus der Dauer der Backoffslots, da jede abzubildende *Modus-Präferenz* einen eigenen Backoffslot benötigt. Die Abbildung 5.12 zeigt den Zusammenhang zwischen den minimal zulässigen Werten für BTSP, MTSP sowie der minimal zulässigen Größe eines Backoffslots und dem maximalen Tickoffset $d_{maxOffset}$. Gerade in Bezug auf die Wahl des BTSP und MTSP wirken die technischen Kenngrößen des CC2420-Transceivers auch im Fall einer perfekten Synchronisation einschränkend. Erst bei einem maximalen Tickoffset von über 149 µs wirken diese Kenngrößen der verwendeten Hardwareplattform in Bezug auf die Backoffslotgröße nicht mehr dominierend.

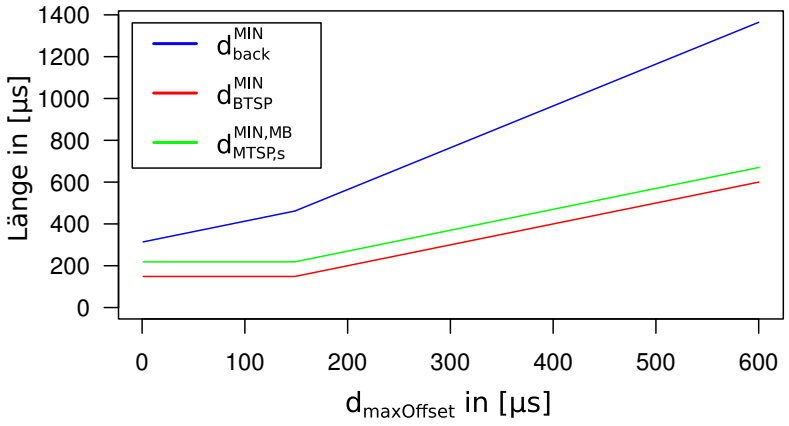


Abbildung 5.12: Einfluss des maximalen Tickoffset auf die minimale Länge von BTSP, MTSP und Backoffslots des modusbasierten MAC-Protokolls.

5.4 Mode-Based Scheduling with Fast Mode-Signaling in drahtlosen Multi-Hop-Netzwerken

Die von uns vorgestellte Implementierung von *Mode-Based Scheduling with Fast Mode-Signaling* unterstützt lediglich Single-Hop-Netzwerke, ermöglicht in diesen jedoch eine effiziente Umsetzung, ohne spezielle Anforderungen an die Codierung oder die verwendeten Transceiver zu stellen. Obwohl der Schwerpunkt dieser Arbeit auf drahtgebundenen Single-Hop-Netzwerken liegt, wollen wir an dieser Stelle dennoch kurz auf einen alternativen Ansatz für die Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* in drahtlosen Multi-Hop-Netzwerken eingehen.

Wir gehen von der Annahme aus, dass eine Übertragung einer modusbasierten Nachricht von allen Knoten des Netzwerks empfangen werden soll. Für die Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* sind daher in einem Multi-Hop-Netzwerk zwei grundlegende Probleme zu lösen: Die Ermittlung sowie Propagierung des Modes mit der höchsten Präferenz durch das gesamte Netzwerk (*Fast Mode-Signaling*) sowie die Übertragung des zu diesem Mode gehörenden Rahmens an alle Knoten (Routing).

Eine Variante für die Umsetzung von *Fast Mode-Signaling* in Multi-Hop-Netzwerken besteht in der Verwendung von ACTP (Arbitrating & Cooperative Transfer Protocol[21], [CGR12, CGSW14, Chr15]) für die Ermittlung und Propagierung des Modes mit der höchsten Präferenz. ACTP ist ein *Binary Countdown*-Protokoll [PAT07] für drahtlose Single- und Multi-Hop-Netzwerke, wobei wir uns im Folgenden zunächst auf die Abläufe in Single-Hop-Netzwerken konzentrieren. ACTP verfolgt die gleiche Idee wie CAN: Vor die Übertragung der Nutzdaten setzt ACTP einen Arbitrierungsprozess, bei dem jeder Knoten eine eindeutige Bit-Sequenz fester Länge (Identifier) überträgt, welche die Priorität der Nachricht repräsentiert. Die Codierung der einzelnen Bits des Identifiers erfolgt mittels Black Bursts (BBs) (eine logische Eins wird durch die Übertragung eines dominanten Signals codiert, bei einer logischen Null wird kein Signal gesendet). Dominante Signale selbst sind kollisionsresistent, da es sich lediglich um Energie handelt, die detektiert wird. Die Arbitrierung erfolgt, wie bei CAN, bitweise und basiert darauf, dass auf dem Medium eine logische Eins (dominant) eine logische Null (rezessiv) überlagert und sich die Eins durchsetzt. Jeder Knoten sendet seinen Identifier, beginnend mit dem höchstwertigen Bit (MSB). Ein Knoten bricht die Übertragung seines Identifiers ab, sobald er ein dominantes Bit empfängt, sein Identifier an dieser Stelle jedoch ein rezessives Bit enthält. Es gewinnt der Knoten, der seinen Identifier vollständig übertragen konnte, d.h. der Knoten mit dem nummerisch größten Identifier. Im Anschluss sendet dieser Knoten seinen Rahmen auf konventionelle Weise (vgl. [CGR12]). Besteht ein Identifier aus n_{bits} Bits, so ist die gesamte Arbitrierungsphase insgesamt $d_{\text{arbitrationPhase,sh}} = d_{\text{bitPhase,sh}} \cdot n_{\text{bits}}$ lang, wobei $d_{\text{bitPhase,sh}}$ die Dauer für die Übermittlung eines Bits bezeichnet.

Es ist offensichtlich, dass für die Dimensionierung der Bit-Phasen die gleichen Größen wie in Bezug auf die Backoffslots zu berücksichtigen sind, d.h. unter anderem der maximale Tickoffset, Umschaltzeiten des Transceivers sowie die maximale Verzögerung für die CCA-Erkennung. Zusätzlich muss sichergestellt werden, dass die Bit-Phasen ausreichend lang sind, damit ein Knoten nach einer Übertragung und vor dem Beginn der nächsten Bit-Phase wieder vom Sende- in den Empfangsmodus wechseln kann. Die Bit-Phasen enthalten eine zusätzliche Zeitspanne $d_{pause} = 16\,\mu\text{s}$ (eine Symbolperiode beim CC2420-Transceiver), damit die Empfänger auch bei Überlagerungen von BBs durch mehrere Sender die einzelnen Bits des Identifiers sicher zuordnen können [CGSW14]. Beim CC2420-Transceiver wird ein BB durch die Übertragung eines 5 Byte langen Rahmens implementiert.

Eine Realisierung von *Fast Mode-Signaling* mittels ACTP lässt sich sehr einfach über eine geeignete Abbildung erreichen, welche die *Modus-Präferenzen* auf Identifier abbildet. Hierbei handelt es sich (wie bei CAN) um eine Form des aktiven *Fast Mode-Signaling*. Nach erfolgter Arbitrierung wird dann der Rahmen konventionell übertragen. Da die in ACTP verwendete Codierung der Identifier pro Bit-Phase

[21] Die Publikationen verwenden zum Teil noch die frühere Bezeichnung AVTP – Arbitrating Value Transfer Protocol.

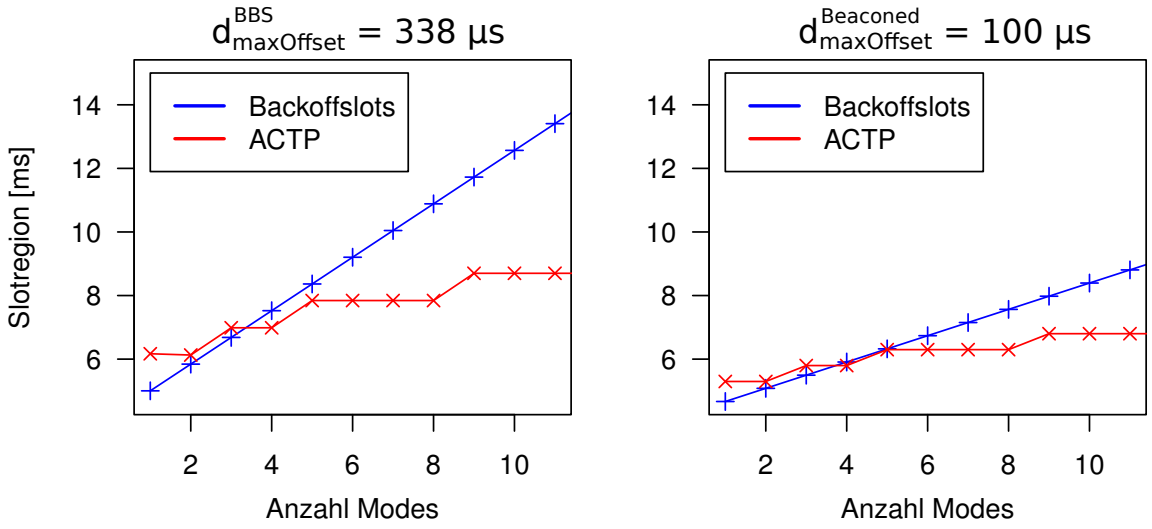


Abbildung 5.13: Länge der minimalen modusbasierten Slotregion in Abhängigkeit von der Anzahl zugeordneter Modes.

zwei Prioritäten abbilden kann, Backoffslots jedoch nur eine Priorität, stellt sich die Frage, welche der beiden Realisierungen von *Mode-Based Scheduling with Fast Mode-Signaling* in Single-Hop-Netzwerken effizienter arbeitet. Daher haben wir für beide Verfahren die minimale Länge einer Slotregion für die Übertragung eines Rahmens mit der maximalen Nutzlast berechnet. Hierfür kamen bei der Backoffslot-basierten Variante die Constraints aus Kapitel 5.2 zum Einsatz. Für die konzeptionelle Umsetzung mittels ACTP kamen die Constraints für die Konfiguration der Bit-Phasen aus [CGR12] zum Einsatz[22].

Abbildung 5.13 zeigt die minimale Länge der modusbasierten Slotregion in Abhängigkeit von der Anzahl zugeordneter Modes für beide Realisierungsvarianten. Dargestellt ist die Situation einmal bei Verwendung von BBS mit einem angenommenen maximalen Tickoffset von 338 µs (links) sowie bei Nutzung einer Beacon-basierten Ticksynchronisation mit einem angenommenen maximalen Tickoffset von 100 µs (vgl. Tabelle 5.2). Wie direkt aus der Grafik ersichtlich, ist der Overhead des ACTP-Verfahrens geringer, je größer die maximale Anzahl der Modes ist, die einer Slotregion zugeordnet werden (darf). Dies liegt an der binären Kodierung für die Identifier. So erlaubt ein Identifier mit n-Bits die Repräsentation von 2^n Modes (*Modus-Präferenzen*) für eine Slotregion. ACTP benötigt für das *Fast Mode-Signaling* in diesem Fall lediglich n Bit-Phasen, während die auf Backoffslots-basierte Lösung hierfür 2^n Backoffslots benötigt. Dies wiegt den Vorteil, dass die Backoffslots kürzer dimensioniert werden können als die Bit-Phasen, ab einer bestimmten Anzahl zugeordneter Modes wieder auf. Bei unserer Implementierung für die *Imote 2*-Plattform ist die Variante mit Backoffslots in den beiden betrachteten Szenarien bei bis zu drei zugeordneten Modes leicht im Vorteil. Ab vier Modes hingegen ist ACTP effizienter und der Abstand wächst weiter, mit zunehmender Anzahl zugeordneter Modes.

Die eigentliche Stärke von ACTP liegt darin, dass die Arbitrierung nicht nur in Single-Hop-Netzwerken, sondern auch in Multi-Hop-Netzwerken funktioniert. Somit lässt sich mit ACTP *Fast Mode-Signaling* auch zuverlässig in Multi-Hop-Netzwerken realisieren. Hierbei kommt ACTP lediglich für das *Fast Mode-Signaling* zum Einsatz, die Übertragung der Nutzdaten erfolgt konventionell und nutzt klassische Routing-Verfahren. Das ACTP-Arbitrierungsverfahren für Multi-Hop-Netzwerke wurde von Christmann et al. in [CGR12] vorgestellt. Auch hier bestehen die Identifier aus einer Bit-Sequenz mit einer festen Länge von n_{bits}. Zusätzlich muss der maximale Netzwerkdurchmesser (oder eine entsprechende obere

[22] Die Constraints wurden um die gleichen implementierungsspezifischen Verarbeitungsverzögerungen ergänzt, die auch bei der Variante auf Basis der Backoffslots berücksichtigt werden.

Schranke) n_{hops} des Netzwerkes bekannt sein. Jede Arbitrierungsphase besteht im Multi-Hop-Fall aus n_{bits} Bit-Phasen, welche ihrerseits wieder in n_{hops} Bit-Runden unterteilt sind. Bei ACTP unterscheidet man im Multi-Hop-Fall zwischen aktiven Knoten und Repeatern. Zu Beginn der Arbitierungsphase sind alle Knoten, die einen Rahmen übertragen wollen, aktive Knoten, alle anderen Knoten Repeater.

Die Arbitrierung erfolgt wieder bitweise, beginnend mit dem MSB[23] des Identifiers. Jede Bit-Phase propagiert ein Bit des Identifiers über einen Radius von n_{hops}. In der Bit-Phase i agiert jeder Knoten gemäß den folgenden Regeln: Jeder aktive Knoten sendet in der ersten Bit-Runde der i-ten Bit-Phase das i-te Bit seines Identifiers. Falls es sich bei dem Bit des Identifiers um ein rezessives Bit handelt (Null), und er empfängt in dieser oder einer der nachfolgenden Bit-Runden ein dominantes Bit (Eins), wird der Knoten für den Rest der Arbitierungsphase zum Repeater, da er die Arbitrierung verloren hat. Wird ein Knoten zum Repeater bzw. empfängt ein Repeater in einer Bit-Runde ein dominantes Bit, so wiederholt er dieses in der darauf folgenden Bit-Runde und bleibt dann für die restliche Bit-Phase inaktiv. Es gewinnt der Knoten die Arbitrierung, der seinen kompletten Identifier senden konnte und am Ende der Arbitrierungsphase kein Repeater ist. Beträgt die Dauer einer Bit-Runde $d_{\text{bitRound,mh}}$, so ist die gesamte Arbitrierungsphase insgesamt $d_{\text{arbitrationPhase,mh}} = d_{bitPhase,mh} \cdot n_{\text{bits}}$ mit $d_{bitPhase,mh} = d_{\text{bitRound,mh}} \cdot n_{\text{hops}}$ lang.

5.5 Entwicklung von Anwendungen für BiPS

Die Entwicklung zeitkritischer MAC-Protokolle sowie der Kernkomponenten von BiPS (Multiplexer, Scheduler, Hardwaretreiber etc.) erfordert die Berücksichtigungen der Charakteristika der Hardwareplattform (spezifische Verzögerungen, Ausführungszeiten) sowie der limitierten Ressourcen der Plattform. Dies ist nur durch eine manuelle Implementierung möglich. Der Aufwand für die Entwicklung und Tests des optimierten Codes ist entsprechend hoch, da das Abstraktionsniveau sehr niedrig ist und detaillierte Kenntnisse der Plattform notwendig sind (direkter Zugriff auf Register und Speicher, Ansteuerung von Peripherie, Arbeitsweise von Caches usw.).

Anwendungen (Schicht 4, vgl. Abbildung 5.2) und Kommunikationsprotokolle auf höheren Abstraktionsstufen (HLPs, Schicht 3) haben in der Regel keine so strikten Echtzeitanforderungen wie die MAC-Protokolle, daher ist hier eine manuelle Implementierung und Optimierung nicht notwendig. Stattdessen steht häufig eine komplexe Logik im Mittelpunkt der Anwendung, und Aspekte wie Wiederverwendbarkeit, Wartbarkeit und Portierbarkeit spielen eine wesentlich wichtigere Rolle. Aufgrund der formulierten Ziele sind diese Softwarekomponenten ideale Kandidaten für klassische modellgetriebene Entwicklungsansätze.

Wir haben daher in [BCGM14] untersucht, inwiefern der modellgetriebene Entwicklungsansatz SDL-MDD (SDL-Model-Driven Development, [KGW06]), welcher ITU's Specification and Description Language (SDL, [Int12, Int00]) für die Spezifikation von Modellen verwendet, genutzt werden kann, um Anwendungen und HLPs für BiPS zu entwickeln. SDL ermöglicht die Spezifikation des Verhaltens mit Hilfe erweiterter endlicher Automaten und definiert hierfür eine grafische Beschreibungssprache. Die formale Syntax und Semantik von SDL [Int00] erlauben es, die entwickelten Verhaltensmodelle automatisch, mit Hilfe von Codegeneratoren, in ausführbaren Code zu transformieren. Kommerzielle Werkzeuge, wie IBM Rational SDL Suite [IBMar], Cinderella SDL [Cinar] und PragmaDev's Real-time Developer Studio (RTDS, [Praar]), unterstützen den Entwickler bei der Spezifikation der Systeme durch Editoren und stellen entsprechende Codegeneratoren inklusive Vorlagen für die notwendige Ausführungsumgebung zur Verfügung. Die Vorteile modellgetriebener Ansätze liegen auf der Hand: Neben einem besseren Abstraktionsvermögen, einer verbesserten Portierbarkeit und einer besseren Wartbarkeit liefern die Codegeneratoren im Allgemeinen qualitativ hochwertigen und fehlerfreien Code. Der generierte Code nutzt i.d.R. jedoch nicht die speziellen Eigenarten der jeweiligen Hardwareplattform aus und besitzt daher noch Optimierungspotential, im Vergleich zu einer manuellen Implementierung durch einen Experten.

[23] Most significant bit

Wir haben uns für PragmaDev's RTDS entschieden und die mitgelieferte Laufzeitumgebung [Pra13] so modifiziert, dass der aus der SDL Spezifikation generierte Code direkt als Anwendung bzw. HLP in BiPS eingebunden werden kann. Der generierte Code wird durch den BAS ausgeführt und hat eine niedrigere Priorität als die MAC-Protokolle sowie die anderen zeitkritischen Komponenten von BiPS. Hierdurch wird sichergestellt, dass deren Ausführung für zeitkritische Operationen unterbrochen werden kann, sodass es bei diesen nicht zu Verzögerungen kommt. Um die Interaktion mit BiPS zu ermöglichen, haben wir ein erweiterbares Environment Framework entwickelt, welches es ermöglicht, aus einem SDL-System heraus mittels SDL-Signalen auf die bereitgestellten Dienste von BiPS zuzugreifen. Hierzu gehört der direkte Zugriff auf die Hardware-Treiber von BiPS, ebenso wie auf den Multiplexer, sodass die in BiPS integrierten MAC-Protokolle aus SDL-Systemen heraus für die Kommunikation genutzt werden können. Weitere Details bezüglich der Implementierung sowie der Evaluation dieses Ansatzes sind in unserem Paper [11] zu finden.

5.6 Abgrenzung zu WirelessHart und ISA 100.11a

Im Bereich der drahtlosen Kommunikation für Industrie- und Produktionsanlagen konkurrieren aktuell zwei Standards um die Vorherrschaft: WirelessHart und ISA 100.11a. Beide Standards beanspruchen jeweils für sich, eine optimale drahtlose Kommunikationslösung für industrielle Sensornetze zur Überwachung und Steuerung von Industrie- und Produktionsanlagen zu liefern.

Sowohl WirelessHart als auch ISA 100.11a definieren einen vollständigen Kommunikationsstack. Dieser umfasst Physical-, Data-Link-, Network- sowie Transport- und Application-Layer. Im Kern ähneln sich beide Technologien sehr [PC11, Mar12]. Beide Physical-Layer basieren auf IEEE 802.15.4 und verwenden für die Übertragung eine Kombination von Direct Sequence Spread Spectrum (DSSS) im 2.4 GHz-Band und Frequenz-Hopping (frequency-hopping spread spectrum, FHSS) mit einer Datenrate von 250 kBit/s. Als Medienzugriffsverfahren kommt bei beiden TDMA zum Einsatz, um deterministisches Verhalten sowie Garantien bzgl. der verfügbaren Bandbreite sowie der auftretenden Verzögerungen realisieren zu können.

Da BiPS zur Zeit noch als reiner Protokollstack auf MAC-Ebene ausgelegt ist und die Integration höherer Protokollfunktionalitäten, wie z. B. einer dienstorientierten Middleware [Kra14] oder Routingfunktionilitäten, Gegenstand anderer Forschungsarbeiten sind, beschränkt sich diese Arbeit auf den Vergleich von BiPS (insbesondere *Mode-Based Scheduling with Fast Mode-Signaling* für BiPS) mit den Funktionalitäten der MAC-Protokolle von WirelessHart und ISA 100.11a.

5.6.1 WirelessHart

Der WirelessHart Standard [HAR07, Mar12] wurde 2007 von der HCF[24] veröffentlicht. In Bezug auf den Data-Link-Layer basiert WirelessHart – wie auch BiPS – auf TDMA. Hierzu unterteilt WirelessHart die Zeit in sogenannte Superframes, die periodisch wiederholt werden. Diese setzen sich aus einer Sequenz von Time Slots mit einer festen Länge von 10 ms zusammen.

WirelessHart verwendet slotbasiertes Frequenz-Hopping, d.h. nach jedem Time Slot wird die Frequenz (Kanal) nach einem vorgegebenen Schema gewechselt. Der Wechsel der Kanäle bei jedem Time Slot reduziert die Anfälligkeit von WirelessHart für Störungen durch andere Übertragungen bzw. Netze. Über ein Blacklisting von einzelnen Kanälen kann der Network Manager darüber hinaus gezielt Kanäle mit vielen Störungen (zur Laufzeit) von der weiteren Verwendung ausschließen. Das Frequenz-Hopping ermöglicht es, mehrere Superframes parallel zu verwenden, solange sichergestellt wird, dass deren Time Slots jeweils unterschiedliche Kanäle verwenden. Durch diese als FDMA (*Frequency-Division Multiple Access*) bezeichnete Technik lässt sich die Übertragungsrate entsprechend erhöhen, da Rahmen

[24] HCF – Highway Addressable Remote Transducer (HART) Communication Foundation

gleichzeitig auf mehreren unterschiedlichen Kanälen gesendet werden können. Bei der Planung muss jedoch berücksichtigt werden, dass die Endgeräte im Allgemeinen zu einem Zeitpunkt (Time Slot) nur an der Kommunikation auf einem Kanal partizipieren [Mar12, PC11] können.

Die Zuteilung der einzelnen Time Slots erfolgt durch den zentralen Network Manager, welcher Routing Informationen sowie Kommunikationsanforderungen (maximale Verzögerungen etc.) nutzt, um den verschiedenen Knoten Sende- und Empfangsslots innerhalb der Superframes zuzuweisen. Um Kollisionen auszuschließen und einen deterministischen Ablauf zu gewährleisten, werden Time Slots exklusiv, d.h. maximal einem Sender zugeordnet. Der zentralistische Kontrollansatz durch den Network Manager gestattet es, die Zuordnung der Time Slots zur Laufzeit zu modifizieren und an die Anforderungen einer veränderten Topologie oder Kommunikationssituation anzupassen. Hierfür muss der Network Manager jedoch zunächst die veränderten Anforderungen identifizieren. Zudem eignet sich eine solche Anpassung – aufgrund der bei der Rekonfiguration entstehenden Kosten – nur für die Abbildung langfristiger Anforderungsänderungen. Im Gegensatz zu *Mode-Based Scheduling* kann bei diesem Ansatz ein ungenutzter Timeslot nicht kurzfristig einem anderen Knoten für dessen Übertragungen überlassen werden.

Des Weiteren unterscheidet sich der zentralistische Kontrollansatz von WirelessHart auch grundlegend von den Designkonzepten und Prinzipien von BiPS. Während WirelessHart sehr stark auf eine dynamische Konfiguration setzt und eine Anpassung der Zuordnung der Time Slots zur Laufzeit ermöglicht, erlaubt BiPS lediglich den Wechsel zwischen verschiedenen statischen Superslot-Konfigurationen und verzichtet auf Frequenz-Hopping sowie FDMA. Die Zuordnung von Knoten zu den jeweiligen Slotregionen ist statisch und über die Definition der TOs an den jeweiligen Superslot gebunden.

Im Gegensatz zu WirelessHart ermöglicht BiPS jedoch mehr Freiheitsgrade in Bezug auf den Aufbau der Superslots. So besteht dieser bei BiPS aus einer Sequenz von Makroslots, die ihrerseits in Slotregionen unterteilt sind. Sowohl die Länge der Makroslots als auch die Länge der Slotregionen ist variabel. Dies erlaubt eine optimale Abstimmung auf die Kommunikationsanforderungen – sofern diese a priori bekannt sind. Zusätzlich gestattet BiPS für jede Slotregion individuell das in der Region verwendete MAC-Protokoll festzulegen. Dies ermöglicht sowohl die Nutzung exklusiver Reservierungen (RB-Protokoll) wie bei WirelessHart, als auch Regionen für Wettbewerb (CB-Protokoll) oder modusbasierten Medienzugriff (MB-Protokoll). Insbesondere die beiden letztgenannten Protokolle kompensieren die fehlende Flexibilität von BiPS im Vergleich zu WirelessHart in Bezug auf die Modifikation exklusiver Reservierungen und erlaubt die flexible dynamische Anpassung der Bandbreitenverteilung zur Laufzeit. Insbesondere das modusbasierte Protokoll gestattet eine sofortige Reaktion auf veränderte Anforderungen durch die Unterstützung von *Modus-Präferenzen* in Verbindung mit einem eingeschränkten Wettbewerb und ermöglicht die bandbreiteneffiziente Einplanung wichtiger, seltener sporadischer Nachrichten bei gleichzeitiger Gewährung deterministischer Garantien. Für diesen Typ von Nachrichten liefert WirelessHart keine Lösungskonzepte, sodass deren Anforderung nur sehr ineffizient durch exklusive Reservierungen abgebildet werden können. Dies führt dazu, dass in diesen Fällen zur Laufzeit viele Time Slots ungenutzt bleiben. Die Auflösung des Wettbewerbs in modusbasierten Slotregionen erfolgt dezentral durch *Fast Mode-Signaling*. Die effiziente Realisierung mittels *Fast Mode-Signaling* erspart die Kommunikationsaufwände, welche bei der Änderung der Zuordnung der Time Slots durch den Network Manager bei WirelessHart anfallen. Ein wesentlicher Vorteil, neben der Ausfallsicherheit, besteht darin, dass *Fast Mode-Signaling* eine sofortige Anpassung der Zuordnung noch innerhalb der gleichen Slotregion ermöglicht.

5.6.2 ISA 100.11a

ISA 100.11a wurde von der International Association of Automation (ISA) entwickelt und 2009 als Standard [ISA09, PC11, Mar12] veröffentlicht. Im Gegensatz zu WirelessHart war eines der Hauptziele bei der Entwicklung von ISA 100.11a, eine möglichst hohe Flexibilität sowie die Zusammenarbeit mit bereits vorhandenen Protokollen zu ermöglichen. Zu diesem Zweck unterstützt ISA 100.11a das Tunneling von existierenden drahtgebundenen Technologien, wie HART oder Modbus, aber auch drahtloser Technologien, wie WirelessHart und Wi-Fi [DK12]. Network- und Transport-Layer von ISA 100.11a basieren auf den offenen Standards 6LoWPAN, IPv6 und UDP [MKHC07, DH98, Pos80]. ISA 100.11a verwendet, wie WirelessHart, den Physical-Layer von IEEE 802.15.4 [IEE03]. Im Gegensatz zu WirelessHart verfügt der Data-Link-Layer von ISA 100.11a über eine integrierte Routing-Funktionalität.

Der Medienzugriff selbst basiert auf TDMA. Auch ISA 100.11a unterteilt die Zeit in Superframes, die aus einer Sequenz von Time Slots bestehen und periodisch wiederholt werden. Die Länge der Time Slots ist zwischen 10 ms und 12 ms konfigurierbar[25]. Wie WirelessHart nutzt auch ISA 100.11a Frequenz-Hopping. Das von ISA 100.11a verwendete Frequenz-Hopping wird als *Slotted Channel Hopping* bezeichnet – am Ende eines Time Slots wechselt ein Knoten den Kanal gemäß einer vorgegebenen Abfolge [PC11]. Über ein adaptives Blacklisting können störbehaftete Kanäle zur Laufzeit aus dieser Sequenz entfernt werden. Neben dem Slotted Channel Hopping unterstützt ISA 100.11a auch das sogenannte *Slow Channel Hopping*. Hierbei wird eine Sequenz von Time Slots gruppiert und nur jeweils am Ende einer solchen Gruppe der Kanal gewechselt. Die Betriebsart *Hybrid* gestattet es, beide Ansätze innerhalb eines Superframes zu kombinieren [PC11, Mar12], d.h., es wechseln sich Zeitabschnitte mit Slow Channel Hopping und solche mit Slotted Channel Hopping ab.

Ziel des Slow Channel Hopping ist die bessere Unterstützung ereignisbasierter Kommunikation für sporadische Nachrichten. Deshalb kommt innerhalb eines Zeitabschnittes mit Slow Channel Hopping kein TDMA zum Einsatz. Stattdessen konkurrieren die Knoten unter Verwendung von CSMA/CA um die Bandbreite. Daher kommt für periodische Nachrichten tendenziell eher Slotted Channel Hopping mit exklusiven Reservierungen zum Einsatz, während für sporadische Nachrichten oder erneute Übertragungen[26] eine Zeitspanne mit Slow Channel Hopping gewählt wird.

Bei den Time Slots, bei denen Slotted Channel Hopping zum Einsatz kommt, unterscheidet ISA 100.11a zwischen *dedicated* und *shared* Time Slots [DK12]. Ein dedicated Time Slot ist frei von Wettbewerb und maximal einem Sender zugeordnet, während in einem shared Time Slot *Priority CSMA-CA Scheme* [DK12] genutzt wird. Hierbei handelt es sich um eine Abwandlung der CSMA/CA Variante aus IEEE 802.15.4, welche die Abbildung von Prioritäten unterstützt [ISA09, DK12]. Der ISA 100.11a-Standard sieht 16 Level zur Repräsentation von Prioritäten vor, empfiehlt jedoch die Verwendung von einer der fünf Kategorien: *urgent*, *high*, *medium*, *low* oder *journal*. Die Prioritäten werden über unterschiedlich lange Wartezeiten abgebildet. Möchte ein Knoten eine Nachricht versenden, wird zunächst eine mit niedrigerer Priorität zunehmende Zeitspanne abgewartet und erst dann geprüft, ob das Medium frei ist. Ist das Medium frei, erfolgt die sofortige Übertragung des Rahmens. Falls das Medium zu diesem Zeitpunkt belegt ist, wird der verwendete Backoffexponent *BExp* erhöht (welcher initial den Wert 0 besitzt) und ein zufälliger Backoff aus dem Intervall $[0, 2^{BExp-1}]$ gezogen. Basierend auf dem gezogenen Backoff wird eine Backoffzeit errechnet. Innerhalb dieser Zeitspanne unternimmt der Knoten keinen erneuten Übertragungsversuch. Nach deren Ende erfolgt ein erneuter Übertragungsversuch, entsprechend dem zuvor beschriebenen Vorgehen [DK12]. Dieser Vorgang wird solange wiederholt, bis die Nachricht übertragen oder deren maximale Lebensdauer überschritten wurde.

Im Gegensatz zu den modusbasierten Slotregionen in BiPS sind Kollisionen innerhalb eines shared Time Slots durchaus möglich, da in einem Slot unterschiedliche Knoten Rahmen mit der gleichen Priorität

[25] Das Timing innerhalb eines Time Slots ist im ISA 100.11a-Standard jedoch nur für Time Slots mit einer Länge von 10 ms spezifiziert.

[26] Hierbei handelt es sich um in der vorherigen Slotted Channel Hopping Zeitspanne fehlgeschlagene Übertragungen.

übertragen dürfen. Dieser Fall ist bei *Mode-Based Scheduling* ausgeschlossen bzw. verboten, da ansonsten keine deterministischen Garantien für die Nachricht mit der höchsten Priorität mehr möglich sind. Wie BiPS unterstützt ISA 100.11a die Unterteilung von Superslots bzw. Superframes in unterschiedliche Bereiche mit verschiedenen Medienzugriffsverfahren. ISA 100.11a unterstützt – im Gegensatz zu BiPS – nur exklusive Reservierungen, wettbewerbsbasierten Zugriff sowie ein prioritätsbasiertes (nicht kollisionsgeschütztes) Verfahren für shared Time Slots. Des Weiteren sind die einzelnen Time Slots von ISA 100.11a zwar konfigurierbar, jedoch nur innerhalb eines eingeschränkten Bereiches, während BiPS dem Designer diesbezüglich keinerlei Einschränkungen auferlegt.

5.6.3 Fazit

In Bezug auf Flexibilität und Anpassbarkeit ist BiPS sowohl WirelessHart als auch ISA 100.11a, was den Aufbau von Superslot bzw. Superframes angeht, überlegen. So erlauben die Makroslots in BiPS die Definition beliebig langer Slotregionen sowie die Zuordnung der Slotregionen zu unterschiedlichen MAC-Protokollen. Zudem können mehrere Makroslots zu einem Superslot kombiniert und zur Laufzeit zwischen unterschiedlichen Superslot-Konfigurationen gewechselt werden. Auf diese Weise ist eine optimale Anpassung an die Kommunikationsanforderungen möglich. Zudem kann BiPS über dessen generische Schnittstellen und das Konzept der Slotregionen jederzeit um neue MAC-Protokolle erweitert werden, sollten die bereits vorhandenen die Anforderungen nicht erfüllen können. Aktuell decken die in BiPS integrierten MAC-Protokolle sowohl eine rein reservierungsbasierte, zeitgetriggerte Betriebsart – wie von WirelessHart verwendet – als auch die Kombination von reservierungs- und wettbewerbsbasiertem Medienzugriff, wie in ISA 100.11a, ab. Insbesondere die modusbasierte Kommunikation mit ihrer Fähigkeit, deterministische Garantien bei gleichzeitiger effizienter Nutzung der vorhandenen Bandbreite zu ermöglichen, wird weder von WirelessHart noch von dem modifizierten CSMA/CA Verfahren von ISA 100.11a erreicht, auch wenn die shared Time Slots von ISA 100.11a die Umsetzung der Konzepte von *Mode-Based Scheduling* grundsätzlich erlauben würden.

Ein Vorteil von WirelessHart und ISA 100.11a gegenüber BiPS ist das Frequenz-Hopping. Hierüber kann in WirelessHart durch den Einsatz von FDMA die nutzbare Bandbreite erhöht oder wie bei ISA 100.11a (bei Verzicht auf FDMA) die Störanfälligkeit reduziert werden. Ein weiterer Vorteil des häufigen Frequenzwechsels ist die bessere Interoperabilität mit anderen Kommunikationstechnologien, die im gleichen Frequenzband operieren, wie z. B. IEEE 802.11 [IEE99] oder IEEE 802.15.1 [IEE02] (Bluetooth). Von besonderem Interesse ist hierbei die Möglichkeit, störbehaftete Kanäle zur Laufzeit vom Frequenz-Hopping auszuschließen [PC11, Mar12]. Dies ist ein insgesamt auch für BiPS interessanter Aspekt, der ggf. in zukünftigen Entwicklungen berücksichtigt werden wird.

5.7 Zusammenfassung

In diesem Kapitel präsentierten wir unsere Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* für Funknetzwerke auf der Grundlage von Backoffslots. Hierzu haben wir die generischen Constraints und Konzepte aus Kapitel 4 so erweitert, dass diese die hardwarespezifischen Verzögerungen und Eigenarten der von uns eingesetzten *Imote 2*-Plattform, in Verbindung mit dem verwendeten CC2420-Transceiver (IEEE 802.15.4), berücksichtigten. Auf diese Weise konnten wir exemplarisch zeigen, wie sich die allgemeinen Constraints an eine konkrete Hardwareplattform und Kommunikationstechnologien adaptieren lassen.

Um die Umsetzbarkeit und Praxistauglichkeit zu demonstrieren, haben wir anstelle eines einfachen Prototypen für die funktionale Evaluation ein vollständiges modusbasiertes MAC-Protokoll für unseren *Black burst-Integrated Protocol Stack* (BiPS) entwickelt und anschließend evaluiert. Die implementierungsspezifischen Eigenarten von BiPS (spezifische Verzögerungen und Implementierungsentscheidungen) wurden bei der Adaption der Constraints ebenfalls berücksichtigt. Das langfristige Ziel hinter diesen Bemühungen

ist es, neue Anwendungsgebiete für *Mode-Based Scheduling* im Bereich der Überwachung und Steuerung von Produktionsanlagen zu identifizieren und diese letztendlich mit BiPS zu erschließen.

Die abschließende funktionale Evaluation unseres modusbasierten MAC-Protokolls erfolgte im Rahmen eines einfachen Anwendungsszenarios. Hierzu wurden die Constraints verwendet, um drei unterschiedliche zulässige Konfigurationen zu bestimmen und diese hinsichtlich ihrer Robustheit und Funktion zu testen. Die Konfigurationsparameter wurden hierbei so gewählt, dass der resultierende Overhead (und somit auch der Sicherheitspuffer) minimiert wurde. Wie unsere Messreihen gezeigt haben, lieferten die Constraints – wie gefordert – in allen drei Fällen einen Satz Konfigurationsparameter, welcher einen zuverlässigen und robusten Betrieb ermöglichte.

Abschließend haben wir noch eine alternative Implementierung von *Fast Mode-Signaling* mittels ACTP betrachtet, welche auf den spezifischen Eigenschaften der BB-Kodierung aufbaut und von der Funktionsweise der Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* ähnelt. ACTP ermöglicht, im Gegensatz zu der auf Backoffslots basierten Variante, auch eine Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling* in Multi-Hop-Netzwerken.

Insgesamt konnten wir zeigen, dass eine Implementierung von *Mode-Based Scheduling with Fast Mode-Signaling* mittels Backoffslots für drahtlose Netzwerke mit handelsüblichen Transceivern problemlos realisierbar ist. Allerdings zeigen die Constraints und die Evaluation auch, dass nicht nur die maximale Synchronisationsungenauigkeit, sondern auch die Hardwareplattform und hier insbesondere der CC2420-Transceiver einen nicht unwesentlichen Anteil an dem Overhead zu verantworten haben. Somit ist eine Realisierung mit dem CC2420-Transceiver zwar möglich, es ergeben sich jedoch noch deutliche Optimierungspotentiale bei der Verwendung eines Tranceivers mit niedrigeren Umschaltzeiten sowie einem schnelleren CCA-Mechanismus.

6 Integration von Mode-Based Scheduling with Fast Mode-Signaling in das FlexRay-Protokoll

FlexRay ist ein zeitgetriggerter deterministischer Feldbus, welcher speziell gemäß den Anforderungen der Automobilindustrie für den Einsatz in sicherheitskritischen *X-by-Wire* Anwendungen (wie z. B. *Steer-by-Wire* oder *Brake-by-Wire*) entwickelt wurde. Langfristiges Ziel der *X-by-Wire*-Initiative ist es, die Ansteuerung der Aktuatoren allein durch digitale Datenübertragung zu realisieren und hierbei gänzlich auf physikalische Verbindungen (als Rückfallebene) zu verzichten. Dies ist jedoch nur möglich, wenn die gleiche Sicherheitsklasse wie bei klassischen Lösungen erreicht wird. Hierdurch ergibt sich aufgrund der hohen Anforderungen, den Charakteristika der Nachrichten sowie der beschränkten Bandbreite ein potentiell interessantes Anwendungsgebiet für *Mode-Based Scheduling with Fast Mode-Signaling*.

In diesem Kapitel präsentieren wir unterschiedliche Ansätze, um *Mode-Based Scheduling with Fast Mode-Signaling* konsistent in das FlexRay-Protokoll zu integrieren, die bereits Gegenstand unserer Patentschrift [6] sind[1]. Unsere Lösung beruht im Kern auf der in Kapitel 4 vorgestellten Abbildung von Modes auf Backoffslots. Die im FlexRay-Standard bereits definierten Strukturen sowie die Constraints für die Ableitung gültiger Konfigurationsparameter werden wir entsprechend um die neuen Konzepte (wie z. B. Backoffslots und BTSPs) erweitern. In Bezug auf eine mögliche Realisierung betrachten wir, inwiefern die jeweiligen Ansätze sich mit bereits existierenden FlexRay-Kommunikationscontrollern[2] umsetzen bzw. sich nachträglich in diese integrieren lassen. Hierbei diente der Funktionsumfang des Kommunikationscontrollers MB88121C von Toshiba [Fuj07] als Referenz[3].

Dieses Kapitel beginnt mit einer kurzen Einführung in das FlexRay-Protokoll sowie dessen technischen Grundlagen und Besonderheiten (Kapitel 6.1). Anschließend stellen wir zwei verschiedene Realisierungen für *Mode-Based Scheduling with Fast Mode-Signaling* in FlexRay vor: Die erste Variante besteht aus einer Integration in das dynamische Segment (Kapitel 6.2), welche lediglich einer Anpassung der Funktionalität der CC bedarf. Die zweite Variante realisierte *Mode-Based Scheduling with Fast Mode-Signaling* innerhalb des statischen Segments (Kapitel 6.3) und erfordert eine Anpassung des FlexRay-Standards. Aber auch diese Form der Umsetzung ist minimalinvasiv umsetzbar und widerspricht nicht den grundlegenden Konzepten und Prinzipien von FlexRay. Abschließend fasst Kapitel 6.4 die Ergebnisse noch einmal kurz zusammen.

Die Ergebnisse dieses Kapitel wurden in [6] publiziert.

6.1 Stand der Technik

Die Entwicklung des FlexRay-Protokolls begann 2000 durch ein Konsortium von Automobil- und Chipherstellern, welches sich unter anderem aus den Firmen BMW, Bosch, Daimler, Freescale, General Motors, NXP Semiconductors und Volkswagen zusammensetzte. Im Rahmen dieser Kooperation wurde zunächst 2002 die FlexRay Communications Systems Protocol Specification V2.0.2, gefolgt von der FlexRay Protocol Specification V2.1 Rev A [Fle05a] (2005), veröffentlicht. Die Arbeit des Konsortiums endete im Jahr 2010 mit der Veröffentlichung der FlexRay Communications Systems Protocol Specification

[1] Das entsprechende Patent wurde 2013 erteilt.

[2] Kurz CC für FlexRay Communication Controller.

[3] Der MB88121C basiert auf dem E-Ray FlexRay IP-Module von Bosch [Rob07]. Dieser gilt als Referenzimplementierung auch aufgrund der Tatsache, dass Bosch wichtiger Partner innerhalb des Konsortiums war.

V3.0.1 [Fle10b]. Diese stellt die Grundlage für die Überführung in den ISO-Standard ISO 17458 dar, welcher 2011 verabschiedet und publiziert wurde [Par12]. Der folgende Abschnitt fasst die Grundlagen von FlexRay zusammen und berücksichtigt hierbei den FlexRay 2.1 sowie FlexRay 3.0.1 Standard; sofern nicht anders angemerkt, gelten die Aussagen jeweils für beide Standards. Da FlexRay 3.0.1 die Grundlage des ISO 17458 Standards ist, gelten die Aussagen entsprechend auch für diesen.

Der FlexRay-Standard definiert die Schichten 1 und 2 (Physical- [Fle06, Fle10a] und Datalink-Layer [Fle05a, Fle10b]) des OSI Schichtenmodells und wurde speziell entsprechend den Anforderungen der Automobilindustrie entworfen [Fle05b]. Als Ausgangsbasis diente das von BMW entwickelte Byteflight Protokoll [BPG00] (vgl. Kapitel 4.1). Im Fokus der Entwicklung stand die Forderung nach einem deterministischen Kommunikationsprotokoll mit beschränkter Verzögerung und möglichst geringem Jitter. Eine weitere Forderung war die Unterstützung von *on-demand communication* (sporadischer Kommunikation). In diesem Kontext wurde ein Mechanismus gefordert, welcher eine dynamische Zuteilung von Bandbreite während der Laufzeit ermöglicht. Gleichzeitig musste jedoch sichergestellt sein, dass diese Form der dynamischen Zuteilung die deterministische Kommunikation nicht beeinträchtigt. Zudem sollte der Anteil der für die dynamische Zuteilung zur Verfügung stehenden Bandbreite konfigurierbar sein. Gesamtziel war die Entwicklung eines skalierbaren, fehlertoleranten und deterministischen Feldbussystems. Die zuvor aufgeführten Punkte (ID 285-288 in [Fle05b]) repräsentieren nur einen kleinen Auszug aus den zentralen Anforderungen für die Entwicklung des FlexRay-Protokolls, die in [Fle05b], als Zielvorgaben des Projektes genannt werden.

Um diese Vorgaben zu erfüllen und insbesondere auch die notwendige Robustheit für den Einsatz in sicherheitskritischen Anwendungen gewährleisten zu können, greifen verschiedene Mechanismen sowohl auf dem Physical- als auch dem Datalink-Layer ineinander. Der folgende Abschnitt gibt eine kurze Einführung in die wichtigsten Grundlagen von FlexRay, beginnend mit dem Physical-Layer, soweit diese für die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in FlexRay relevant und für eine Abgrenzung hilfreich sind.

6.1.1 Physical-Layer

Ein FlexRay Cluster besteht aus einer Menge von Knoten, die über einen oder zwei physikalische Kommunikationskanäle (kurz Kanäle) miteinander verbunden sind. Die Spezifikation von FlexRay sieht hierbei sowohl die Verwendung elektrischer als auch optischer Medien vor, wobei zur Zeit lediglich der Electrical Physical Layer (ELP) spezifiziert wurde [Rau08, Fle06, Fle10a]. Die beiden Kanäle werden mit *A* und *B* bezeichnet und können sowohl zur redundanten Übertragung von Daten als auch zur Verdopplung der verfügbaren Bandbreite genutzt werden. Der Standard sieht pro Kanal Datenraten von bis zu 10 MBit/s vor (weitere zulässige standardisierte Datenraten sind 2,5 und 5 MBit/s).

Der ELP unterstützt verschiedene Netzwerktopologien, hierzu zählen Bustopologien, Sterntopologien mit aktiven oder passiven Sternkopplern[4] sowie Kombinationen beider Topologieformen. Auch die Verwendung unterschiedlicher Topologien für beide Kanäle ist zulässig. Des Weiteren ist es ebenfalls erlaubt, Knoten nur über einen Kanal mit dem Cluster zu verbinden bzw. gänzlich bei der Auslegung der Topologie auf die Nutzung eines zweiten Kanals zu verzichten.

Ein FlexRay Knoten besteht im Allgemeinen aus einem Mikrocontroller, welcher die eigentliche Anwendungslogik implementiert – dem sogenannten Host – sowie einem FlexRay CC, welcher die Protokolllogik umsetzt. Jeder Kanal verfügt über einen separaten Bustreiber, welcher das binäre Signal des CC in ein Differenzsignal mit 4 Pegeln – Idle_LowPower[5], Idle, High und Low – und umgekehrt das empfangene Differenzsignal zurück in ein binäres Signal überführt.

[4] Aktive Sternkoppler verstärken das Signal und führen bei der Weiterleitung im Allgemeinen zu einer Verzögerung. Die Anzahl der kaskadierten aktiven Sternkoppler ist auf zwei beschränkt.

[5] Befinden sich die Bustreiber im Schlaf-Modus, so liegt eine Potentialdifferenz von 0 Volt an.

6.1.2 Datalink-Layer

FlexRay ermöglicht deterministische Garantien in Bezug auf Verzögerungen durch den Einsatz von TDMA. Hierzu wird die Zeit in eine Sequenz von 64 Kommunikationszyklen unterteilt, die permanent wiederholt werden. Eine solche Sequenz bezeichnen wir im Folgenden als Makrozyklus[6]. Die Kommunikationszyklen sind identisch strukturiert und werden konsekutiv nummeriert, wobei der erste Kommunikationszyklus eines Makrozyklus die Nummer 0 trägt.

Jeder Kommunikationszyklus ist unterteilt in bis zu vier Segmente: ein *statisches Segment*, ein *dynamisches Segment*, ein *Symbol Window* sowie die *Network Idle Time*. Die Länge dieser Segmente kann entsprechend den Vorgaben des FlexRay-Standards[7] konfiguriert werden.

Die Abbildung 6.1 gibt einen detaillierten Überblick über die zeitliche Strukturierung des Kommunikationszyklus sowie dessen feingranulare Unterteilung durch Einführung einer zeitlichen Hierarchie, die in den folgenden Kapiteln erläutert wird.

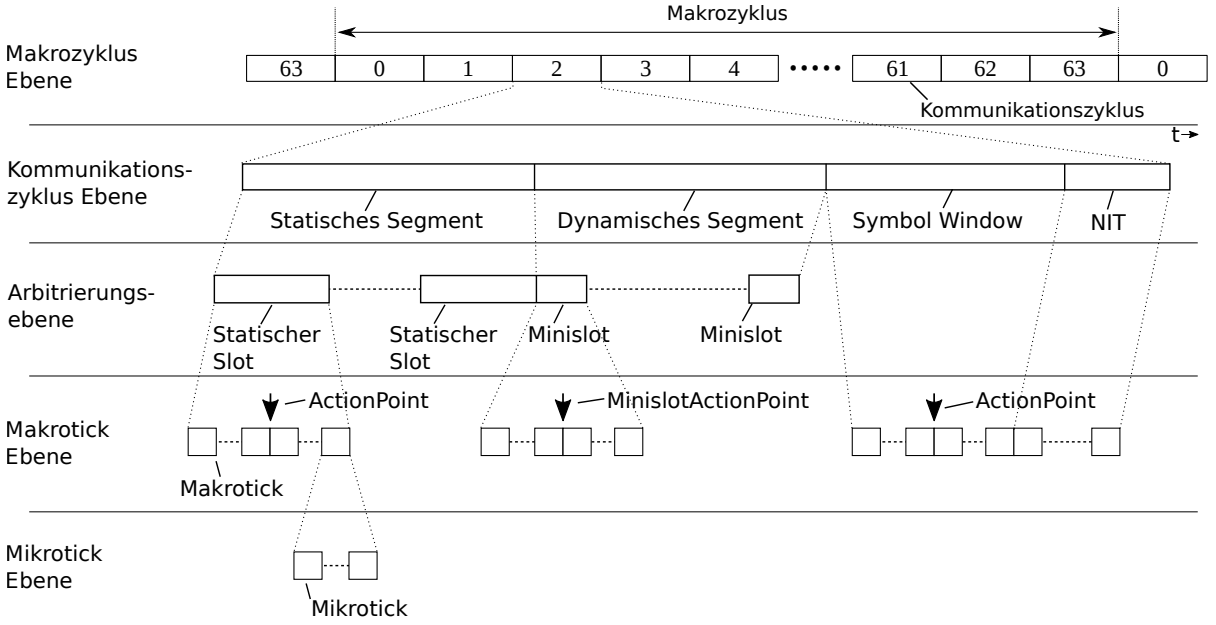


Abbildung 6.1: Zeitliche Hierarchie von FlexRay basierend auf [Fle10a, Abbildung 5-1].

Statisches Segment: Das statische Segment ist obligatorischer Bestandteil des Kommunikationszyklus und in eine feste Anzahl von statischen Slots unterteilt. Der Medienzugriff erfolgt mittels TDMA und basiert auf exklusiven Reservierungen.

Dynamisches Segment: Innerhalb des dynamischen Segments kommt FTDMA zum Einsatz, um den Medienzugriff zu realisieren. FTDMA erlaubt eine dynamische prioritätsbasierte Zuordnung der verfügbaren Bandbreite innerhalb des Segments (vgl. Byteflight). Das dynamische Segment ist optional und kann auch entfallen.

Symbol Window: Das ebenfalls optionale Symbol Window kann, durch den Einsatz speziell definierter Symbole, für den Austausch von Managment Informationen verwendet werden.

[6] Dieser Begriff ist nicht Bestandteil des FlexRay-Standards – dort wird kein Begriff für die beschriebene Sequenz eingeführt.

[7] Der FlexRay 3.0.1 Standard listet hierzu 73 Constraints auf. Das Einhalten der Constraints bei der Konfiguration garantiert eine reibungslose Kommunikation. Die Vielzahl der Parameter und Optionen erlaubt zwar eine optimale Anpassung der Kommunikationszyklen an die Anforderungen des Anwendungsgebietes, erschwert aber im Gegenzug auch die Konfiguration des Gesamtsystems.

Network Idle Time (NIT): Innerhalb der NIT erfolgt die Resynchronisation der Uhren der Knoten. Hierzu werden Korrekturwerte bestimmt und angewendet, die auf der Grundlage von Informationen ermittelt werden, die während des Kommunikationszyklus gesammelt wurden.

Ein Makrotick ist die kleinste gemeinsame Zeiteinheit aller Knoten des Clusters und kann in einem Intervall zwischen 1 µs und 6 µs konfiguriert werden. Die Gleichheit der Makroticks besteht nicht nur formal – durch eine identische Konfiguration –, sondern wird durch die Resynchronisation der Uhren der Knoten innerhalb der NIT während der gesamten Laufzeit aufrechterhalten [Rau08, Fle05a, Fle10b]. Daher erfolgt die Festlegung der Länge der Segmente und der meisten anderen Konfigurationsparameter in Makroticks.

Während Makroticks auf Clusterebene die kleinste Zeiteinheit darstellen, sind dies auf Knotenebene die Mikroticks. Ein Makrotick setzt sich wiederum aus einer Anzahl von Mikroticks zusammen. Da die Mikroticks direkt von den Taktgebern der Knoten abgeleitet werden, unterscheidet sich deren Länge in Abhängigkeit von dem jeweiligen Knoten. Dementsprechend kann sich auch die Anzahl von Mikroticks, die einen Makrotick bilden, je nach Knoten unterscheiden. Die Anzahl der Mikroticks, die einen Makrotick bilden, ist zudem während des Betriebs des Clusters nicht konstant, sondern wird im Rahmen der Resynchronisation angepasst, um den ClockSkew zwischen den Knoten zu kompensieren.

6.1.2.1 Das statische Segment

Das statische Segment ist unterteilt in eine feste (konfigurierbare) Anzahl gleichlanger statischer Slots – mindestens zwei[8] und maximal 1023 Slots. Die Länge der statischen Slots wird, gemäß den Constraints der FlexRay-Spezifikation, in Abhängigkeit von der gewählten maximalen Nutzlast der Rahmen konfiguriert. Die maximal zulässige Nutzlast beträgt 254 Bytes; in der Praxis werden jedoch meist kleinere Werte gewählt. So erlaubt zum Beispiel der von BMW verwendete Kommunikationszyklus [BPS08] lediglich eine maximale Nutzlast von 16 Bytes. Die statischen Slots eines Kommunikationszyklus sind sukzessive nummeriert, wobei der erste Slot des statischen Segments die Nummer 1 trägt.

Jeder statische Slot wird höchstens einem Knoten zur exklusiven Nutzung zugeordnet, sodass die Kollisionsfreiheit garantiert ist. Falls einem Knoten ein Slot x zugeteilt wird, so gilt diese Zuordnung für alle Slots mit der Nummer x in allen Kommunikationszyklen des Makrozyklus. Um dem Babbling Idiot Fehler [Tem98] zu begegnen, sendet ein Knoten in jedem ihm zugeordneten Slot des statischen Segments einen Rahmen. Werden von der Anwendung keine Daten zur Verfügung gestellt, so sendet der CC selbstständig einen leeren Rahmen – einen sogenannten Nullframe. Diese Form der Zuordnung von Slots zu Knoten wird vom Standard als *cycle-independent slot assignment* bezeichnet und in dieser Form sowohl von FlexRay 2.1a als auch von FlexRay 3.0.1 unterstützt [Fle05a, Fle10b].

FlexRay 3.0.1 unterstützt zusätzlich auch ein *cycle-dependent slot assignment*. Hierdurch kann, im Gegensatz zum *cycle-independent slot assignment*, ein statischer Slot einem Knoten nur für ausgewählte Kommunikationszyklen eines Makrozyklus exklusiv zugewiesen werden. Auf diese Weise können sich Knoten einen Slot mit der gleichen Slotnummer (in unterschiedlichen Kommunikationszyklen des Makrozyklus) teilen. Dies wird als *Slot Multiplexing* bezeichnet. Hierdurch ist eine effizientere bzw. flexiblere Slotzuordnung und Bandbreitennutzung möglich, ohne auf die Kollisionsfreiheit zu verzichten. FlexRay 2.1a unterstützt weder *cycle-dependent slot assignment* noch ist *Slot Multiplexing* im statischen Segment zulässig[9].

Pro Kommunikationszyklus können jedem Knoten beliebig viele statische Slots zugeordnet werden. Zusätzlich kann festgelegt werden, ob sich eine Reservierung auf beide oder nur einen der beiden Kanäle erstreckt. Bei der Verwendung beider Kanäle können diese verwendet werden, um den gleichen Rahmen zu übertragen (Redundanz) oder unterschiedliche Rahmen zur Verdopplung der Bandbreite.

[8] Es werden mindestens zwei Slots für die Zeitsynchronisation und den Startup-Vorgang des Clusters benötigt [Fle05a, Fle10b].
[9] Der FlexRay 2.1a Standard verbietet *Slot Multiplexing* im statischen Segment sogar explizit.

Jedem Knoten können ein (FlexRay 2.1a, FlexRay 3.0.1) oder zwei statische Slots (FlexRay 3.0.1), als primärer bzw. sekundärer *Keyslot* zugeordnet werden. Hierbei handelt es sich um *cycle-dependent slot assignments*, die für beide Kanäle gelten. Die reservierten Slots beider Kanäle müssen aber nicht zur redundanten Übertragung von Daten genutzt werden. Nur innerhalb von Keyslots ist es ausgewählten Knoten erlaubt, spezielle Rahmen – sogenannte Sync- oder Startup-Frames[10] – zu senden, die für die Synchronisation des Clusters sowie dessen Start notwendig sind. Für die Synchronisation benötigt FlexRay pro Kommunikationszyklus mindestens zwei Sync-Frames[11]. Aus Gründen der Ausfallsicherheit dürfen auch mehr als zwei Knoten Sync-Frames senden. FlexRay wendet bei der Synchronisation sowohl eine Offsetkorrektur als auch eine Steigungskorrektur an, um den Clock Skew zwischen den Knoten auszugleichen. Bei der Steigungskorrektur wird die Anzahl von Mikroticks, die einen Makrotick bilden, über den gesamten Kommunikationszyklus hinweg angepasst (Details siehe [WL88, Fle10b, Rau08]). Für die Integration neuer Knoten sowie den Start des Clusters werden pro Kommunikationszyklus mindestens zwei Startup-Frames benötigt. Auch hier können weitere Knoten (sogenannte Coldstarter) gewählt werden, welche Startup-Frames innerhalb ihrer Keyslots senden.

Die Abbildung 6.2 zeigt den Aufbau eines statischen Slots. Dieser enthält einen *ActionPoint*, der den Zeitpunkt des Übertragungsbeginns innerhalb des statischen Slots festlegt. Der Abstand des ActionPoints zum Beginn des Slots wird in Makroticks gemessen und als *ActionPointOffset* bezeichnet[12]. Der ActionPoint hat die gleiche Aufgabe wie der MTSP in Kapitel 4.4 und stellt sicher, dass alle Knoten den Beginn einer Übertragung dem gleichen Slot zuordnen können, und zwar auch dann, wenn die Uhren der Knoten maximal voneinander abweichen. Die maximale Abweichung zwischen den Uhren zweier beliebiger Knoten wird in FlexRay als *Precision* bezeichnet und prägt maßgeblich die minimal zulässige Größe des ActionPointOffsets gemäß Constraint 12 bzw. 13 [Fle10b][13].

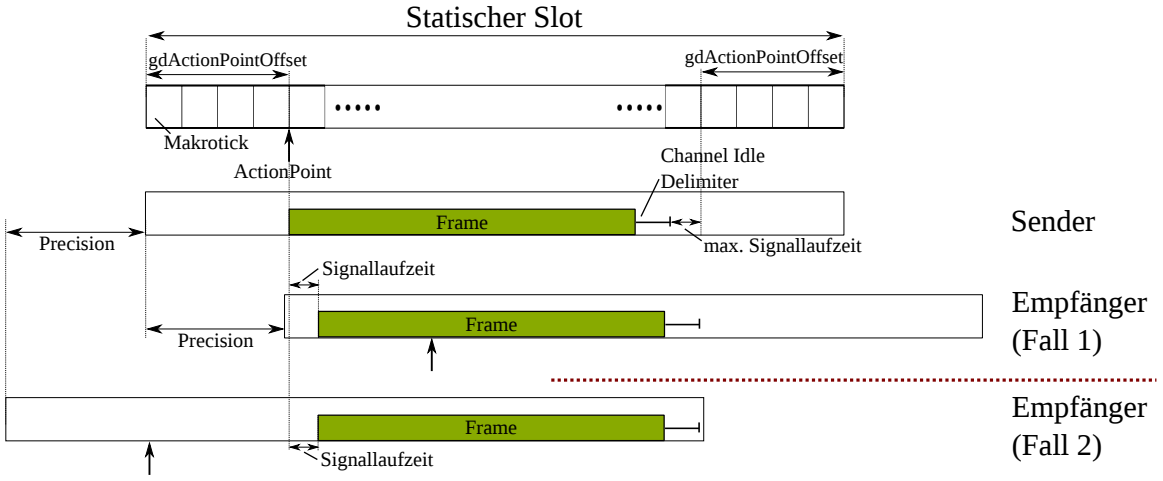


Abbildung 6.2: Aufbau des statischen Slots in Anlehnung an [Rau08, Abbildung 4.5] und [Fle10b, Abbildung 5-4].

[10] Bei der Konfiguration gibt es die Einschränkung, dass jeder Startup-Frame auch gleichzeitig ein Sync-Frame sein muss. Ein Startup- oder Sync-Frame wird durch ein gesetztes Bit im Header identifiziert und kann beliebige Nutzdaten transportieren. Auch evtl. durch den CC gesendete Nullframes haben das entsprechende Startup- resp. Sync-Bit gesetzt.

[11] Bei FlexRay 2.1a darf jeder Knoten höchstens einen Sync-Frame pro Kommunikationszyklus senden. FlexRay 3.0.1 erlaubt hingegen einen speziellen Synchronisationsmodus, bei dem ein einzelner Knoten zwei Sync-Frames pro Kommunikationszyklus sendet und damit alleine die Zeitbasis vorgibt.

[12] Der entsprechende Konfigurationsparameter wird im Standard mit *gdActionPointOffset* bezeichnet.

[13] Die angegebenen Constraints beziehen sich im Folgenden auf den FlexRay 3.0.1 Standard, da dieser aktueller ist und teilweise kleinere Korrekturen enthält. Bis auf die Korrekturen sind die Constraints jedoch weitestgehend identisch mit denen des FlexRay 2.1a Standards.

Um sicherzustellen, dass eine Übertragung für alle Knoten auch innerhalb der Grenzen des Slots abgeschlossen wird, geht der gewählte ActionPointOffset (und damit indirekt die Precision) doppelt in die Festlegung der minimalen Länge eines statischen Slots ein (Constraint 20 [Fle10b]). Zusätzlich werden ebenfalls Signallaufzeiten, Bauteilverzögerungen, die maximale Dauer der Übertragung des Rahmens sowie Verzögerungen für das Erkennen des Endes einer Übertragung berücksichtigt. Die Abbildung 6.2 zeigt neben dem Aufbau eines statischen Slots auch die Auswirkungen der Uhrenabweichung bei der Übertragung eines Rahmens. Die Uhrenabweichung zwischen Sender und Empfänger entspricht in diesem Beispiel der Precision. Der ActionPointOffset wurde bei diesem Beispiel so gewählt[14], dass er im Wesentlichen der Precision entspricht. Hierbei ist anzumerken, dass Fall 1 und Fall 2 nicht gleichzeitig innerhalb des Clusters auftreten können (da ansonsten die Abweichung zwischen den beiden gezeigten Empfängern der doppelten dargestellten Precision entsprechen würde). Wie in der Abbildung zu sehen, stellt die Konfiguration (entsprechend den Constraints des Standards) in beiden Fällen sicher, dass Beginn und Ende der Übertragung bei den Empfängern innerhalb der Slotgrenze liegt.

6.1.2.2 Das dynamische Segment

Innerhalb des optionalen dynamischen Segments erfolgt der Zugriff auf das Medium unter Verwendung von FTDMA. Der verwendete FTDMA-Mechanismus weist hierbei Parallelen zu Byteflight [BPG00] auf. Das dynamische Segment ist unterteilt in gleich lange Minislots (vgl. Constraint 51 [Fle10b]). Diese werden zur Laufzeit zu dynamischen Slots zusammengefasst. Dabei ist die Anzahl der Minislots, welche einen dynamischen Slot bilden, von der Länge des versendeten Rahmens abhängig. Jedoch besteht jeder dynamische Slot aus mindestens einem Minislot. Die dynamischen Slots sind, wie die statischen Slots, fortlaufend nummeriert (die Nummerierung der statischen Slots des statischen Segments wird fortgeführt).

Jeder dynamische Slot darf, für jeden Kanal, maximal einem Knoten pro Kommunikationszyklus zugeteilt werden[15]. Auf diese Weise ist sichergestellt, dass auch im dynamischen Segment keine Kollisionen auftreten. Die Zuordnung von dynamischen Slots zu Knoten kann sowohl in Form von *cycle-independent slot assignments* als auch *cycle-dependent slot assignments* erfolgen. Auch das *Slot Multiplexing*, also die Zuordnung eines dynamischen Slots zu unterschiedlichen Knoten in verschiedenen Kommunikationszyklen des Makrozyklus, ist sowohl in FlexRay 3.0.1 als auch in FlexRay 2.1a explizit erlaubt. Im Gegensatz zum statischen Segment werden keine Nullframes gesendet, wenn innerhalb eines reservierten dynamischen Slots kein Rahmen gesendet wird. In diesem Fall bleibt der dynamische Slot ungenutzt und hat eine Länge von einem Minislot.

Um den Medienzugriff zu realisieren, verwalten die CCs einen unabhängigen Slotzähler für die beiden physikalischen Kanäle, welcher mit jedem statischen bzw. dynamischen Slot um eins erhöht wird. Die Aktualisierung des Slotzählers erfolgt im dynamischen Segment jeweils unmittelbar vor dem Ende des Minislots. Läuft zu diesem Zeitpunkt bereits eine Übertragung, so entfällt die Aktualisierung. Das heißt, der Slotzähler wird solange nicht erhöht, bis eine laufende Übertragung vollständig abgeschlossen ist. Verfügt ein Knoten über eine Reservierung innerhalb des dynamischen Segments und es liegt eine Nachricht zur Übertragung vor, so vergleicht der CC jeweils den internen Slotzähler mit der Slotnummer des reservierten dynamischen Slots. Sobald diese beiden Werte identisch sind, beginnt der CC mit der Übertragung des Rahmens. Zuvor prüft der Knoten jedoch, ob die Übertragung noch innerhalb der Grenzen des dynamischen Segments abgeschlossen werden kann[16]. So wird sichergestellt, dass das dynamische

[14] In dem hier gezeigten Beispiel wurde bei der Wahl des ActionPointOffset nur Constraint 12 berücksichtigt. Constraint 13 betrifft ebenfalls die Wahl des ActionPointOffset, dessen Berücksichtigung ist jedoch optional; vergleiche hierzu die Erläuterungen zur Cliquenbildung in [Fle10b, Rau08].

[15] Reservierungen von Slots gelten im dynamischen Segment jeweils nur für einen Kanal. Redundante Übertragungen werden nicht unterstützt, da aufgrund des FTDMA Verfahrens eine gleichzeitige Übertragung von Rahmen mit der gleichen dynamischen Slotnummer auf unterschiedlichen Kanälen nicht gewährleistet ist.

[16] Hierfür existiert der Konfigurationsparameter *pLatestTX* des CC. Dieser gibt den letzten Minislot an, in dem der CC eine Übertragung starten darf (Constraint 36 [Fle10b]).

Segment eine feste Länge besitzt und sich die Übertragungen im nachfolgenden Kommunikationszyklus nicht verzögern.

Der interne Aufbau der Minislots ähnelt dem der statischen Slots. Jeder Minislot verfügt über einen *MinislotActionPoint*. Dieser legt den Zeitpunkt fest, zu dem ein Sender seine Übertragung innerhalb des Minislots (gemäß seiner lokalen Uhr) startet. Der Abstand zwischen dem Beginn eines Minislots und dessen MinislotActionPoint wird als *MinislotActionPointOffset* bezeichnet und in Makroticks angegeben. Wie der ActionPoint bei den statischen Slots stellt der MinislotActionPoint sicher, dass alle Knoten den Beginn einer Übertragung dem gleichen Minislot zuordnen können[17]. ActionPointOffset und MinislotActionPointOffset lassen sich getrennt konfigurieren und ermöglichen es dem Designer, einen Tradeoff zwischen Bandbreitenanforderung und Robustheit zwischen Segmenten zu wählen. Dies ist sinnvoll, da innerhalb des statischen Segments in der Regel die sensibleren Daten übertragen werden. Daher wird der ActionPointOffset häufig größer gewählt als der MinislotActionPointOffset.

Die minimale Länge eines Minislots hängt von der Wahl des MinislotActionPointOffsets, der Precision sowie der maximalen Übertragungsverzögerung ab und wird von Constraint 18 [Fle10b] beschrieben. Werden die Minislots sehr klein gewählt, benötigt man unter Umständen zusätzliche Minislots, damit alle Knoten das Ende der Übertragung sicher erkennen und dem gleichen Minislot zuordnen können[18] (vgl. Constraint 27 [Fle10b]). Diese Zeitspanne wird als *Dynamic Slot Idle Phase* bezeichnet. Für Übertragungen im dynamischen Segment gilt, dass diese durch den Sender beim Erreichen des MinislotActionPoint gestartet werden und auch bei einem MinislotActionPoint enden. Um dies zu erreichen, wird die Übertragung durch den CC ggf. künstlich verlängert (*dynamic trailing sequence – DTS*). Die Abbildung 6.3 zeigt das dynamische Segment eines Kommunikationszyklus, den Aufbau der Minislots sowie die Aktualisierung des Slotzählers. Dargestellt wird die Übertragung von zwei Rahmen in den dynamischen Slots m und $m+3$, während die dynamischen Slots $m+1$ und $m+2$ in diesem Kommunikationszyklus ungenutzt bleiben.

Die Zuordnung von dynamischen Slots zu Nachrichten (resp. Knoten) entspricht einer Priorisierung, eine niedrigere Slotnummer repräsentiert eine höhere Priorität. Hierdurch wird eine prioritätsbasierte Vergabe der Bandbreite des dynamischen Segments umgesetzt. Wann genau und ob eine Nachricht innerhalb des dynamischen Segments übertragen wird, hängt davon ab, wie viele Nachrichten mit höherer Priorität in diesem Kommunikationszyklus gesendet werden und wie groß deren Nutzdaten sind. In der

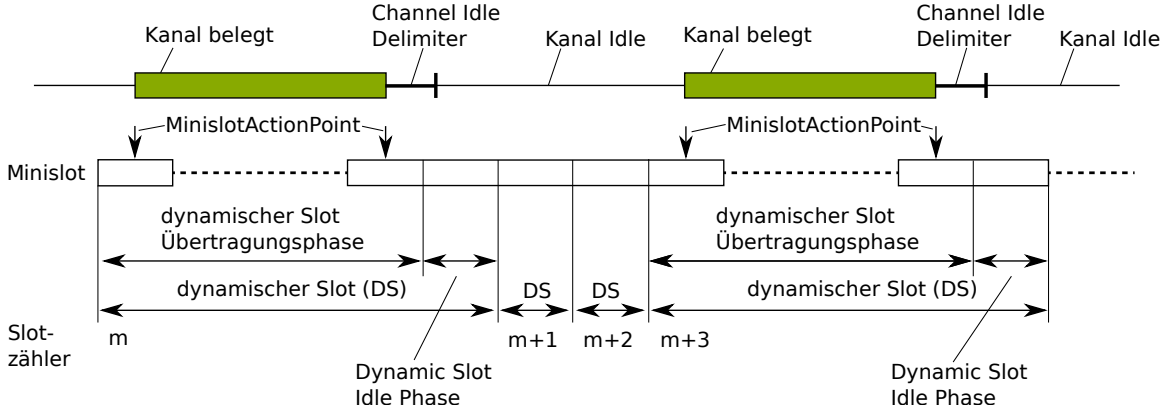


Abbildung 6.3: Aufbau des dynamischen Segments [Fle10b, Abbildung 5-7].

[17] Dass ActionPoint und MinislotActionPoint die gleichen Anforderungen erfüllen, ist auch daran ersichtlich, dass die entsprechenden Constraints identisch sind (Constraint 12 und 14).

[18] Ein Kanal wird als frei erkannt, wenn für die Zeitspanne (Channel Idle Delimiter) vom Bustreiber kein LOW-Pegel an den CC weitergereicht wird.

Konsequenz ist der zu erwartende Jitter umso größer, je niedriger die Priorität einer Nachricht ist. Ein weiterer Nachteil ist, dass jede zugeordnete Priorität, im Gegensatz zu CAN, Bandbreite beansprucht, auch wenn kein Rahmen übertragen wird, da die minimale Länge eines dynamischen Slots ein Minislot beträgt.

6.1.3 Abgrenzung zu Mode-Based Scheduling with Fast Mode-Signaling

Das statische Segment von FlexRay mit seiner Unterteilung in statische Slots unterstützt ausschließlich exklusive Reservierungen, d.h., ein statischer Slot wird höchstens einem Knoten zugeteilt. Durch den festen Aufbau und die feste Länge des Kommunikationszyklus ermöglichen die exklusiven Reservierungen deterministische Garantien bezüglich der Echtzeitfähigkeit. Gerade für seltene sporadische Nachrichten führt die Verwendung exklusiver Reservierungen jedoch zu einer sehr ineffizienten Nutzung der verfügbaren Bandbreite, da die reservierten Slots häufig ungenutzt bleiben.

Für sporadische Nachrichten sieht FlexRay daher die Verwendung des dynamischen Segments vor. Dieses erlaubt eine prioritätsbasierte (dynamische) Zuteilung der verfügbaren Bandbreite auf Grundlage von FTDMA. Allerdings können deterministische Garantien bzgl. der auftretenden Verzögerungen nur für die Nachricht mit der höchsten Priorität gewährt werden. Dies liegt darin begründet, dass sich der Zeitpunkt der Übertragung von Nachrichten mit niedrigeren Prioritäten (relativ zum Start des dynamischen Segments) danach richtet, ob in diesem Zyklus bereits Nachrichten mit höheren Prioritäten gesendet wurden und wie groß diese waren. Dementsprechend wird auch der maximale Jitter der auftreten kann mit niedrigerer Nachrichtenpriorität größer. Unter Umständen können, abhängig vom Ablaufplan und dem Nachrichtenaufkommen innerhalb eines Kommunikationszyklus, niedrig priorisierte Nachrichten gar nicht mehr in dem aktuellen Zyklus übertragen werden. Aus diesen Gründen sind für Nachrichten mit niedrigeren Prioritäten nur eingeschränkt Garantien bzgl. der Echtzeitfähigkeit möglich.

Im Gegensatz hierzu erlaubt der von *Mode-Based Scheduling with Fast Mode-Signaling* eingesetzte kontrollierte, slotbasierte Wettbewerb sowohl eine effizientere Bandbreitennutzung bei seltenen sporadischen Nachrichten als auch deterministische Garantien bzgl. der Echtzeitfähigkeit. Durch die Verwendung von Modes und *Modus-Präferenzen* sind deterministische Garantien bzgl. der auftretenden Verzögerungen für die Nachricht mit der höchsten *Modus-Präferenz* eines Slots gewährleistet. Im Gegensatz zu exklusiven Reservierungen kann der Slot jedoch für die Übertragung niedriger priorisierter Nachrichten verwendet werden, wenn keine höher priorisierte Nachricht übertragen werden muss.

Aus unserer Sicht ergänzt *Mode-Based Scheduling with Fast Mode-Signaling* daher in idealer Weise die Übertragungsmöglichkeiten, die FlexRay bereits anbietet, und hilft dabei die Anforderungen seltener sporadischer Nachrichten mit hohen Echtzeitanforderungen noch besser und gleichzeitig effizienter (in Bezug auf die Nutzung der verfügbaren Bandbreite) abzubilden. Wir haben zwei unterschiedliche Ansätze entwickelt *Mode-Based Scheduling with Fast Mode-Signaling* in FlexRay zu integrieren: Eine vollständig mit dem FlexRay-Standard konforme Abbildung innerhalb des dynamischen Segments (Kapitel 6.2) sowie eine Integration in das statische Segment, welche eine Erweiterung des FlexRay-Standards erfordert (Kapitel 6.3).

6.2 Mode-Based Scheduling with Fast Mode-Signaling innerhalb des dynamischen Segments

Wir betrachten zunächst die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in das dynamische Segment. Ziel der Entwicklung war es, die bereits vorhandenen Mechanismen und Freiheitsgrade des FlexRay-Protokolls so auszunutzen, dass *Mode-Based Scheduling* ohne Anpassung des Protokolls auf konzeptioneller Ebene umgesetzt werden kann. Der im dynamischen Segment verwendete FTDMA Mechanismus ermöglicht die einfache Realisierung von *Mode-Based Scheduling with Fast Mode-Signaling*

mittels Backoffslots. Hierbei übernehmen im Wesentlichen die Minislots die Aufgabe der Backoffslots, und der FTDMA-Mechanismus implementiert die Logik für das *Fast Mode-Signaling*. Wir beschreiben zunächst die Realisierung eines einzigen modusbasierten Slots pro Kommunikationszyklus (und Kanal) am Anfang des dynamischen Segments. Anschließend untersuchen wir Implementierungsmöglichkeiten unter Verwendung bereits existierender CC und diskutieren die hierbei auftretenden Probleme und Einschränkungen. Aufbauend hierauf diskutieren wir dann Verallgemeinerungen, die mehrere modusbasierte Slots innerhalb eines dynamischen Segments gestatten.

Das abstrakte Kommunikationsmodell aus Kapitel 2, welches die Grundlage für die Definition der Modes, *Modus-Präferenz-* sowie *Slot-Assignment*-Funktion bildet, ist hinreichend allgemein, so dass die Definitionen aus Kapitel 2 weiterhin sinngemäß verwendet werden können. Hierzu genügt es, den Makrozyklus von FlexRay mit den Makroslots aus Definition 2.1 gleichzusetzen. Die dort eingeführte Sequenz der Mikroslots entspricht konzeptionell der Sequenz von statischen Slots und modusbasierten Slots des dynamischen Segments. NIT, Symbol Window sowie die verbleibenden dynamischen Slots des dynamischen Segments sind ebenfalls spezielle Mikroslots, die jedoch bereits reserviert sind. Zur Vereinfachung betrachten wir die beiden physikalischen Kanäle von FlexRay jeweils wie zwei voneinander unabhängige Busse. Dies erlaubt einen höheren Freiheitsgrad (unterschiedliche Modes, aber vor allem auch unterschiedliche *Modus-Präferenz-* sowie *Slot-Assignment*-Funktionen pro Kanal), und gestattet es, die Definitionen des abstrakten Kommunikationsmodell unverändert zu verwenden[19].

6.2.1 Realisierung eines modusbasierten Slots im dynamischen Segment jedes Kommunikationszyklus

Die einfachste Variante der Umsetzung eines modusbasierten Slots besteht darin, diesen an den Anfang des dynamischen Segments zu legen und sich auf einen einzigen modusbasierten Slot pro dynamischem Segment (pro Kanal) zu beschränken. Das bedeutet, ein Makrozyklus enthält bei dieser Umsetzung maximal 64 modusbasierte Slots. Ein dynamischer modusbasierter Slot setzt sich aus mehreren dynamischen Slots zusammen: Die Minislots übernehmen die Rolle der Backoffslots, und die Priorisierung (bzw. das *Fast Mode-Signaling*) wird über den FTDMA-Mechanismus realisiert.

Dadurch, dass der modusbasierte Slot den Anfang des dynamischen Segments bildet, kommt es weder zu Verzögerungen noch zu Jitter bei der Übertragung eines modusbasierten Rahmens durch vorherige Übertragungen. Der Vorteil dieser Variante besteht in der vollständigen Kompatibilität zum FlexRay-Standard. Zudem genügen geringfügige Anpassungen an der Logik des CC, um diese Variante zu implementieren. Daher konnten wir mit Hilfe eines handelsüblichen FlexRay 2.1a CC zumindest zeigen, dass das vorgestellte Konzept funktioniert. Die aus unserer Sicht notwendigen funktionalen Anpassungen eines CC, um eine praxistaugliche Realisierung zu erhalten, werden am Ende dieses Abschnittes kurz beschrieben.

6.2.1.1 Abbildung der Modes (Modus-Präferenzen) auf dynamische Slotnummern

Die grundlegende Idee bei der Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* innerhalb des dynamischen Segments besteht darin, einen modusbasierten Slot durch eine Menge von dynamischen Slots (Slotnummern) am Anfang des Segments zu realisieren. Analog zu den bisherigen Vorgehen definieren wir eine mit der *Modus-Präferenz*-Funktion verträgliche Abbildung der Modes auf dynamische Slotnummern (vgl. Definition 4.3), welche die Bandbreite optimal verwendet[20] (vgl. Definition 4.5). Hierbei nutzen wir, dass eine kleinere *Modus-Präferenz* ebenso wie eine niedrigere dynamische

[19] Die Definition eines neuen Kommunikationsmodells für beide Kanäle erfordert lediglich eine Erweiterung der Funktionsdefinitionen um einen Parameter für den physikalischen Kanal. Da diese jedoch keine neuen Erkenntnisse bringt, verzichten wir hierauf.

[20] Das heißt sicherstellt, dass jeder Slotnummer ein Mode zugewiesen wird, für den ein Slot-Assignment für diesen modusbasierten dynamischen Slot existiert.

Slotnummer einer höheren Priorität entspricht. Die Logik, welche für *Fast Mode-Signaling* sowie den Medienzugriff benötigt wird, leistet dann in großen Teilen der bereits vorhandene FTDMA-Mechanismus.

Die in Kapitel 4 vorgestellten Strukturen für die Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* mittels Backoffslots existieren innerhalb des dynamischen Segments des FlexRay-Protokolls bereits, inkl. enstprechender Constraints, welche die korrekte Wahl der Konfigurationsparameter sicherstellen. Die Rolle der Backoffslots übernehmen bei dieser Realisierung die Minislots. Der bei den Backoffslots eingeführte BTSP – welcher den Startzeitpunkt einer Übertragung definiert und sicherstellt, dass alle Knoten den Start einer Übertragung dem korrekten Backoffslot (resp. Minislot) zuordnen können – entspricht dem MinislotActionPoint. Sowohl die Contraints für die Konfiguration des MinislotActionPoint als auch der Länge der Minislots berücksichtigen die gleichen Aspekte, wie die in Kapitel 4.4 formulierten Vorgaben zur Konfiguration des BTSP und der Backoffslots, jedoch bereits angepasst an die spezifischen Eigenschaften des FlexRay-Protokolls. Die Verwendung der bereits in FlexRay vorhandenen Konzepte gestattet die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* innerhalb von FlexRay, ohne hierfür das Protokoll selbst modifizieren zu müssen.

Definition 6.1 präsentiert eine optimale mit der *Modus-Präferenz*-Funktion *mp* verträgliche Abbildung (Definition 4.3) der Modes auf dynamische Slotnummern. Die Abbildung berücksichtigt hierfür nur die Modes, für die ein entsprechendes Slot-Assignment in dem jeweiligen modusbasierten Slot existiert.

Definition 6.1

Sei V die Menge der Knoten eines Netzwerkes, S die Menge aller Slots eines Makrozyklus, $S_{MB} \subseteq S$ die Menge der modusbasierten Slots und M die Menge der Modes. Des Weiteren sei mp eine Modus-Präferenz-Funktion und SA die Slot-Assignment Funktion. Zusätzlich bezeichne gNumberOfStaticSlots die Anzahl der statischen Slots des Kommunikationszyklus.
Dann ordnet die partielle Funktion $dsn_{mp} : S_{MB} \times M \to \mathbb{N}_0$ jedem Mode $m \in M$, eines modusbasierten Slots $s \in S_{MB}$ einen dynamischen Slot innerhalb des dynamischen Segments (anhand der Slotnummer) zu, in dem die Übertragung eines Rahmens dieses Modes gestartet werden darf. Die partielle Funktion $dsn_{mp} : S_{MB} \times M \to \mathbb{N}_0$ ist wie folgt definiert:

$$\forall s \in S_{MB}, \forall m \in M \text{ mit } SA(s,m) \text{ definiert} : dsn_{mp}(s,m) \equiv |M_s \backslash M_{s,m}| + gNumberOfStaticSlots + 1$$

mit

$$M_s \equiv \{m \in M | SA(s,m) \text{ ist definiert}\}$$
$$M_{s,m} \equiv \{m' \in M_s | mp(s,m) \leq mp(s,m')\}$$

Erfolgt die Abbildung der Modes wie in Definition 6.1 dargestellt, so leistet der FTDMA-Mechanismus eine Priorisierung entsprechend der vergebenen *Modus-Präferenzen*, in dem der Nachricht mit der höchsten *Modus-Präferenz* innerhalb eines modusbasierten Slots jeweils die niedrigste dynamische Slotnummer der Nachricht mit der zweithöchsten *Modus-Präferenz* die zweitniedrigste dynamische Slotnummer usw. zugeordnet wird. Die Verwendung von FTDMA führt jedoch dazu, dass alle für einen modusbasierten Slot eingeplanten Nachrichten (die über ein Slot-Assignment verfügen), nacheinander, entsprechend der durch die *Modus-Präferenzen* vorgegebenen Reihenfolge, gesendet werden, sofern das dynamische Segment ausreichend lang ist. Dies entspricht jedoch nicht der Idee von *Mode-Based Scheduling*, dort wird nur die Nachricht mit der höchsten *Modus-Präferenz* gesendet. Um dieses Verhalten nachzubilden, genügt es, das interne Verhalten des CC bei der Auswahl einer Nachricht für einen modusbasierten Slot geringfügig zu modifizieren. Hierbei nutzen wir aus, dass ein dynamischer Slot, in dem ein Rahmen übertragen wird, aufgrund des Paddings (DTS), mindestens zwei Minislots lang ist. Constraint 6.1 beschreibt die Anpassung des Verhaltens der CC für modusbasierte Slots des dynamischen Segments.

Constraint 6.1

Sei $f \in F$ ein Rahmen mit dem Mode $m \in M$ des Knotens $v \in V$ und $s \in S_{MB}$ ein modusbasierter Slot (des dynamischen Segments) sowie dsn_{mp} eine Abbildung gemäß Definition 6.1. Des Weiteren gelte $SA(s,m) = v$, d.h., es existiert ein Slot-Assignment des Knotens v mit dem Mode m für den Slot s. Zusätzlich bezeichne minislotCounter die Nummer des jeweils aktuellen Minislots unter der Annahme, dass diese fortlaufend nummeriert sind und der erste Minislot jedes Kommunikationszyklus die Nummer 1 *trägt.*
Um einen Rahmen für die Übertragung in einem modusbasierten Slot $s \in S_{MB}$ auszuwählen, durchsucht der CC des Knotens v seine Puffer für ausgehende Nachrichten gemäß einer festen Reihenfolge und sendet jeweils den ersten gefundenen Rahmen $f \in F$ für den gilt:
1.

$$isValid(cycleCounter, slotCounter, channel, f) \tag{6.1}$$

Das Prädikat $isValid : \mathbb{N}_0 \times \mathbb{N} \times Chan \times F$ prüft, ob für den gespeicherten Rahmen f in dem Kommunikationszyklus cycleCounter, dem dynamischen Slot slotCounter und dem Kanal $channel \in Chan \equiv \{A, B\}$ eine exklusive Reservierung existiert.
2.

$$minislotCounter - (dsn_{mp}(s,m) - gNumberOfStaticSlots) = 0 \tag{6.2}$$

Die erste Bedingung entspricht dem normalen Auswahlkriterium für die Übertragung von Rahmen gemäß dem FTDMA-Verfahren, welches FlexRay verwendet und wird in der Protokoll-Spezifikation (informell) beschrieben.
Das zweite Kriterium stellt sicher, dass in einem modusbasierten Slot des dynamischen Segments nur die Nachricht mit der höchsten Modus-Präferenz übertragen wird.

Das Constraint 6.1 lässt sich durch eine Anpassung des Verhaltens der CC realisieren und kann in eingeschränktem Rahmen sogar mit einem handelsüblichen CC nachgebildet werden[21]. Die Modifikation ist insoweit mit dem FlexRay-Standard konform, als dass sie weder unmodifizierte CC stört noch gegen die Grundkonzepte verstößt, welche das dynamische Segment prägen. Nicht modifizierte CC können die so versendeten modusbasierten Rahmen korrekt empfangen, auswerten und weiterverarbeiten.

Mode-Based Scheduling sieht vor, dass auch der Mode mit der niedrigsten *Modus-Präferenz* innerhalb eines Slots übertragen werden kann, sofern zuvor nicht bereits ein Rahmen mit einer höheren *Modus-Präferenz* gesendet wurde. Um dies sicherzustellen, muss das dynamische Segment ausreichend lang sein. Dies wird durch Constraint 6.2 gewährleistet. Es formuliert eine weitere Anforderung für die Wahl des Konfigurationsparameters *gNumberOfMinislots*, welcher die Länge des dynamischen Segments festlegt, die zusätzlich zu den bereits existierenden Vorgaben der FlexRay-Spezifikation erfüllt werden muss.

Constraint 6.2

Sei aMinislotPerDynamicFrame die benötigte Anzahl von Minislots zur Übertragung eines FlexRay Rahmens mit der konfigurierten maximal zulässigen Nutzdatenmenge gemäß Gleichung 26 [Fle10b].
Dann muss bei der Verwendung von Mode-Based Scheduling with Fast Mode-Signaling innerhalb des dynamischen Segments das dynamische Segment mindestens aus

$$gNumberOfMinislots \geq \max_{s \in S_{MB}, m \in M} (|M_s \backslash M_{s,m}|) + aMinislotPerDynamicFrame$$

Minislots bestehen.

[21] Unsere Lösung verwendet FlexRay 2.1a CC, ist jedoch für die Praxis nur sehr bedingt tauglich (vgl. Kapitel 6.2.1.3).

6.2.1.2 Beispiel

Die Tabelle 6.1 zeigt ein Beispiel für die Abbildung der Modes (und *Modus-Präferenzen*) auf die dynamischen Slots (Slotnummern) von FlexRay mit der Abbildung dsn_{mp} aus Definition 6.1. Das dargestellte Szenario besteht aus den Knoten $V = \{v_1, \ldots, v_8\}$ und verwendet die Modes $M = \{Emergency, Safety, Regular, Stream\}$. In diesem Beispiel beginnen jeweils ausgewählte dynamische Segmente des Kommunikationszyklus (eines Makrozyklus) mit einem modusbasierten Slot. Die Tabelle 6.1 zeigt die vier modusbasierten Slots und deren Zuordnung zu den Kommunikationszyklen des Makrozyklus (Spalte Zyklus) sowie das verwendete Slot-Assignment und die gewählten *Modus-Präferenzen*. Die Spalte mit dem Namen *pLatest* gibt die Nummer des letzten Minislots an, in welchem der CC die Übertragung eines Rahmens mit dem jeweiligen Mode starten darf. Das heißt, wenn der *miniSlotCounter* aus Ausdruck (6.2) in Constraint 6.1 den Wert *pLatest* annimmt, so ist die dort aufgeführte Bedingung erfüllt. Somit wird sichergestellt, dass jeweils nur der Rahmen mit der höchsten *Modus-Präferenz* in einem modusbasierten Slot des dynamischen Segments übertragen wird.

Slot s $\in$ S_{MB}	**Zyklus**	**Mode m $\in$ M**	*SA*	*mp*	$\lvert M_s \rvert$	$\lvert M_{s,m} \rvert$	*pLatest*	dsn_{mp}
1	8	*Emergency*	v_1	0	3	3	1	92
1	8	*Safety*	v_2	5	3	2	2	93
1	8	*Regular*						
1	8	*Stream*	v_4	15	3	1	3	94
2	16	*Emergency*	v_5	3	3	3	1	92
2	16	*Safety*	v_6	6	3	2	2	93
2	16	*Regular*						
2	16	*Stream*	v_3	14	3	1	3	94
3	32	*Emergency*	v_1	0	3	3	1	92
3	32	*Safety*	v_7	1	3	2	2	93
3	32	*Regular*	v_3	11	3	1	3	94
3	32	*Stream*						
4	63	*Emergency*	v_5	3	2	2	1	92
4	63	*Safety*						
4	63	*Regular*						
4	63	*Stream*	v_2	15	2	1	2	93

Tabelle 6.1: Zuordnung der *Modus-Präferenzen* zu den dynamischen Slots mit der Funktion dsn_{mp}.

Als Grundlage für alle unsere Beispiele dient der Aufbau des Kommunikationszyklus von BMW[22], welcher in [BPS08] vorgestellt wurde. Zum Einsatz kommt ein Kommunikationszyklus mit einer Dauer von 5 ms und einem dynamischen Segment mit einer Länge von 2 ms. Das statische Segment besteht aus insgesamt 91 statischen Slots mit einer maximalen Nutzlast von 16 Bytes pro Rahmen. Das dynamische Segment besteht aus 289 Minislots mit einer maximalen Nutzlast von 254 Bytes. Daher werden für die Übertragung eines Rahmens mit der maximalen Nutzlast, bei dieser Konfiguration, 40 Minislots benötigt. Somit ist das dynamische Segment auch hinreichend lang für die Abbildung der Modes (Constraint 6.2).

[22] Da BMW nicht alle Konfigurationsdetails veröffentlicht hat, wurden fehlende Parameter aus den vorhandenen Informationen extrapoliert und die Konfiguration an FlexRay 3.0.1 angepasst (die publizierte Konfiguration basierte auf FlexRay 2.1a). Die vollständige Konfiguration wurde mit Hilfe eines von uns entwickelten Werkzeuges erstellt, welches die Constraints aus dem FlexRay-Standard [Fle10b] prüft und Hinweise in Bezug auf die Grenzwerte, in Abhängigkeit der bereits festgelegten Parameter, gibt. Die entsprechenden Ausgaben des Tools mit näheren Informationen zu dem Aufbau des verwendeten Kommunikationszyklus sind im Anhang D.1 zu finden.

6.2.1.3 Realisierung mittels existierender CC

Um die Umsetzbarkeit unseres vorgestellten Konzepts zu überprüfen, haben wir im Rahmen einer prototypischen Implementierung ermittelt, inwiefern sich dieser Ansatz mit marktüblichen FlexRay 2.1a CC realisieren lässt. Als Ausgangsbasis für den Prototyp diente die Anbindung des FlexRay 2.1a CC MB88121C von Toshiba [Fuj07, Rob07] an den *Imote 2*, welche in der Diplomarbeit [Wie10, 1] von Matthias Wiebel entwickelt wurden. Für unseren Versuchsaufbau haben wir uns auf zwei Knoten v_1 und v_2 und zwei Modes $M = \{m_1, m_2\}$ beschränkt, um zu untersuchen, welche Probleme sich bei der Implementierung der vorgestellten Variante des *Fast Mode-Signaling* mit den Standard CC ergeben.

Der verwendete Kommunikationszyklus für die Evaluation ist identisch mit der im vorherigen Abschnitt vorgestellten Konfiguration (Länge des Kommunikationszyklus, Aufbau der Segmente etc.), allerdings verfügt in diesem Beispiel jedes dynamische Segment, d.h. jeder Kommunikationszyklus, über einen modusbasierten Slot. In unserem Szenario überträgt der Knoten v_1 die Rahmen mit dem Mode m_1, Knoten v_2 die Rahmen mit dem Mode m_2. Beide Knoten verfügen für jeden modusbasierten Slot über ein Slot-Assignment und planen auch in jedem Slot die Übertragung eines Rahmens des entsprechenden Modes ein. Der Mode m_1 besitzt in allen modusbasierten Slots in ungeraden, der Mode m_2 in allen modusbasierten Slots in geraden Kommunikationszyklen die höchste *Modus-Präferenz*.

Die Tabelle 6.2 identifiziert die modubasierten Slots anhand der Nummer ihres Kommunikationszyklus innerhalb des Makrozyklus und zeigt das verwendete Slot-Assignment sowie die *Modus-Präferenzen* und deren Abbildung auf dynamische Slotnummern mit der partiellen Funktion dsn_{mp} aus Definition 6.1. Die Spalte mit dem Namen *pLatest* gibt die Nummer des letzten Minislots an, in welchem der CC die Übertragung eines Rahmens mit dem jeweiligen Mode starten darf (vgl. Kapitel 6.2.1.2).

Zyklus	**Mode**	*SA*	*mp*	*pLatest*	dsn_{mp}
ungerade	m_1	v_1	0	1	92
ungerade	m_2	v_2	2	2	93
gerade	m_1	v_1	2	2	93
gerade	m_2	v_2	0	1	92

Tabelle 6.2: Zuordnung der *Modus-Präferenzen* zu den dynamischen Slots.

Bevor wir uns den Aspekten der Implementierung widmen und auf die Probleme, die sich durch den Einsatz handelsüblicher CC in der Praxis ergeben, eingehen, wollen wir an dieser Stelle bereits das Ergebnis vorwegnehmen. Über die Abbildung der *Modus-Präferenzen* auf die dynamischen Slots lässt sich *Fast Mode-Signaling* wie beschrieben realisieren. Allerdings erweist es sich als problematisch sicherzustellen, dass nur jeweils die Nachricht mit der höchsten *Modus-Präferenz* übertragen wird (vgl. Ausdruck (6.2) aus Constraint 6.1). Die von uns hierfür gefundene Lösung für Standard CC führt dazu, dass die modusbasierten Knoten innerhalb des restlichen dynamischen Segments nicht mehr als Sender agieren können.

Um in dem beschriebenen Szenario zu erreichen, dass jeweils nur der Rahmen mit der höchsten *Modus-Präferenz* übertragen wird, setzen wir bei beiden Knoten den Konfigurationsparameter $pLatestTX$ auf den Wert 2. Dieser Konfigurationsparameter dient eigentlich dazu, die Verletzung der Grenzen des dynamischen Segments zu verhindern, indem der Designer in diesem Parameter die Nummer des letzten Minislots hinterlegt, in dem der CC eine Übertragung starten darf. Da ein dynamischer Slot, in dem ein Rahmen übertragen wird, mindestens zwei Minislots lang ist, erreichen wir in diesem Fall unser gewünschtes Ziel. Die Wahl von $pLatestTX = 2$ funktioniert jedoch nur für das konkrete Szenario mit einer Zuordnung von zwei Modes pro modusbasiertem Slot (vgl. Spalte *pLatest* in Tabelle 6.2). Im

allgemeinen Fall muss $pLatestTX$ für jeden Knoten individuell auf die Nummer des Minislots gesetzt werden, in dem der letzte ihm (für einen Mode) zugeordnete dynamische Slot beginnt, wenn nicht zuvor die Übertragung eines anderen Rahmens mit einer höheren *Modus-Präferenz* gestartet wurde (siehe Ausdruck (6.2), Constraint 6.1).

Neben dem Nachteil, dass ein Knoten selbst nicht mehr als Sender innerhalb des restlichen dynamischen Segments auftreten kann, hat die Verwendung des Parameters $pLatestTX$ weitere Auswirkungen. So gilt der Parameter $pLatestTX$ für alle Kommunikationszyklen sowie alle Nachrichtenpuffer und kann zur Laufzeit nicht umkonfiguriert werden. Dies erweist sich als problematisch, wenn ein Knoten in verschiedenen Kommunikationszyklen Rahmen, deren Modes auf unterschiedliche dynamische Slotnummern abgebildet werden, verschicken soll. Dann muss $pLatestTX$ für jeden Knoten maximal gewählt werden. Das bedeutet, der Konfigurationsparameter $pLatestTX_v$ des Knotens $v \in V$ muss auf die größte dynamische Slotnummer gesetzt werden, welche v für den Versand eines Modes zugeteilt wird, d.h.

$$pLatestTX_v = \max_{m \in M, s \in S_{MB} \text{ mit } SA(m,s)=v} (dsn_{mp}(s,m) - gNumberOfStaticSlots).$$

Bei sehr vielen zugeordneten Modes pro modusbasiertem Slot muss (ggf. durch Fülldaten) sichergestellt werden, dass die Übertragung eines Rahmens stets mindestens bis zum $\max_{v \in V_{MB}}(pLatestTX_v + 1)$ Minislot andauert. Unsere Patentschrift [6] diskutiert diese Fragestellungen noch einmal ausführlich. Wie sich zeigt, ist hier eine elegante Lösung mit den vorhandenen Funktionalitäten der CC jedoch nicht realisierbar.

Zusätzlich wird die Praxistauglichkeit noch weiter eingeschränkt durch die Nachrichtenpuffer (TMB – *Transmit Message Buffer*), welche das Konzept der Modes nicht unterstützen. Die TMB können nur einzelnen Slotnummern zugeordnet werden, entweder für individuelle Zyklen oder nach bestimmten Mustern [Fle10b, Fuj07]. Werden einem Mode mehrere (verschiedene) dynamische Slotnummern in unterschiedlichen modusbasierten Slots (Kommunikationszyklen) zugeordnet, ist es erforderlich, mehrere getrennte TMBs zu reservieren. Eine zu sendende Nachricht muss nun manuell (durch den verwendeten Protokollstack) in den jeweils für den nächsten modusbasierten Slot zuständigen TMB verschoben werden, damit eine frühestmögliche Übertragung gewährleistet ist. Wird der Rahmen nicht versendet (aufgrund einer zu niedrigen *Modus-Präferenz*), muss die Nachricht aus dem TMB gelöscht und wieder rechtzeitig in den nächsten zuständigen TMB geschrieben werden. Dies führt zu einer starken temporalen Kopplung zwischen den Kommunikationszyklen und der Software, die innerhalb des Host läuft, sowie hohen Echtzeitanforderungen. Gerade dies sollte durch die Einführung von TMB vermieden werden, um die Zuverlässigkeit zu verbessern [Rau08]. Insgesamt ist hier die Entwicklung einer TMB-Struktur, welche die Ausführungssemantik des *Mode-Based Scheduling* direkt unterstützt, für einen zuverlässigen Einsatz in der Praxis sinnvoll und notwendig. Diese TMB-Puffer könnten dann auch über individuelle Konfigurationsparameter verfügen, die festlegen, welches der letzte Minislot ist, in dem die Übertragung einer abgelegten Nachricht gestartet werden darf. Somit ließe sich auch das im vorherigen Absatz beschriebene Problem mit dem Parameter $pLatestTX$ weitestgehend lösen.

Insgesamt zeigt unser Prototyp, dass das grundlegende Konzept der Abbildung von Modes (*Modus-Präferenzen*) auf dynamische Slots funktioniert und sogar bedingt mit handelsüblichen FlexRay 2.1a CC umsetzbar ist. Die skizzierten Anpassungen der Logik des CC, um zu einer praxistauglichen Lösung zu gelangen, sind vergleichsweise gering und erfordern keine konzeptuellen Veränderungen des FlexRay-Protokolls.

6.2.2 Realisierung mehrerer Mode-Based Scheduling-Slots innerhalb des dynamischen Segments

Konzeptionell lassen sich auch mehrere modusbasierte Slots pro dynamischem Segment abbilden, indem diese sequentiell hintereinander angeordnet werden. Hierfür ist lediglich eine entsprechende Anpassung der verwendeten Abbildung (Definition 6.1) der Modes auf Slotnummern erforderlich, sodass dieses bei

der Zuordnung, die bereits zu einem früheren modusbasierten Slot gehörenden dynamischen Slotnummern mitberücksichtigt (vgl. [6]). Als aufwändiger erweist es sich, wenn weiterhin sichergestellt werden soll, dass die Übertragungen ohne Jitter erfolgen, da der FTDMA-Mechanismus nicht erzwingt, dass ein dynamischer Slot zu einem bestimmten Zeitpunkt (d.h. festen Minislot) beginnt. Die skizzierte Umsetzung würde dazu führen, dass der exakte Übertragungsstart innerhalb der modusbasierten Slots eines dynamischen Segments von den Übertragungen (und deren Rahmenlänge) in vorherigen modusbasierten Slots abhängen würde. Der hierbei zu erwartende Jitter wächst mit der Anzahl eingeplanter modusbasierter Slots pro dynamischem Segment.

Um dies zu beheben, wird eine Möglichkeit benötigt, die sicherstellt, dass der erste dynamische Slot jedes modusbasierten Slots in jedem Makroslot zum gleichen Zeitpunkt (d.h. mit dem gleichen Minislot) beginnt. Da der FTDMA-Mechanismus diese Funktionalität nicht vorsieht, könnte man sich mit der Übertragung von Füllrahmen behelfen. Hierfür wird zwischen den modusbasierten Slots jeweils ein dynamischer Slot reserviert, in dem ein Füllrahmen der jeweilig benötigten Länge übertragen wird, um die korrekte Ausrichtung des nachfolgenden modusbasierten Slots zu erzwingen (siehe [6]). Diese Lösung reduziert jedoch die nutzbare Bandbreite.

Eine Alternative ist die Erweiterung des FTDMA-Mechanismus, sodass der Beginn eines dynamischen Slots fest konfiguriert werden kann. Dies würde letztendlich der Einführung eines neuen Segment-Typs entsprechen. Aus unserer Sicht ist – wenn man ohnehin eine Anpassung des Protokolls in Erwägung zieht – die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in das statische Segment von FlexRay eine bessere und flexiblere Lösung, insbesondere wenn es um die Übertragung wichtiger sporadischer Daten mit hohen Anforderungen bzgl. Echtzeitfähigkeit und Zuverlässigkeit geht.

6.3 Integration von Mode-Based Scheduling with Fast Mode-Signaling in das statische Segment

Ein alternativer Ansatz zur Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in das FlexRay-Protokoll stellt die Erweiterung der Funktionalität des statischen Segments dar. Dies ist jedoch mit einer Modifikation des FlexRay-Standards verbunden sowie entsprechenden Anpassungen des funktionalen Verhaltens der CC. Hierzu definieren wir zunächst zwei unterschiedliche Typen von statischen Slots: *exklusive statische Slots (ESSs)* und *modusbasierte statische Slots (MSSs)*. Bei den ESSs handelt es sich um klassische statische Slots, welche exklusiv einem Knoten zugeordnet werden. Die MSSs gestatten die modusbasierte Kommunikation und unterstützen *Fast Mode-Signaling* auf der Basis von Backoffslots (vgl. Kapitel 4). Im Gegensatz zu den klassischen statischen Slots ist ein MSS nicht exklusiv maximal einem Knoten pro Kommunikationszyklus zugeordnet, sondern einer Gruppe von ausgewählten Knoten, die zueinander in Konkurrenz stehen – entsprechend der Idee des *Mode-Based Scheduling*. Aus diesem Grund wird in MSSs generell auf die Übertragung von Nullframes verzichtet.

Das statische Segment besteht weiterhin aus einer Sequenz statischer Slots. Der Benutzer kann ESSs und MSSs beliebig miteinander kombinieren, um den Anforderungen seiner Anwendung optimal entsprechen zu können. Da die statischen Slots (ESSs und MSSs) in sich abgeschlossen sind und eine feste Länge aufweisen, kann das statische Segment beliebig viele ESSs und MSSs enthalten, ohne dass diese sich gegenseitig beeinflussen (z. B. Jitter). Dies stellt einen wesentlichen Vorteil gegenüber der skizzierten Realisierung mehrerer modusbasierter Slots pro dynamischem Segment dar[23]. Der Kommunikationszyklus selbst besteht aus dem statischen Segment und der NIT sowie einem optionalen dynamischen Segment und dem optionalen Symbol Window. Das statische Segment muss mindestens zwei ESSs enthalten, für die Übertragung der Startup- und Sync-Frames, damit die Mechanismen von FlexRay für Startup, Integration und Synchronisation weiterhin korrekt funktionieren.

[23] Ohne weitere Maßnahmen, um den Jitter zu kompensieren, welche wiederum zu Lasten der nutzbaren Bandbreite gehen.

Beim Aufbau der statischen MSSs nutzen wir die bereits vorhandene Struktur der klassischen statischen Slots und kombinieren diese mit dem in Kapitel 4.4.3 vorgestellten Aufbau modusbasierter Mikroslots. Hierbei versuchen wir gemeinsame Konzepte – soweit wie möglich – aus dem FlexRay-Standard zu übernehmen, um notwendige Anpassungen des Standards so minimal wie möglich zu halten. Deshalb folgen wir auch zunächst der Vorgabe von FlexRay, dass alle Slots des statischen Segments die gleiche Länge haben und identisch aufgebaut sind. Daher verwenden MSSs und ESSs die gleichen Konfigurationsparameter.

Wir konzentrieren uns zunächst auf den Aufbau der MSSs (Kapitel 6.3.1) und widmen uns dann der Erweiterung der Constraints der *FlexRay Protocol Specification 3.0.1* [Fle10b] (Kapitel 6.3.2). Kapitel 6.3.3 zeigt ein Beispiel für eine mit Hilfe dieser Constraints erstellte Konfiguration sowie das hierfür verwendete Werkzeug. Abschließend diskutieren wir mögliche Optimierungen, die jedoch mit umfangreicheren Modifikationen des FlexRay-Protokolls verbunden sind (Kapitel 6.3.4).

6.3.1 Aufbau modusbasierter statischer Slots

Der Aufbau eines MSSs entspricht dem eines modusbasierten Mikroslots (vgl. Kapitel 4.4.3). Ein MSS besteht aus einer bestimmten Anzahl von Backoffslots für die Implementierung des *Fast Mode-Signaling*, einem MTSP, der den frühesten Übertragungsbeginn vorgibt, sowie BTSPs innerhalb der einzelnen Backoffslots. Die Länge des Slots muss so gewählt werden, dass auch im Falle der maximalen Uhrenabweichung die Integrität der angrenzenden Slots gewahrt wird.

Viele dieser Merkmale erfüllen die statischen Slots aufgrund ihrer Constraints [Fle10b, Fle05a] bereits. Dies erlaubt es uns, MSSs als Erweiterung klassischer ESSs zu definieren und die bereits durch den FlexRay-Standard vorgegebenen Strukturen zu verwenden. So gibt der ActionPoint bei statischen Slots den Zeitpunkt an, zu dem ein Sender mit der Übertragung seines Rahmens beginnt; in einem MSS markiert der ActionPointOffset den frühesten Zeitpunkt, zu dem ein Sender mit der Übertragung seines Rahmens beginnen darf. Somit entspricht der ActionPoint in einem MSS dem MTSP eines modusbasierten Mikroslots. Die Backoffslots innerhalb der MSSs folgen sequentiell ohne Pause aufeinander und sind fortlaufend nummeriert, wobei der erste Backoffslot eines MSSs jeweils die Nummer 0 trägt. Der BTSP des ersten Backoffslots ist deckungsgleich mit dem ActionPoint des MSSs (vgl. Abbildung 6.4). Die Abbildung der Modes auf Backoffslots erfolgt anhand der in Definition 4.5 eingeführten partiellen Funktion slt_{mp}, welches mit der *Modus-Präferenz*-Funktion verträglich ist und dafür sorgt, dass jeder Backoffslot aufsteigend mit einem Mode (ohne Lücken) assoziiert wird.

Abbildung 6.4 zeigt den Aufbau eines MSSs mit 3 Backoffslots. Die Backoffslots sind in blau dargestellt; die rot gestrichelte Linie zeigt den konfigurierten BTSP. Hinsichtlich der minimalen Länge eines MSSs müssen die gleichen Anforderungen erfüllt werden wie bei einem ESSs. So muss gewährleistet werden, dass für alle Knoten die Übertragung innerhalb der Grenzen des Slots endet (vgl. Abbildung 6.2), auch wenn eine Übertragung im letzten Backoffslot mit der maximal zulässigen Nutzlast sowie maximaler Uhrenabweichung gestartet wird.

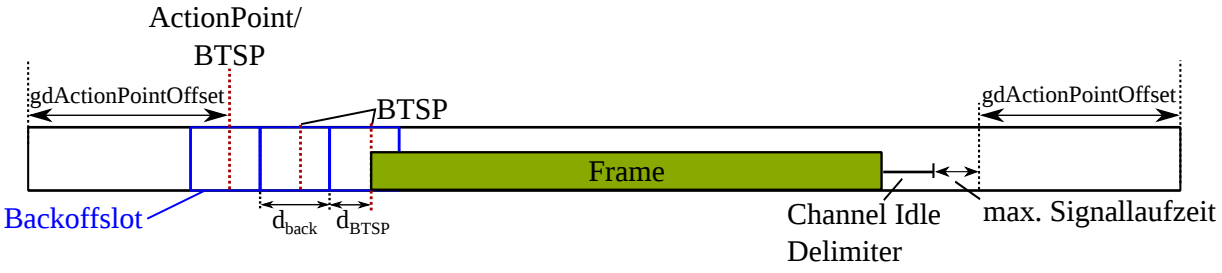


Abbildung 6.4: Aufbau eines modusbasierten statischen Slots (MSSs).

Der Parameter d_{back} gibt die Länge des Backoffslots an, während d_{BTSP} den Abstand zwischen dem BTSP und dem Beginn des Backoffslots quantifiziert. Durch die Deckungsgleichheit des BTSP des ersten Backoffslots sowie dem ActionPoint wird der Mode mit der höchsten *Modus-Präferenz* ohne Verzögerungen (im Vergleich zu einem regulären statischen Slot) übertragen. Die Überdeckung von BTSP und ActionPoint ist zulässig, da der ActionPointOffset bereits so gewählt werden muss, dass trotz einer maximalen Uhrenabweichung gewährleistet ist, dass alle Knoten den Übertragungsbeginn dem korrekten Slot zuordnen – ein Umstand, der ebenfalls für die korrekte Wahl des BTSP zwingend erforderlich ist. Da der ActionPointOffset durch den Designer – zum Zwecke einer höheren Robustheit – größer gewählt werden kann als unbedingt notwendig, ist es nur sinnvoll, für die Konfiguration des BTSP zu fordern, dass d_{BTSP} kleiner[24] als der ActionPointOffset gewählt werden muss.

6.3.2 Konfiguration der modusbasierten statischen Slots

Um die korrekte Funktionsweise von *Fast Mode-Signaling* zu garantieren, muss die Länge der Backoffslots ebenso wie der BTSP so gewählt werden, dass alle Knoten einen Übertragungsbeginn dem gleichen Backoffslot zuordnen. Ebenso muss die Anzahl der benötigten Backoffslots bei der Konfiguration der Länge der statischen Slots berücksichtigt werden.

Die Herleitung von Constraints zur Bestimmung der minimalen zulässigen Länge eines Backoffslots sowie der Positionierung des BTSP innerhalb der Backoffslots gestaltet sich sehr einfach, da letztendlich bei der Umsetzung von *Fast Mode-Signaling* innerhalb der MSSs wieder FTDMA als Ausgangsbasis für die Realisierung dienen kann. Konzeptuell besteht jeder MSS aus einem (individuellen) in sich abgeschlossenen *dynamischen Segment* mit genau einem modusbasierten Slot. Die Backoffslots entsprechen in diesem Fall wieder Minislots[25] und der BTSP dem MinislotActionPoint. Für Minislots sowie die Wahl des MinislotActionPoint formuliert die *FlexRay Protocol Specification* [Fle10b, Fle05a] jedoch bereits entsprechende Constraints. Die einzige Anpassung an diesen Constraints besteht daher darin, für Backoffslots und BTSP eigene Konfigurationsparameter einzuführen. Auch die Herleitung eines neuen Constraints für die minimale Länge eines MSSs ist auf dieser Grundlage leicht umzusetzen, da lediglich die Anzahl der benötigten Backoffslots und deren Länge zusätzlich berücksichtigt werden muss. Wir verzichten daher hier auf eine ausführliche Darstellung der Constraints und verweisen stattdessen auf den Anhang D.2.

6.3.3 Beispiel

Wir haben unser Werkzeug für die Erstellung von FlexRay-Konfigurationen aus Kapitel 6.2.1 um die in Anhang D.2 definierten Constraints für MSSs erweitert. Mit dessen Hilfe haben wir die Konfiguration des Kommunikationszyklus von BMW [BPS08] (vgl. Anhang D.1) so angepasst, dass innerhalb des statischen Segments *Mode-Based Scheduling with Fast Mode-Signaling* eingesetzt werden kann. Als Länge für die Backoffslots haben wir $gdBack = 2\,\text{MT}$ gewählt (MT – Makrotick), der BTSP wurde auf $gdBTSP = 1\,\text{MT}$ konfiguriert (die jeweils minimal zulässigen Größen, vgl. Anhang D.2). Wie die Ausgaben (Listing 6.1) zeigen, liefert das Werkzeug auch Vorschläge für die Anpassung der Konfiguration bzw. Implikationen, welche sich aus der aktuellen Wahl der Parameter ergeben. Das Listing 6.1 zeigt lediglich einen Auszug der Meldungen und beschränkt sich auf Informationen zu den MSSs.

Die Zeilen 4-8 in Listing 6.1 fassen die Informationen bzgl. der aktuell gewählten Konfiguration hinsichtlich Backoffslot, BTSP sowie maximaler Nutzdatenmenge zusammen. Implikationen hieraus, hinsichtlich der Mindestgröße der MSSs für die gewählte maximale Payload, in Abhängigkeit von der Anzahl abbildbarer Slot-Assignments, werden in den Zeilen 12-18 ausgegeben (Constraint D.4).

[24] Zumal ein großer Wert für d_{BTSP} den Overhead des *Fast Mode-Signaling* erhöht.

[25] Mit der Ausnahme, dass die Nummerierung der Backoffslots im Gegensatz zu den Minislots (und dynamischen Slotnummern) bei 0 beginnt und nur der Rahmen mit dem Mode mit der höchsten *Modus-Präferenz* übertragen wird.

Listing 6.1: Kommunikationszyklus unter Verwendung von *Mode-Based Scheduling with Fast Mode-Signaling* innerhalb des statischen Segments.

```
Konfigurationsparameter:
----------------------------------------------
[..]
BackoffSlots for modebased-Scheduling
----------------
 Laenge eines BackoffSlots (gdBack):  2 MT= 2.75 us
 Maximale Payload in einem modebased-Slot:  16 Bytes eingeplant
 Gewaehlter ActionPointOffset (gdBTSP):  1 MT
[..]
> # Output Modebased

Using Backoff slots for the static segment
----------------
 Minimal length of the static slot using 1 modes per static slot:  24 MT = 33 us
 Minimal length of the static slot using 2 modes per static slot:  26 MT = 35.75 us
 Minimal length of the static slot using 3 modes per static slot:  28 MT = 38.5 us
 Minimal length of the static slot using 4 modes per static slot:  30 MT = 41.25 us
 Minimal length of the static slot using 5 modes per static slot:  32 MT = 44 us

Assuming gdStatic consists of 24 MT = 33 us
----------------
 Mode 1 accepts as maximal payload  16 Bytes.
 Mode 2 accepts as maximal payload  12 Bytes.
 Mode 3 accepts as maximal payload  10 Bytes.
 Mode 4 accepts as maximal payload  8 Bytes.
 Mode 5 accepts as maximal payload  4 Bytes.
```

Ab Zeile 20 wird die maximal zulässige Nutzlast in Abhängigkeit von der Anzahl zulässiger Slot-Assignments für einen MSS mit einer vorgegebenen Länge bestimmt (Constraint D.4). In diesem konkreten Beispiel wurde die Länge der MSSs auf 24 MT gesetzt. Hiermit lassen sich pro MSS bis zu drei Slot-Assignments zuweisen, sofern sich die maximale Nutzlast auf 10 Bytes beschränkt (Zeile 24).

Ist dies nicht akzeptabel, können die statischen Slots von 24 MT auf 28 MT vergrößert werden (Zeile 16), um auch innerhalb der MSSs die maximale Nutzlast von 16 Bytes pro Rahmen nutzen zu können (bei maximal drei Slot-Assignments pro MSS). Eine dahingehend modifizierte FlexRay-Konfiguration, die ausschließlich statische Slots der Länge 28 MT verwendet, wird in Anhang D.3 dargestellt. Die dort abgedruckte Zusammenfassung der wichtigsten Konfigurationsparameter stammt direkt aus unserem Werkzeug. Die Prüfung aller Constraints des FlexRay-Standards sowie unserer Erweiterungen der Constraints haben keinen Fehler für diesen Satz von Konfigurationsparametern ergeben. Die Ausgaben unseres Werkzeuges finden sich in Auszügen im Anhang D.3. Der konfigurierte Kommunikationszyklus ist konform mit dem FlexRay-Standard, insofern, dass alle Slots des statischen Segments eine einheitliche Länge aufweisen (MSSs und ESSs).

6.3.4 Bewertung und Optimierungsmöglichkeiten

Die vorgestellte Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in das statische Segment ist natürlich nicht mit einem Standard CC realisierbar. Um als Sender in MSSs agieren zu können, werden spezielle modusbasierte CCs benötigt, welche die entsprechenden Protokollfunktionalitäten direkt integrieren. Dennoch sind Standard CCs – bei Verwendung identischer Längen für MSSs und ESSs – in der Lage, die in MSSs übertragenen Rahmen fehlerfrei zu empfangen und weiterzuverarbeiten. Ein Mischbetrieb ist somit störungsfrei möglich. Allerdings fehlt Standard CCs die Möglichkeit, den Mode aus dem Empfangszeitpunkt zu ermitteln. Da der Empfangszeitpunkt zudem nicht aufgezeichnet wird, kann der Mode auch nicht nachträglich durch den Host bestimmt werden. Die Entwicklung von CC, die die

modusbasierte Kommunikation in MSSs unterstützen, dürfte nicht allzu schwierig sein, da alle benötigten Funktionalitäten, Mechanismen und Konzepte im Wesentlichen auf FTDMA beruhen und die benötigte Logik daher ohnehin zur Verfügung steht. Etwas aufwändiger dürfte sich diesbezüglich die Bereitstellung einer geeigneten Pufferstruktur mit direkter Unterstützung von Modes erweisen (vgl. Kapitel 6.2.1.3).

Die in diesem Abschnitt vorgestellte Integration versucht die Anpassungen des FlexRay-Protokolls möglichst minimal zu halten, in der Hoffnung eine höhere Akzeptanz in Fachkreisen zu erreichen, da der Kern von FlexRay erhalten bleibt und *Mode-Based Scheduling with Fast Mode-Signaling* lediglich als neue zusätzliche Option zur Verfügung steht. Ist man bereit, weiterführende Modifikationen vorzunehmen, so kann die Bandbreitennutzung weiter optimiert werden: So eröffnet eine individuelle Konfiguration der Länge der einzelnen statischen Slots schon bei den klassischen ESSs Einsparmöglichkeiten bei stark heterogenen Nachrichtenlängen. Bei MSSs kann die optimale Länge eines Slots in Abhängigkeit von der Anzahl der für diesen Slot benötigten Backoffslots (Slot-Assignments) sowie der Länge der Nutzdaten gewählt werden. Nachteile bestehen durch einen hieraus resultierenden höheren Konfigurationsaufwand und höhere Aufwände, falls der Ablaufplan einmal modifiziert werden muss.

Eine weitere Option – die insbesondere bei sehr langen MSSs und kurzen Nutzdaten interessant ist – besteht darin, nicht nur den Mode mit der höchsten Präferenz, sondern noch weitere Rahmen mit niedrigerer *Modus-Präferenz* zu übertragen, damit die Bandbreite des Slots optimal ausgenutzt wird. Auf diese Weise würden die MSSs zu kleinen dynamischen Segmenten für eine eingeschränkte Gruppe von Knoten. Die entsprechenden Constraints für solche Konfigurationen leiten sich von den Constraints des dynamischen Segments ab. Auch wenn wir an dieser Stelle auf eine detaillierte Darstellung verzichten, bietet unser Konfigurationswerkzeug bereits eine Unterstützung für den Entwurf von Kommunikationszyklen mit dieser Betriebsvariante an.

6.4 Zusammenfassung und Ausblick

Wir haben in diesem Kapitel drei unterschiedliche Ansätze für die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in das FlexRay-Protokoll vorgestellt, die sich entweder ohne oder nur mit geringfügigen Modifikationen in der Protokoll-Spezifikation sowie funktionalen Erweiterungen der CC realisieren lassen. Die ersten beiden Varianten nutzen das dynamische Segment und dessen FTDMA-Mechanismus, wobei sich die erste Variante auf die Realisierung eines einzigen modusbasierten Slots pro dynamischem Segment beschränkt. Für diese haben wir die Funktionalität und Realisierbarkeit des präsentierten Konzeptes mit Hilfe eines Prototypen nachgewiesen. Hierbei stellte sich heraus, dass das Konzept zwar funktional tragfähig, die Implementierung mit einem handelsüblichen CC jedoch nicht praxistauglich möglich ist. In der zweiten Variante haben wir den Ansatz dahingehend modifiziert, mehrere modusbasierte Slots pro dynamischem Segment zu erlauben. Dies führt jedoch, je nach Anwendung, zu einem hohen maximalen Jitter der Nachrichten oder wird ineffizient und ist daher, aus Sicht des Autors, nicht besonders interessant für die Praxis. Die dritte Lösung realisiert *Mode-Based Scheduling with Fast Mode-Signaling* innerhalb des statischen Segments. Hierzu werden modusbasierte statische Slots (MSSs) eingeführt und das *Fast Mode-Signaling* mittels Backoffslots realisiert.

Für die letzte Variante haben wir, in Anlehnung an die *FlexRay Protocol Specification* [Fle10b, Fle05a], Constraints für die Wahl der Konfigurationsparameter der MSSs definiert, sodass die (deterministische) Funktionalität des *Fast Mode-Signaling* gewährleistet wird. Auch die Umsetzung im statischen Segment ist insofern mit dem FlexRay-Standard verträglich, dass handelsübliche CC weiterhin ohne Störungen im gleichen Cluster betrieben werden können. Die notwendigen Anpassungen der Protokollspezifikation für die Integration von MSS widerspricht nicht den grundlegenden Konzepten von FlexRay. Vielmehr baut es auf den bereits vorhandenen Strukturen und Mechanismen (wie z. B. FTDMA) auf und erweitert diese.

Mit der Integration von *Mode-Based Scheduling with Fast Mode-Signaling* wird FlexRay um ein flexibles, anpassbares Verfahren erweitert, welches die Bandbreitennutzung für seltene sporadische Nach-

richten deutlich verbessert, ohne hierbei auf deterministische Garantien verzichten zu müssen. Insofern stellt es eine ideale Ergänzung zu den exklusiven Reservierungen des statischen Segments und dem prioritätsgesteuerten dynamischen Segment dar. Die modusbasierte Kommunikation im statischen Segment eignet sich besonders für sicherheitskritische seltene sporadische Nachrichten, die hohe Anforderungen bzgl. der erlaubten Verzögerungen und Zuverlässigkeit stellen und deterministische Garantien benötigen. Diese Anforderungen können mit unserer Erweiterung gewährt werden, ohne einen Slot exklusiv reservieren zu müssen.

Der nächste Schritt besteht in der Entwicklung eines Prototypen eines CC zur nativen Unterstützung von *Mode-Based Scheduling with Fast Mode-Signaling* innerhalb des statischen Segments sowie dessen Evaluation. Da die aktuell für FlexRay 2.1a eingesetzten IP-Cores für CC nicht verfügbar sind, wären die hieraus resultierenden Aufwände allein für die Entwicklung eines standardkonformen CC bereits sehr hoch. Daher war eine Umsetzung im Rahmen dieser Arbeit nicht möglich. Der Vorteil einer vollständigen Neuentwicklung besteht andererseits darin, dass auch die diskutierten Optimierungen für das statische Segment in den CC integriert werden könnten.

Ebenfalls von Interesse, für weiterführende Forschungen, sind die Eigenschaften des Electrical Physical Layer [Fle06, Fle10a] von FlexRay. Dieser unterstützt vier Signalpegel (Idle_LowPower, Idle, High und Low), von denen die Pegel High und Low bei einer Überlagerung auf dem Bus dominant gegenüber den Idle-Pegeln sind. Hier stellt sich die Frage, ob auf Basis einer bitweisen Codierung der *Modus-Präferenzen* (z. B. High für eine logische Eins, Idle für eine logische Null) und einer Arbitrierung nach dem Vorbild von CAN eine effizientere Realisierung von *Fast Mode-Signaling* umsetzbar wäre. Bei einer Beschränkung der Geschwindigkeit dieses Arbitrierungsprozesses (vgl. Kapitel 3.1) könnte dann, nach dem Vorbild von CAN FD, mit unterschiedlichen Geschwindigkeiten bei der Arbitrierungs- und der Datenübertragungsphase gearbeitet werden.

Teil II

FERAL

Inhaltsverzeichnis für Teil II

7 Einführung in FERAL

Die Entwicklung und Bereitstellung echtzeitfähiger Kommunikationstechnologien ist zwar eine der grundlegenden Voraussetzungen für verteilte eingebettete Echtzeitsysteme, doch dies ist kein Garant für funktional fehlerfreie Systeme. Aufgrund der zunehmenden Komplexität eingebetteter Systeme in Kombination mit deren Einsatz in sicherheitskritischen Bereichen – z. B. Assistenzsysteme in modernen Autos oder Flugzeugen – werden Entwicklungsansätze benötigt, um diese Systeme so sicher und zuverlässig wie möglich zu gestalten. Dies ist besonders relevant, da sich dieser Trend in Zukunft eher noch verstärken wird, betrachtet man etwa aktuelle Forschungsbemühungen in Bezug auf autonome oder teilautonome Fahrzeuge.

Ein praxisrelevanter Lösungsansatz, um dies trotz der gestiegenen Komplexität zu erreichen, besteht in der Kombination modellgetriebener Entwicklungsansätze mit Virtual Prototyping. Die modellgetriebene Entwicklung ermöglicht zusammen mit Codegeneratoren die Erzeugung qualitativ hochwertigen Codes und erlaubt es, den Fokus stärker auf die Funktionalität statt auf die Implementierung zu lenken. Die Nutzung von Virtual Prototyping gestattet nicht nur den Test einzelner funktionaler Komponenten eines verteilten Echtzeitsystems, sondern zusätzlich auch den Test und die Evaluation der Interaktion der verteilten Komponenten in einer realistisch simulierten Systemumgebung. Neben der Simulation des Kommunikationsverhaltens (Verzögerungen, Verlust von Nachrichten, ...) werden auch die *externen* Stimuli des eingebetteten Systems durch die simulierte Umgebung – z. B. das physikalische Verhalten eines Automobils – nachgebildet. Dies beginnt bei simulierten Sensoreingaben und endet bei der Reaktion auf die Ansteuerung eines Aktuators und gestattet so Evaluation, Validierung und Tests unter Berücksichtigung des Verhaltens des Gesamtsystems und unter Bezugnahme auf die Systemumgebung.

Um Fehler so früh wie möglich erkennen und beheben zu können, sollte daher Virtual Prototyping bereits in den frühen Entwicklungsphasen angewendet werden. So gelingt es, Kosten sowie Risiken bei der Entwicklung komplexer System zu reduzieren [SBB+02]. Ein weiteres Anwendungsszenario ist die Evaluierung von Designalternativen in frühen Entwicklungsphasen. Anstatt sich auf Erfahrungen in Bezug auf wichtige Designentscheidungen – wie z. B. der Wahl einer geeigneten Kommunikationstechnologie – zu verlassen, können solche Entscheidungen und deren Auswirkungen über Simulationen durch Performance-Messungen quantifiziert und optimiert werden. Ebenso kann bei veränderten Anforderungen mit diesen Mitteln geprüft werden, ob Designentscheidungen noch tragfähig sind oder revidiert werden müssen.

Damit Virtual Prototyping den gesamten Entwicklungsprozess begleiten kann, muss das verwendete Framework in der Lage sein, Komponenten auf unterschiedlichen Abstraktionsstufen zu unterstützen: Im Idealfall, von einem ersten funktionalen Modell bis hin zur Simulation der finalen Hardwareplattform, auf der der (erzeugte) Code anschließend ausgeführt wird. Wird darüber hinaus die Interaktion einzelner Komponenten des zu simulierenden Systems unterschiedlicher Abstraktionsstufen ermöglicht, so können diese durch schrittweise Verfeinerung bis zur finalen Version entwickelt werden. So steht während allen Entwicklungsphasen, eine Plattform für die kontinuierliche Evaluation und Validierung zur Verfügung; Integrationstests lassen sich zu jedem Zeitpunkt durchführen, da stets das Gesamtsystem und dessen Verhalten simuliert werden kann. Die Unterstützung von Komponenten auf unterschiedlichem Abstraktionsniveau erlaubt auch den Umgang mit funktionalen Komponenten anderer Hersteller oder Lieferanten, welche unter Umständen nur als funktionales Modell oder als *Blackbox* Komponente bereitgestellt werden. Aber auch bei der regulären Entwicklung ist dies nützlich: So können einige Komponenten (zunächst)

vereinfacht als Platzhalter – mit oberflächlich implementierten Verhalten – vorgesehen werden, um sich auf die Entwicklung der Hauptkomponenten konzentrieren zu können.

Ein aktueller Trend, vor allem im Automobilbereich, besteht darin, standardisierte Hardware- und Anwendungsplattformen (wie z. B. AutoSAR [AUT14a]) zu nutzen, anstatt wie bisher abgeschlossene Einheiten mit speziell zugeschnittener Hard- und Software zu entwickeln und zu vertreiben. Dieser Ansatz erlaubt es, komplexe Gesamtsysteme (wie z. B. Autos) verstärkt als Ausführungsplattform zu sehen, bestehend aus einer generischen Hardware-, einer Anwendungsplattform und hierauf aufbauenden Funktionen/Anwendungen. Funktionalitäten – wie etwa Assistenzsysteme oder Komfortfunktionen – werden dann in Form von Anwendungen bereitgestellt. Durch diese Standardisierung eröffnet sich den Zulieferern ein größerer Markt an potentiellen Abnehmern, während sich die Abnehmer von der größeren Konkurrenz gleichzeitig Kostenersparnisse erhoffen. Für die Entwicklung bedeutet dies jedoch, dass solche Anwendungen mit einer sehr viel größeren Varianz von Konfigurationen und Umgebungen zurechtkommen müssen. Auch hier stellt Virtual Prototyping einen Ansatz dar, Aufwand und Kosten für Tests und Validierungen zu reduzieren.

Frameworks für Virtual Prototyping

Betrachtet man den aktuellen Stand der Forschung und Entwicklung (vgl. Kapitel 13), so existiert bereits eine Vielzahl von Simulatoren, die ausgewählte Aspekte eines virtuellen Prototypen oder eingebetteten Systems mit sehr hoher Genauigkeit simulieren können. Mathworks Matlab Simulink [Mat12] gestattet beispielsweise sowohl die modellbasierte Entwicklung funktionaler Komponenten als auch deren Simulation. Andererseits erlauben Netzwerksimulatoren, wie der *Network Simulator 3* (ns-3, [ns3ar]), die exakte Simulation von Netzwerken. SystemC sowie die entsprechenden Werkzeuge (z. B. Synopsis [Rey13]) hingegen ermöglichen taktgenaue Simulationen von Hardwareplattformen. Diesen Ansätzen ist gemein, dass sie sich jeweils nur auf einen kleinen Aspekt des Gesamtverhaltens beschränken, diesen aber dafür mit sehr hoher Genauigkeit abbilden. Um jedoch Virtual Prototyping mit den vorgestellten Absichten zu betreiben, wird ein ganzheitlicher Ansatz benötigt, der in der Lage ist, alle Aspekte eines komplexen verteilten eingebetteten Echtzeitsystems mit hinreichend großer Genauigkeit sowie deren Zusammenwirken zu simulieren. Denn gerade aus dem Zusammenspiel spezieller Aspekte können gegenseitige Wechselwirkungen resultieren, die – unter Umständen – zu schwer identifizierbaren Fehlern führen. Die Entwicklung eines einzigen Simulators für alle Aspekte komplexer (heterogener) Systeme ist kaum zu leisten, insbesondere wenn dieser nicht auf bestimmte Anwendungsszenarien festgelegt sein soll. Daher besteht unser Ansatz in der Entwicklung eines offenen Frameworks für die Kopplung bereits existierender spezialisierter Simulatoren. Durch die Kopplung unterschiedlichster Simulatoren kann das Verhalten und die Interaktion komplexer Gesamtsysteme simuliert werden, ohne Einschränkungen auf einen bestimmten Aspekt.

Durch die Bereitstellung eines solchen Frameworks für die Kopplung von Simulatoren sind die Aufwände für die Integration der spezialisierten Simulatoren nur ein einziges Mal zu tragen. Nach der Integration können diese in beliebigen Simulationen eingesetzt und miteinander kombiniert werden. Die größte Herausforderung bei der Kopplung der Simulatoren liegt darin, dass diese häufig auf unterschiedlichen Ausführungsmodellen beruhen (z. B. zeit- vs. ereignisgetriggert). Daher besteht eine zentrale Aufgabe eines Frameworks zur Simulatorkopplung in der Bereitstellung von Mechanismen, um unterschiedliche Ausführungsmodelle semantisch korrekt miteinander zu verbinden, sodass die Simulatoren und deren simulierte Komponenten interagieren können. Hierzu gehört sowohl die Sicherstellung der Synchronisation, die Definition von Interaktionspunkten sowie die Schaffung von Möglichkeiten für den Datenaustausch zwischen den Simulatoren. Nur so kann die Korrektheit der Gesamtsimulation gewährleistet werden. Da Erweiterbarkeit ein zentrales Konzept darstellt, muss das Framework so entworfen werden, dass die Integration neuer Simulatoren und Ausführungsmodelle so einfach wie möglich ausfällt.

In der Literatur finden sich verschiedene Ansätze zur Kopplung von Simulatoren, diese sind aber stark auf ein konkretes Szenario zugeschnitten und beschränken sich auf die Kopplung von zwei oder drei ausgewählten Simulatoren. Die Lösungen sind in diesen Fällen sehr problemspezifisch und eignen sich nicht als Basis für eine allgemeinere Vorgehensweise. Einige Simulatoren bieten zwar proprietäre Schnittstellen, die eine Anbindung anderer Simulatoren ermöglichen, allerdings ist die Akzeptanz solcher Insellösungen ebenso beschränkt wie deren Einsatzbereich. Häufig ist die Schnittstelle jedoch auf Produkte des gleichen Herstellers beschränkt, sodass auch hier nur bestimmte (durch den Hersteller unterstützte) Szenarien abgebildet werden können.

Da zum Zeitpunkt unserer Recherchen kein Framework für die Simulatorkopplung existierte, welches die beschriebenen Anforderungen hinsichtlich Erweiterbarkeit und Integrationsfähigkeit vereinte, starteten wir 2010, im Rahmen des Innovationszentrums Applied System Modeling des Fraunhofer-Instituts für Experimentelles Software-Engineering IESE und der TU Kaiserslautern, die Entwicklung von FERAL. FERAL – unser **F**ramework for **E**fficient simulator coupling on **R**equirements and **A**rchitecture **L**evel – ermöglicht die Kopplung spezialisierter Simulatoren – und hierüber die Simulation der funktionalen Komponenten eines verteilten Systems sowie dessen Systemumgebung. Ähnlich wie das Simulationsframework Ptolemy[1] unterscheidet FERAL zwischen Simulatoren, die als eigenständige Simulationskomponenten in FERAL integriert werden, und Ausführungsmodellen (*Execution Models*), welche in Form von *Direktoren* (*Directors*) abgebildet sind. Die Direktoren haben die Aufgabe, die Ausführung von Simulationskomponenten zu steuern und deren Austausch von Nachrichten zu koordinieren.

Simulationskomponenten integrieren Simulatoren, indem diese als Bindeglied zwischen Simulator und Direktor dienen und dem zuständigen Direktor (indirekt) die Kontrolle über den Simulator und dessen Ausführung ermöglichen. Da die Simulatoren unabhängig voneinander entwickelt wurden, folgen diese unterschiedlichen Paradigmen hinsichtlich ihres *Model of Computation and Communication* (MOCC). Das MOCC legt die Ausführungssemantik (z. B. zeit- oder ereignisgetriggert) und die Zeitpunkte fest, zu denen ein Datenaustausch bzw. eine Interaktion mit anderen Simulationskomponenten semantisch zulässig[2] ist. Unterschiedliche Direktoren implementieren unterschiedliche MOCCs und verwalten jeweils nur Simulationskomponenten (und somit Simulatoren) desselben MOCCs. Zu den Aufgaben der Direktoren gehört sowohl deren Ausführung als auch die Realisierung des Austauschs von Nachrichten mit anderen Simulationskomponenten entsprechend des MOCCs, da sich nur durch dieses Zusammenspiel eine semantisch korrekte Simulation ergibt. Zusätzlich übernimmt FERAL auch die Synchronisation der nebenläufig ausgeführten Simulationskomponenten (Simulatoren) und Direktoren. Über die generischen Schnittstellen und Datentypen, die FERAL anbietet, können beliebige Simulatoren miteinander interagieren, und zwar unabhängig von deren MOCC. Um Simulatoren mit unterschiedlichen MOCCs in einem Simulationssystem einsetzen zu können, erlaubt FERAL die Schachtelung von Direktoren. Hierbei stellt FERAL die semantische Korrektheit der Simulation sicher. Um den Aufwand für die Integration neuer Simulatoren zu reduzieren, stellt FERAL mehrere Direktoren für unterschiedliche Ausführungsmodelle bereit, die unter anderem auch als Vorlage dienen können.

Vorgehen und Abgrenzung

Die Entwicklung der Basiskonzepte von FERAL (Direktoren, Schnittstellen, Nachrichtentypen) wurde im Wesentlichen von Thomas Kuhn im Rahmen der Kooperation mit dem Fraunhofer IESE als Teil des Innovationszentrums für *Applied Systems Modeling* vorangetrieben und baut sowohl auf dem Simulationsframework Ptolemy [EJL+03] als auch seinen eigenen Forschungen im Bereich der Simulatorkopplung [Kuh09] auf. Insofern ist Kapitel 7.1 als Grundlagenkapitel zu verstehen, welches den bereits existierenden Stand der Technik zu Beginn dieser Arbeit beschreibt und beschränkt sich daher auf eine Zusammenfassung der Grundlagen von FERAL, sofern diese für das weitere Verständnis erforderlich sind.

[1] Eine Abgrenzung zwischen Ptolemy und FERAL ist in Kapitel 7.1.2 sowie Kapitel 13 zu finden.

[2] Bei zeitgetriggerten Simulationskomponenten wäre dies z. B. nur nach dem Abschluss eines Simulationsschrittes der Fall.

Für tiefer gehende Fragestellungen, insbesondere auch in Bezug auf die Korrektheit der Kopplung von MOCCs, sei daher auf die Publikation [BGFK13] verwiesen. Die in den Kapiteln 8 bis 12 beschriebenen Ergebnisse wurden im Rahmen dieser Dissertation entwickelt. Teilweise gab es im Laufe der Arbeit kleinere Kooperationen mit Kollegen, deren Beiträge sind innerhalb der jeweiligen Kapitel gesondert kenntlich gemacht. Die Ergebnisse dieser Arbeit wurden in [4, 5, 7, 10, 12] publiziert.

Nach der Beschreibung der grundlegenden Konzepte von FERAL in diesem Kapitel, widmet sich das **Kapitel 8** der Kopplung von Simulatoren zur Abbildung des funktionalen Verhaltens von Systemkomponenten. Um verteilte Systeme simulieren zu können, haben wir zusätzlich eigene Simulatoren für aktuelle Kommunikationstechnologien für FERAL entwickelt und integriert (**Kapitel 9**), wobei hier der Schwerpunkt auf etablierten Feldbussen im Automobilbereich lag. **Kapitel 10** stellt die von uns entwickelten Konzepte und Techniken für die Evaluation von Designalternativen in Bezug auf unterschiedliche Kommunikationstechnologien vor. Anschließend demonstrieren wir in **Kapitel 11** den Einsatz von FERAL bei der Entwicklung eines verteilten *Adaptive Cruise Control Systems*, einem Assistenzsystem für Autos, welches Tempomat, automatisches Abstandskontrollsystem sowie Bremsassistent kombiniert. Abschließend präsentieren wir einen spezialisierten Ansatz für Hardware-in-the-Loop Simulationen mit FERAL (**Kapitel 12**), welcher insbesondere die letzte Entwicklungsphase sowie Integrationstests auf der Zielplattform unterstützt. **Kapitel 13** fasst den aktuellen Stand der Forschung und Technik zusammen, **Kapitel 14** liefert eine Zusammenfassung sowie einen Ausblick auf weitere mögliche Forschungsthemen.

7.1 Technische Grundlagen von FERAL

FERAL ist unser Java-basiertes Framework für die effiziente Kopplung spezialisierter Simulatoren. Diese werden in FERAL, in Form von Simulationskomponenten eingebettet, die als Bindeglied zwischen dem Simulator und FERAL agieren. Die Ausführung der Simulationskomponenten (bzw. der von diesen repräsentierten Simulatoren) wird durch die von FERAL bereitgestellten Direktoren kontrolliert, die zusätzlich die Aufgabe haben, die semantische Korrektheit der Gesamtsimulation sicherstellen.

Durch eine lose Kopplungsstruktur der Simulationskomponenten (vgl. Kapitel 7.1.1) können beliebige Simulatoren instanziiert und flexibel zu komplexen Simulationssystemen zusammengestellt werden, um unterschiedlichste Anwendungsszenarien abzubilden. Die Kombination und Zusammenarbeit verschiedener spezialisierter Simulatoren erlaubt es, alle relevanten Teilaspekte bei der Evaluation eines komplexen Systems zu simulieren: So kann eine Simulation neben dem rein funktionalen Verhalten der einzelnen (Software-)Komponenten (Funktionen) eines verteilten Systems auch die Kommunikation (Verzögerungen, Übertragungsfehler etc.) bis hin zur Simulation der physikalischen Umwelt (z. B. die Reaktionen von Einwirkungen auf Basis mathematischer Modelle) umfassen. Auf diese Weise lassen sich etwa komplexe verteilte Regelungssysteme vollständig virtuell entwickeln, evaluieren und validieren, sodass auch Fehler in der Interaktion der einzelnen Komponenten oder resultierende Probleme (z. B. bzgl. der Regelgüte) bereits früh entdeckt werden können.

Die einheitlichen Schnittstellen von FERAL für den Datenaustausch zwischen den Simulationskomponenten gestatten es, Simulationskomponenten auf unterschiedlichen Abstraktionsniveaus zusammen innerhalb eines Simulationssystems einzusetzen und ermöglicht deren Interaktion. So können während des Projektverlaufs die einzelnen Simulationskomponenten schrittweise verfeinert bzw. ersetzt werden. Auf diese Weise steht während der gesamten Entwicklung (beginnend bei den frühen Entwicklungsphasen) ein ausführbares Simulationssystem zur Verfügung, welches kontinuierliche (Integrations-)Tests erlaubt. Hierdurch können Fehler früh identifiziert und Kosten und Entwicklungsrisiken reduziert werden (vgl. [SBB+02]). Zudem erlaubt der Einsatz von FERAL in frühen Entwicklungsphasen dessen Nutzung für die Evaluation von Designalternativen. So können Entscheidungen auf Basis objektiver Fakten getroffen werden, die aus Simulationen und damit verbundenen Performance-Bewertungen stammen.

7.1.1 Konzepte und Architektur von FERAL

Die Kopplung mehrerer Simulatoren zu einem einzigen zusammenhängenden Simulationssystem ist – ohne ein geeignetes Framework – eine sehr anspruchsvolle und zeitaufwändige Aufgabe, wie entsprechende Forschungsarbeiten belegen (vgl. Kapitel 13). FERAL hat es sich zum Ziel gesetzt, diesen Vorgang erheblich zu vereinfachen. Ein Vorteil eines generischen Kopplungsframeworks besteht darin, dass die Aufwände für die Integration eines spezialisierten Simulators nur ein einziges Mal anfallen. Existiert einmal eine Integration in Form einer speziell für den Simulator entwickelten Simulationskomponente für FERAL, so kann dieser für beliebige Simulationsszenarien herangezogen werden, ohne dass weitere Anpassungen erforderlich sind.

Um die Integration beliebiger Simulatoren zu ermöglichen, definiert FERAL die folgenden Typen und Schnittstellen: `SimulationComponent` ist der Basistyp für alle von FERAL ausgeführten Komponenten. Diese unterteilen sich in `Director`s und `ConcreteSimulationComponent`s, wie in Abbildung 7.1 dargestellt. Entsprechend dem Composite-Pattern [GHJV09] bilden die `ConcreteSimulationComponent`s die Blätter innerhalb des so repräsentierten Komponentenbaums und repräsentieren die integrierten Simulatoren. Die Direktoren bilden die inneren Knoten und sind für die Ausführung der ihnen untergeordneten `SimulationComponent`s verantwortlich. Bei diesen kann es sich wiederum selbst um Direktoren oder einen instanziierten Simulator (in Form einer `ConcreteSimulationComponent`) handeln. Direktoren etablieren eine semantische Domäne, die festlegt, nach welchem Ausführungsmodell die untergeordneten `SimulationComponent`s ausgeführt werden und zu welchen Zeitpunkten und nach welchem Kommunikationsmodell der Austausch von Nachrichten mit anderen `SimulationComponent`s erfolgt. Das heißt, jeder Direktor implementiert sein eigenes MOCC, welches für die ihm direkt untergeordneten `SimulationComponent`s verbindlich ist.

Die Kommunikation zwischen `SimulationComponent`s – und somit den von ihnen repräsentierten Simulatoren – erfolgt über unidirektionale Ports: `InputPort`s und `OutputPort`s, welche über `Link`s miteinander verbunden sind. Jedem `InputPort` ist darüber hinaus ein `Receiver` zugeordnet, welcher das semantische Verhalten des `InputPort`s festlegt und durch den übergeordneten Direktor der Komponente instanziiert wird. So ist sichergestellt, dass die durch den Receiver implementierte Semantik

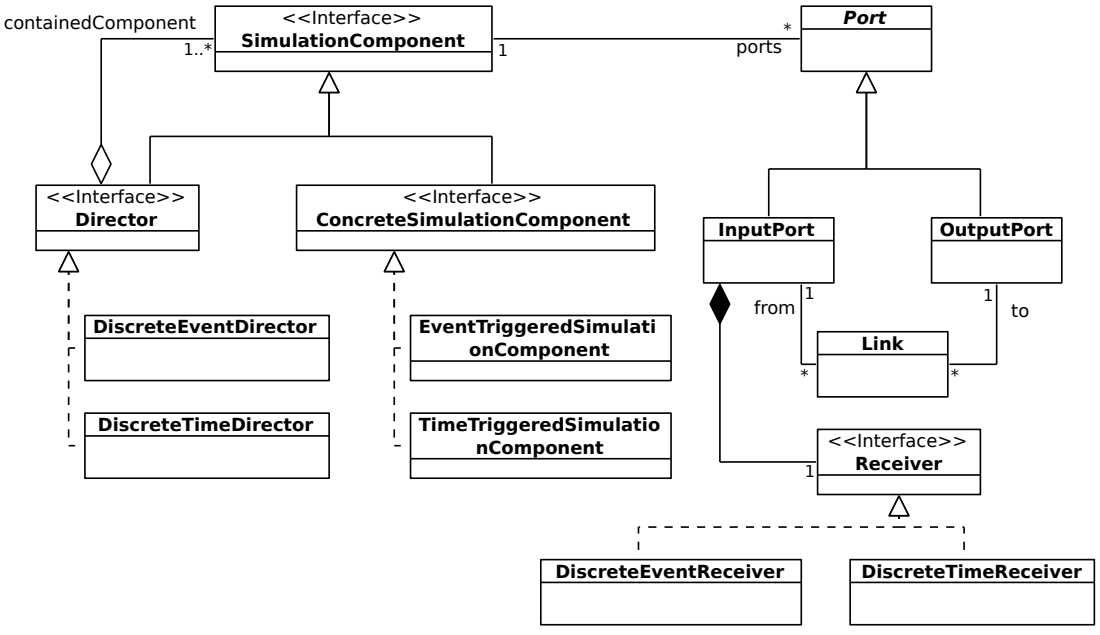


Abbildung 7.1: UML-Diagramm der Basisklassen von FERAL.

mit dem MOCC des Direktors kompatibel ist. Das Verhalten der Receiver reicht dabei von einfachen einelementigen Nachrichtenpuffern (zeitgetriggerte Semantik) bis zu einer geordneten Warteschlange (ereignisgetriggerte Semantik).

Die Interaktion zwischen den einzelnen `SimulationComponent`s erfolgt asynchron über den Austausch von Nachrichten. Die Links leiten die Nachrichten als Kopie von OutputPorts zu InputPorts weiter. Der Zeitpunkt, zu dem Nachrichten weitergeleitet werden, wird durch den jeweiligen Direktor entsprechend dem MOCC festgelegt, sodass eine semantisch korrekte Ausführung sichergestellt wird (vgl. [BGFK13]). Aus dem gleichen Grund werden Links, welche über die Grenze eines Direktors hinweg Ports von Simulationskomponenten miteinander verbinden, an den Grenzen der jeweiligen Direktoren aufgespalten und sogenannte Proxy-Ports eingefügt. Diese erlauben es den Direktoren sicherzustellen, dass auch hier die Weiterleitung und Zustellung der Nachrichten in Übereinstimmung mit dem MOCC erfolgt.

Die Abbildung 7.1 zeigt neben der allgemeinen Struktur der Direktoren, den Simulationskomponenten und den konkreten Simulationskomponenten, auch jeweils zwei unterschiedliche Implementierungen von Direktoren und Receivern. Der `DiscreteEventDirector` implementiert zusammen mit dem `DiscreteEventReceiver` ein ereignisbasiertes MOCC. Die Klasse `EventTriggeredSimulationComponent` kann hingegen als Vorlage für die Integration eines Simulators mit einem solchen MOCC dienen. Analog gilt dies in Bezug auf eine zeitgetriggerte Semantik für den `DiscreteTimeDirector`, dessen Receiver `DiscreteTimeReceiver` sowie die `TimeTriggeredSimulationComponent`. Letztere diente z. B. als Ausgangspunkt für die Integration eines Simulators für Matlab Simulink-Modelle (vgl. Kapitel 8.3).

Damit beliebige Simulatoren interagieren können, definiert FERAL ein eigenes Typsystem für Nachrichten und Datentypen. Alle Nachrichten, die zwischen den `SimulationComponent`s über Links ausgetauscht werden, müssen hierfür die generische `Message`-Schnittstelle implementieren. FERAL stellt mit den `ValueMessage`s und `TimedValueMessage`s[3] bereits einfache Nachrichtentypen zur Verfügung, welche die grundlegenden Anforderungen für den Datenaustausch erfüllen und nach Möglichkeit genutzt werden sollten. Jede Nachricht kann Nachrichtendaten transportieren, sodass sich die Simulatoren bzw. `SimulationComponent`s nicht nur über Ereignisse informieren, sondern auch (komplexere) Daten, wie z. B. Simulationsergebnisse, austauschen können. Für die Kodierung der Nachrichtendaten stellt FERAL ein eigenes Typsystem bereit. Diese Nachrichtendatentypen implementieren die `MessageData`-Schnittstelle und stellen Klassen für die Repräsentation elementarer und komplexer Datentypen innerhalb von FERAL bereit. Über die Implementierung der `MessageData`-Schnittstelle lassen sich aber auch spezifische Nachrichtendatentypen ergänzen. So verwendet etwa der CAN-Simulator einen eignen Nachrichtendatentyp für die Repräsentation von CAN-Rahmen (vgl. Kapitel 8.1, Abbildung 8.2). Die Anpassung von Nachrichten- sowie Nachrichtendatentypen ist Bestandteil der Adaption von FERAL, die für die Integration neuer Simulatoren ggf. erforderlich ist (vgl. Kapitel 8.1).

Das UML-Diagramm 7.2 zeigt die von FERAL bereitgestellten Nachrichtentypen sowie eine Auswahl von Nachrichtendatentypen[4]. An den Namen ist bereits erkennbar, dass beispielsweise `BooleanData` als Container für den Java-Typ `boolean` bzw. `Boolean` fungiert. Entsprechendes gilt für die anderen primitiven Java-Datentypen. Die Klasse `StructuredData` erlaubt die Abbildung komplexerer Datentypen auf der Basis einer assoziativen Wert-Schlüssel-Zuordnung.

[3] Die `TimedValueMessage` speichert zusätzlich den Zeitpunkt der Nachrichtenerzeugung.
[4] Wir beschränken uns hier auf einige wenige Beispiele für Nachrichtendatentypen.

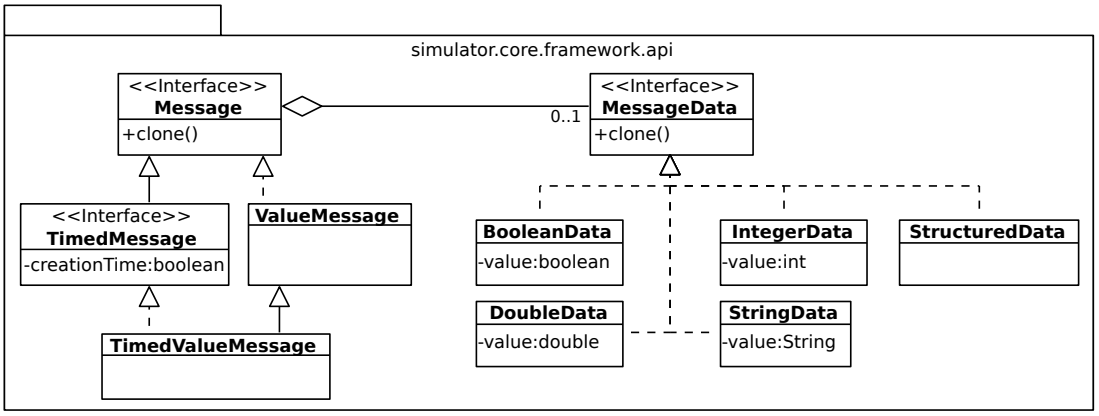


Abbildung 7.2: UML-Klassendiagramm des Nachrichtentypsystems von FERAL.

7.1.2 FERALs Simulationsmodell

Der von uns gewählte Ansatz für die Integration von Simulatoren sowie deren Ausführungsmodellen greift die Konzepte des Simulationsframeworks Ptolemy für die Integration unterschiedlicher MOCCs auf. Ptolemy strukturiert ein Simulationssystem in semantische Domänen und Akteure. Jede semantische Domäne wird durch einen Direktor kontrolliert, welcher ein spezifisches Ausführungs- und Kommunikationsmodell (MOCC) implementiert. Die Akteure (in FERAL vergleichbar mit den `Concrete-SimulationComponents`) implementieren das eigentliche Verhalten des simulierten Systems (z. B. spezifische Funktionalitäten). Die Ausführung der Akteure innerhalb einer Domäne erfolgt durch den für den Akteur zuständigen Direktor. Ptolemy unterstützt die Verwendung unterschiedlicher semantischer Domänen innerhalb eines Simulationssystems über die Schachtelung semantischer Domänen. Dies wird über die Schachtelung der entsprechenden Direktoren abgebildet. FERAL übernimmt die von Ptolemy eingeführten Konzepte hinsichtlich der Akteure, den semantischen Domänen mit ihren Direktoren, den Ports, Links und Receivern (vgl. [EJL+03]).

In FERAL besteht ein Simulationssystem aus (geschachtelten) Simulationskomponenten, bei denen es sich entweder um Direktoren oder konkrete Simulationskomponenten handelt. Die Direktoren kontrollieren – wie bei Ptolemy – die Ausführung der ihnen untergeordneten Simulationskomponenten und definieren über das von ihnen implementierte MOCC eine semantische Domäne. Konkrete Simulationskomponenten ähneln den Akteuren insofern, dass diese als Träger des Verhaltens innerhalb der Simulation fungieren. Doch anstatt das Verhalten direkt zu implementieren[5], greifen die konkreten Simulationskomponenten in FERAL auf Simulatoren zurück und dienen somit lediglich als Schnittstelle zu diesen Simulatoren. Die Simulatoren führen ihrerseits Verhaltensmodelle aus (z. B. ein Matlab Simulink-Modell) und reproduzieren auf diese Weise das innerhalb der Modelle beschriebene Verhalten in FERAL. Sowohl bei Ptolemy als auch FERAL bilden die Akteure bzw. konkreten Simulationkomponenten die Blätter innerhalb eines Komponentenbaums eines Simulationssystems. Die Kommunikation zwischen ihnen wird über Ports und Links abgebildet, wobei das Verhalten des Receivers durch den jeweiligen Direktor kontrolliert und festgelegt wird.

Die Abbildung 7.3 zeigt FERALs `SimulationComponent`-Schnittstelle. Auch bei der Definition der Schnittstelle übernimmt FERAL viele der von Ptolemy definierten Kontrollprimitiven. So erlaubt die Methode `preInit` die lokale Initialisierung der Simulationskomponente (SC). Die Methode `init` erlaubt komponentenübergreifende Initialisierungen, da diese erst aufgerufen wird, wenn bereits alle SC erzeugt wurden.

[5] Auch eine direkte Implementierung des funktionalen Verhaltens in Form einer nativen Simulationskomponente ist bei FERAL möglich, wird jedoch meist nur für kleine Komponenten und zu Prototyping-Zwecken genutzt (vgl. Kapitel 8.2).

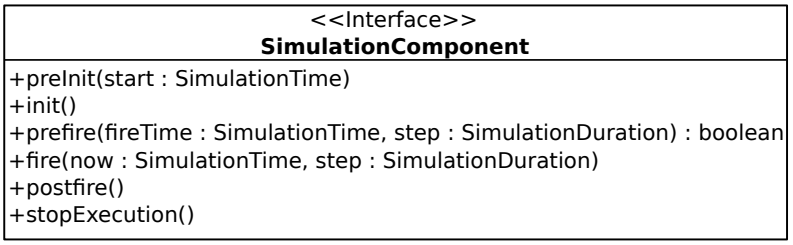


Abbildung 7.3: Die `SimulationComponent`-Schnittstelle von FERAL.

Über die Methode `prefire` (die vor der Methode `fire` aufgerufen wird) kann der Direktor anhand des Rückgabewertes prüfen, ob die SC gerade ausgeführt werden muss. Die `fire`-Methode veranlasst schließlich die Ausführung des Verhaltensmodells, verändert jedoch nicht den nach außen sichtbaren Zustand der SC, d.h., die nach außen hin sichtbaren Werte der OutputPorts der SC bleiben unverändert. Der Direktor beendet einen Ausführungsschritt mit dem Aufruf der `postfire`-Methode. Dies führt zu einer Aktualisierung des nach außen hin sichtbaren Zustands der Simulationskomponente, indem die erzeugten Nachrichten der jeweiligen OutputPorts über Links an die Receiver der verbundenen InputPorts übergeben werden. Durch Aufrufe der entsprechenden Methoden und die Kontrolle über die Receiver der InputPorts können die Direktoren sowohl die Ausführung der ihnen untergeordneten Simulationskomponenten als auch deren Kommunikationsverhalten mit anderen Komponenten steuern.

Während sich die Kernkonzepte von FERAL und Ptolemy stark ähneln, weicht das Ausführungsmodell von FERAL von dem von Ptolemy ab, sodass die speziellen Anforderungen bei der Kopplung von Simulatoren besser abgebildet werden können. Bei Ptolemy werden alle Simulationskomponenten/Akteure synchron ausgeführt und teilen eine globale Simulationszeit. Auf diese Weise liefert die Simulation sehr exakte Ergebnisse. Die für die sehr enge Kopplung notwendige Synchronisation geht jedoch zu Lasten der Simulationsgeschwindigkeit. Ein ähnlicher Ansatz würde bei FERAL, im Vergleich zu Ptolemy, noch größere Kosten verursachen, da es sich bei den Trägern der Systemaktivität nicht um eingebettete Akteure handelt, sondern externe Simulatoren (durch Kommunikationsmechanismen) in die Synchronisation einbezogen werden müssten. Aus diesem Grund erweitert FERAL das Ausführungsmodell von Ptolemy um die Möglichkeit einer temporalen Entkopplung der Simulationskomponenten, mit dem Ziel, so eine höhere Performance bei der Simulation zu erreichen. Hierzu verwaltet in FERAL jede Simulationskomponente ihre eigene lokale Simulationszeit, die von der Simulationszeit anderer Komponenten abweichen kann und darf. Die maximale Abweichung, welche die Genauigkeit der Simulation beeinflusst, wird durch regelmäßige Synchronisationen durch die Direktoren von FERAL begrenzt.

FERAL realisiert dieses Konzept durch die Einführung von Aktivitätsperioden, mit deren Hilfe der Grad der Entkopplung vorgegeben wird. Innerhalb einer Aktivitätsperiode kann eine Simulationskomponente beliebig Berechnungen durchführen oder Ereignisse verarbeiten, ohne hierbei die Kontrolle an den ausführenden Direktor zurückzugeben. Die Länge der Aktivitätsperiode wird durch den Direktor vorgegeben und den `prefire` und `fire`-Methoden beim Aufruf als Parameter übergeben. Hierbei gibt der erste Parameter `now` den Zeitpunkt an, zu dem die Ausführung der Simulationskomponente startet, der zweite Parameter die Dauer der Aktivitätsperiode. Während der Ausführung der `fire`-Methoden ändert sich weder der Zustand der InputPorts noch werden generierte Nachrichten an andere Simulationskomponente weitergereicht, d.h., der extern sichtbare Zustand einer Simulationskomponente bleibt die gesamte Aktivitätsperiode unverändert.

Da es zwischen den Simulationskomponenten innerhalb der jeweiligen Aktivitätsperioden zu keiner Interaktion kommt, können diese unabhängig voneinander und nebenläufig ausgeführt werden. Eine Synchronisation ist nur am Ende der Aktivitätsperioden notwendig, sodass der Aufwand reduziert und eine effizientere Simulation erreicht wird. Allerdings kann die seltenere Synchronisation und Interaktion

zu Ungenauigkeiten und abweichenden Ergebnissen oder Verhalten im Vergleich mit dem realen System führen – wobei die Wahrscheinlichkeit mit längeren Aktivitätsperioden steigt. Da die Länge der Aktivitätsperioden durch den Designer konfiguriert wird, kann dieser einen geeigneten Tradeoff zwischen Genauigkeit und Effizienz wählen. Die Wahl ist unter anderem abhängig von der Entwicklungsphase, den zu beobachtenden Effekten, der angestrebten Simulationsdauer sowie der benötigen Geschwindigkeit der Simulation.

Insbesondere darf der Designer auch unterschiedliche Aktivitätsperioden innerhalb eines Simulationssystems[6] definieren. Auf diese Weise lassen sich wichtigere Simulationskomponenten mit einer höheren Genauigkeit simulieren als weniger relevante. So lässt sich das Tempo der Simulation erhöhen, ohne an relevanter Genauigkeit zu verlieren. Eine ausführliche Erklärung, in Bezug auf die Sicherstellung der semantischen Korrektheit der Simulation bei der Kopplung unterschiedlicher MOCCs in Verbindung mit Aktivitätsperioden, ist in [BGFK13] zu finden.

Zusätzlich zu den Aktivitätsperioden unterstützt das zeitgetriggerte MOCC auch die Definition von Schrittweiten zur Festlegung von Verarbeitungszeiten. Die Schrittweite ist ein Parameter zeitgetriggerter Direktoren und gibt die Dauer eines Simulationsschrittes (simulierte Zeitspanne, die bei einem Aufruf von `fire` verstreicht) bei einem zeitgetriggerten MOCC vor. Auf diese Weise kann eine Aktivitätsperiode innerhalb eines zeitgetriggerten Direktors weiter unterteilt werden und so beispielsweise ein Simulink-Modell (bzw. die `fire`-Methode des Simulators) mit einer festen Periode (unabhängig von der Aktivitätsperiode) ausgeführt werden. Ein solcher Ausführungsschritt wird als atomar betrachtet, d.h., dieser wird entweder vollständig innerhalb der aktuellen Aktivitätsperiode des Direktors ausgeführt oder muss in eine nachfolgende Aktivitätsperiode verschoben werden. Hierdurch stellt FERAL sicher, dass die lokale Simulationszeit untergeordneter Direktoren und konkreter Simulationskomponenten nicht die Aktivitätsperioden übergeordneter Simulationskomponenten überschreiten und somit die maximale Abweichung durch die vorgegebenen Aktivitätsperioden begrenzt bleibt.

7.2 Strukturierung von FERAL

Um die modulare Struktur und Natur von FERAL auch innerhalb des Java-Codes abbilden zu können, setzen wir OSGi [OSG14] ein, welches Java um ein Komponentenmodell erweitert. Von besonderer Bedeutung für unsere Anwendung ist die Möglichkeit, explizit die öffentliche Schnittstelle von Komponenten zu definieren sowie das automatische Auflösen von Abhängigkeiten und Nachladen benötigter Komponenten auf Basis dieser Schnittstelle durch eine zentrale Komponenten-Registry. Zudem erzwingt OSGi die Einhaltung elementarer Designkonzepte eines modularen Aufbaus, wie etwa das Verbot zyklischer Abhängigkeiten zwischen Komponenten.

Eine in sich abgeschlossene Komponente wird in OSGi als Bundle bezeichnet. Bundles selbst lassen sich durch sogenannte Bundle Fragmente erweitern und können ihrerseits andere Bundles (basierend auf deren öffentlichen Schnittstellen) verwenden. Ein Bundle Fragment stellt die eigenen Bestandteile und Inhalte dem übergeordneten Bundle zur Laufzeit zur Verfügung. Die von FERAL bereitgestellten Schnittstellen und Klassen bilden das `simulator.core`-Bundle, welches von allen anderen Bundles genutzt wird und die Basis von FERAL bildet, die Kernkonzepte implementiert und daher für die Ausführung eines Simulationssystems unverzichtbar ist.

Die klar definierten öffentlichen Schnittstellen und das automatische Nachladen erlauben die Realisierung einer losen Kopplung der einzelnen Komponenten. Wir bedienen uns dieses Konzeptes durch die Realisierung der einzelnen Simulatoren als eigenständige Bundles. Ein solches Bundle für die Integration eines Simulators enthält alle notwendigen Implementierungen, bestehend aus neuen Nachrichtentypen, Nachrichtendatentypen sowie der simulatorspezifischen Implementierung der `ConcreteSimulation-`

[6] Generell gilt jedoch, dass ein Direktor die Aktivitätsperioden der untergeordneten Simulationskomponenten zwar reduzieren, aber nie vergrößern darf.

`Component`-Schnittstelle für den Simulator (vgl. Kapitel 8.1). Über die Implementierung dieser Schnittstelle kann FERAL den Simulator sowie das von diesem ausgeführte Verhaltensmodell steuern und kontrollieren. Das Bundle beinhaltet darüber hinaus auch eine ausführbare Version des spezialisierten Simulators selbst. Durch diese Art der Kapselung werden bei der Ausführung eines konkreten Simulationssystems nur die Bundles (respektive Simulatoren) geladen und ausgeführt, die tatsächlich benötigt werden. Erweiterungen um neue Simulatoren gestalten sich durch diese Form der Modularisierung sehr einfach, da es genügt, ein neues Bundle für den Simulator zu erstellen und dieses beim Start von FERAL anzugeben.

Um das automatische Laden der bei einer Simulation benötigten Simulatoren über OSGi zu realisieren, wird auch das jeweilige Simulationssystem als eigenständiges Bundle implementiert. Die Implementierung des Szenarios nutzt die öffentlichen Schnittstellen der anderen Bundles, um die benötigten Simulatoren zu instanziieren und diesen die auszuführenden Verhaltensmodelle oder Konfigurationen zuzuweisen. Die Verhaltensmodelle der einzelnen Simulationskomponenten (z. B. Simulink-Modelle) sind ihrerseits in eigene Bundle Fragmente des Simulationssystems eingebettet. Genauere Informationen zu Aufbau und Struktur des Simulationssystems finden sich in Kapitel 10.1 und 10.5.

Unsere Implementierung basiert auf dem OSGi-Framework Equinox[7], welches ebenfalls die Basis für die Entwicklungsumgebung Eclipse[8] bildet und daher hinreichend stabil und verbreitet ist, um eine Weiterentwicklung auch in Zukunft zu gewährleisten.

7.3 Allgemeiner Aufbau eines Simulationssystems für FERAL

Bisher haben wir uns ausschließlich mit dem internen Aufbau von FERAL sowie den Mechanismen, die bei der Integration von Simulatoren und der eigentlichen Simulation ineinandergreifen, beschäftigt. Diese Sichtweise entspricht der eines Entwicklers, welcher sich mit der Erweiterung von FERAL – z. B. durch die Integration neuer Simulatoren – beschäftigt. An dieser Stelle wollen wir FERAL einmal aus einer anderen Perspektive heraus betrachten: Der Sicht eines Anwenders, welcher mittels FERAL ein komplexes eingebettetes verteiltes System sowie dessen Umwelt simulieren möchte. Die Abbildung eines solchen Szenarios innerhalb von FERAL erfordert die Konstruktion eines Simulationssystems, welches FERAL und dessen integrierte Simulatoren verwendet, um das Verhalten des Gesamtsystems nachzubilden. Wie bereits erläutert, wird dazu das Simulationssystem selbst als OSGi-Bundle realisiert, sodass die Abhängigkeiten automatisch aufgelöst und die benötigten Komponenten instanziiert werden können. Wir wollen uns an dieser Stelle dem konzeptionellen Aufbau eines solchen Simulationssystems widmen, während der konkrete Aufbau des Bundles Gegenstand des Kapitels 10.5 ist.

Aus Sicht eines Anwenders genügt jedes Simulationssystem für FERAL dem Metamodell aus Abbildung 7.4. Jedes Simulationssystem ist strikt hierarchisch aufgebaut. Die Wurzel bildet ein einzelner Direktor, welcher die Ausführung aller Simulationskomponenten koordiniert. Die Simulationskomponenten selbst interagieren über Links miteinander, deren Enden jeweils die Ports[9] bilden. Simulationskomponenten sind entweder wieder selbst Direktoren oder konkrete Simulationskomponenten. Die konkreten Simulationskomponenten unterteilen sich in verhaltensbasierte Simulationskomponenten (*FSCs – Functional Simulation Components*, *CSCs – Communication-based Simulation Components*) und konvertierende Simulationskomponenten (*Bridges* und *Gateways*), wobei diese Unterteilung lediglich einer Klassifizierung auf Basis des Anwendungsbereiches auf konzeptioneller Ebene entspricht und sich nicht in der Implementierung widerspiegelt[10].

[7] http://www.eclipse.org/equinox/

[8] http://www.eclipse.org

[9] Da die Ports in unserem Metamodell fest zu den Simulationskomponenten gehören und lediglich ein Implementierungsdetail sind, haben wir auf deren Darstellung verzichtet.

[10] Für FERAL handelt es sich lediglich um Objekte, welche die *SimulationComponent*-Schnittstelle implementieren und auf dieser Basis ausgeführt werden.

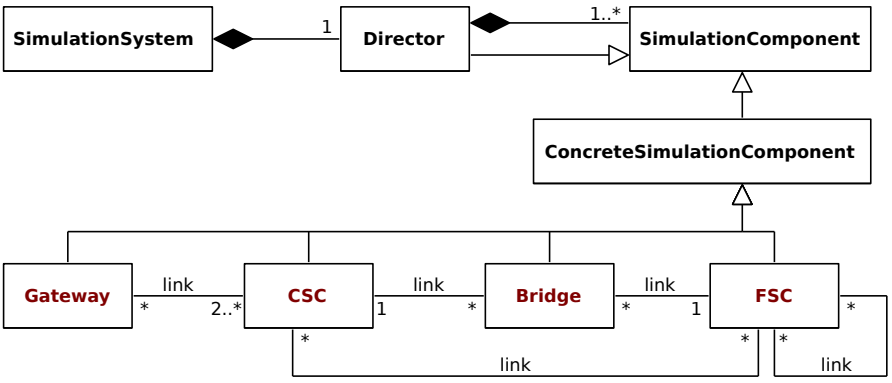


Abbildung 7.4: UML-Metamodell eines Simulationssystems für FERAL.

Der Umstand, dass Direktoren selbst Simulationskomponenten sind, gestattet die hierarchische Schachtelung. Diese Schachtelung bildet die Basis für unterschiedliche semantische Domänen (verschiedene MOCCs, Aktivitätsperioden, ...) innerhalb eines Simulationssystems (vgl. Kapite 7.1.1). Das eigentliche funktionale Verhalten der Komponenten des verteilten Systems, welches der Anwender simulieren möchte, wird durch konkrete Simulationskomponenten repräsentiert. Die Unterteilung der verhaltensbasierten Simulationskomponenten erfolgt in *funktionale Simulationskomponenten (FSC – Functional Simulation Components)*, welche das Verhalten einer Komponente innerhalb des verteilten Systems simulieren (z. B. ein Regler innerhalb eines verteilten Regelungskreises), und *Simulationskomponenten für die Simulation von Kommunikationstechnologien und Medien (CSC – Communication-based Simulation Components)*. Die verhaltensbasierten Simulationskomponenten verwenden spezialisierte Simulatoren, die ihrerseits ein bestimmtes Verhaltensmodell ausführen (z. B. führt der Matlab Simulink Simulator ein Matlab Simulink Model aus) oder über ein fest integriertes Verhaltensmodell verfügen, welches durch geeignete Konfigurationen angepasst wird (z. B. ein Simulator für einen CAN-Bus mit einer spezifischen Datenrate).

Gateways und *Bridges* sind konvertierende Simulationskomponenten, deren Aufgaben die Verbindung zwischen FSCs und CSCs (Bridges) sowie die Realisierung einer Schnittstelle zwischen CSCs (Gateways) sind. Somit dienen sie als *Glue* zwischen FSCs und CSCs sowie CSCs und CSCs. Bridges erlauben die Beschreibung des Kommunikationsverhaltens von FSCs, ohne hierbei kommunikationsspezifisches Wissen oder Eigenheiten der Kommunikationstechnologie in deren Verhaltensmodelle einzubringen. Dies ermöglicht es, Designalternativen in Bezug auf verschiedene Kommunikationstechnologien zu evaluieren, ohne immer wieder die Verhaltensmodelle der beteiligen FSCs aufwändig anpassen zu müssen. Das Gleiche leisten die Gateways für miteinander verbundene CSCs. Zusätzlich kann der Entwickler durch die Gateways auf eine generalisierte Simulationskomponente für die Weiterleitung zurückgreifen, ohne diese selbst entwerfen zu müssen. Stattdessen kann er sich darauf konzentrieren, lediglich das Weiterleitungsverhalten zu beschreiben. Die Konzepte der Bridges und Gateways bilden die Grundlage für die Evaluation der Designalternativen hinsichtlich verschiedener Kommunikationstechnologien und werden in Kapitel 10 ausführlich erläutert.

Die in rot hervorgehobenen Komponenten in Abbildung 7.4 bilden den Schwerpunkt dieser Arbeit und werden im Folgenden detailliert beschrieben. Die bereitgestellten Kernkomponenten von FERAL, die im Wesentlichen von Thomas Kuhn entwickelt wurden, sind in der Abbildung 7.4 schwarz dargestellt. Für die folgende Beschreibung der Integration von spezialisierten Simulatoren in FERAL bleiben wir, zur besseren Unterscheidung, bei der hier eingeführten konzeptionellen Unterteilung der konkreten Simulationskomponenten in FSCs, CSCs, Bridges und Gateways.

8 Functional Simulation Components

Dieses Kapitel erläutert das Vorgehen bei der Integration von Simulatoren in FERAL. Hierbei liegt der Schwerpunkt auf der Simulation von funktionalem Verhalten einzelner Komponenten oder Knoten eines verteilten Systems. Im Rahmen dieser Arbeit wurden drei unterschiedliche Simulatoren für FSCs in FERAL integriert. Diese nutzen unterschiedliche Ansätze zur Beschreibung und Modellierung des von ihnen simulierten funktionalen Verhaltens: Für die schnelle Entwicklung von ersten Prototypen, insbesondere in sehr frühen Entwicklungsphasen, können die Simulationskomponenten direkt nativ in Java implementiert werden.

Für die modellbasierte Entwicklung von FSCs unterstützen wir die beiden Spezifikationstechniken SDL und Matlab Simulink [Mat12]. Während Matlab Simulink vorwiegend bei der Entwicklung und Simulation eingebetteter, dynamischer Systeme im Kontext mathematischer oder regelungstechnischer Problemstellungen zum Einsatz kommt, liegt der Schwerpunkt bei SDL auf der Entwicklung reaktiver, verteilter Echtzeitsysteme. Da FERAL es erlaubt, mit unterschiedlichen Techniken entwickelte FSCs in einem Simulationssystem gemeinsam zu simulieren und zu evaluieren, kann für jede Komponente individuell die für die jeweilige Aufgabe/Funktionalität am besten geeignete Technik gewählt werden. Hierdurch lassen sich bereits existierende Modelle weiterverwenden[1]. Aber auch im Laufe der Entwicklung einer FSC können die unterschiedlichen Spezifikationstechniken sinnvoll kombiniert werden. So kann ein erster Entwurf beispielsweise als native Simulationskomponente mit reduziertem Verhaltensmodell erfolgen – z. B. um Interaktionen und Schnittstellen entwerfen sowie Anforderungen früh testen zu können. Dieser Entwurf kann dann in späteren Phasen bis hin zu einer fertigen Softwarekomponente verfeinert werden, z. B. unter Verwendung von Matlab Simulink.

Wir betrachten zunächst allgemein in Kapitel 8.1, welche Schritte für die Integration eines neuen Simulators in FERAL erforderlich sind. Für die drei Simulatoren für FSCs (Native Simulationskomponenten, Matlab Simulink und SDL) existieren jeweils separate Kapitel, welche die Besonderheiten und Details näher beschreiben.

Die Ergebnisse dieses Kapitels wurden in [5, 7, 10] publiziert.

8.1 Adaption von FERAL

Die Adaption von FERAL zur Integration neuer Simulatoren erfolgt in drei Schritten:

1. Bereitstellung einer Implementierung der `SimulationComponent`-Kontrollschnittstelle, sodass FERAL die Ausführung des Simulators kontrollieren kann.
2. Definition und Implementierung von Nachrichtentypen und Nachrichtendatentypen für die Interaktion mit anderen Simulationskomponenten innerhalb von FERAL.
3. Instanziierung des Simulators mit einem Verhaltensmodell im Rahmen beliebiger Simulationssysteme.

Der erste Schritt der Integration eines neuen Simulators besteht in der Implementierung einer Java-Kontrollkomponente für den zu integrierenden Simulator. Die spezifische Java-Kontrollkomponente stellt die Schnittstelle zwischen FERAL und dem Simulator dar und implementiert die `SimulationComponent`-Schnittstelle. Über die in der Schnittstelle definierten Methoden kontrolliert und steuert

[1] Dies betrifft vor allem Matlab Simulink-Modelle, da Simulink in der Industrie eine sehr hohe Verbreitung genießt.

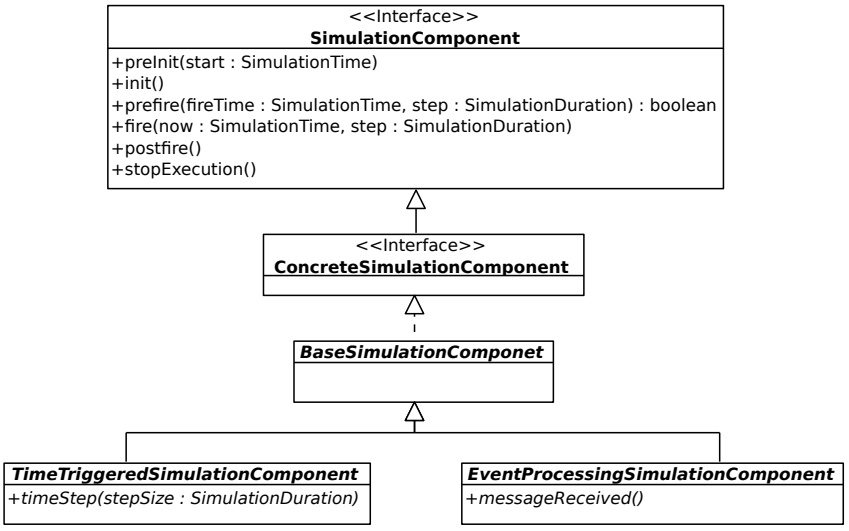


Abbildung 8.1: Struktur und Implementierung der `SimulationComponent`-Schnittstelle.

FERAL (bzw. der für die Simulationskomponente verantwortliche Direktor) die Ausführung des Simulators und somit indirekt auch die Ausführung seines Verhaltensmodells.

Das UML-Diagramm 8.1 gibt einen Überblick über die `SimulationComponent`-Schnittstelle, deren Methoden sowie deren Struktur und Umsetzung (vgl. Kapitel 7.1.2). Die `init`-Methoden der `SimulationComponent`-Schnittstelle dienen der Initialisierung des angesteuerten Simulators, während die Methode `fire` durch den Direktor aufgerufen wird, um den jeweiligen Simulator auszuführen (z. B. für einen Zeitschritt bei einem zeitgetriggerten MOCC oder die Verarbeitung eines Ereignisses bei einem ereignisgetriggerten MOCC). Die Methoden `prefire` und `postfire` dienen, dazu die Ausführung vorzubereiten bzw. abzuschließen. Innerhalb der `prefire`-Methode werden empfangene Nachrichten der InputPorts der Java-Kontrollkomponente (siehe Schritt 2) an den Simulator transferiert und ermittelt, ob dieser ausgeführt werden muss. Die `postfire`-Methode überträgt – nach dem Aufruf von `fire` – Nachrichten oder Ereignisse, welche durch den Simulator oder die Ausführung seines Verhaltensmodells generiert wurden, an die OutputPorts der Java-Kontrollkomponente. Zusätzlich verfügt die `SimulationComponent`-Schnittstelle noch über Methoden zum Anlegen und Verwalten von Input- und OutputPorts, auf deren Darstellung aus Gründen der Übersichtlichkeit verzichtet wurde.

Um die Realisierung einer Java-Kontrollkomponente zu vereinfachen, liefert FERAL eine Implementierung der `SimulationComponent`-Schnittstelle in Form der abstrakten Klasse `BaseSimulationComponent`, welche sich auf allgemeine und von dem MOCC unabhängige Funktionalitäten beschränkt. Die beiden abstrakten Klassen[2] `TimeTriggeredSimulationComponent` und `EventProcessingSimulationComponent` erweitern diese um spezifisches Verhalten für das zeit- bzw. ereignisgetriggerte MOCC. Um eine fertige Java-Kontrollkomponente zu erhalten, genügt es häufig, die abstrakten Klassen, um Glue-Code zur Ansteuerung des konkreten Simulators – innerhalb der Methode `timeStep` bzw. `messageReceived` – zu ergänzen. Die in Form der abstrakten Klassen bereitgestellte Implementierung trägt bereits dafür Sorge, dass die korrekten Methoden für jeden Zeitschritt bzw. bei der Ankunft eines Ereignisses aufgerufen werden.

Der Aufwand für die Integration erhöht sich, wenn der jeweilige Simulator selbst nicht, wie FERAL, in Java, sondern in einer anderen Sprache realisiert wurde. In diesem Fall wird zusätzlicher Glue-Code benötigt, um die Interaktion zwischen der Java-Kontrollkomponente und dem Simulator zu realisieren (z. B.

[2]FERAL liefert noch weitere abstrakte Klassen für andere MOCCs mit, wir beschränken uns hier jedoch auf die für unsere Fälle relevanten Klassen.

unter Einsatz von JNI – Java Native Interface [Ora] – bei einem in C realisierten Simulator, vgl. Kapitel 8.3). Hierzu gehört auch die ggf. notwendige Konvertierung von Nachrichten- und Nachrichtendatentypen von FERAL in die Datenstrukturen der jeweiligen Zielsprache und umgekehrt.

Der zweite Schritt bei der Integration eines neuen Simulators besteht in der Definition einer geeigneten Nachrichtenschnittstelle, über die der neu integrierte Simulator mit anderen Simulationskomponenten interagieren und Daten austauschen kann. Hierzu müssen zunächst allgemein[3] die für die Interaktion benötigten Input- und OutputPorts für den Simulator (ggf. in Abhängigkeit von den Verhaltensmodellen) identifiziert werden. Diese Ports werden von der Instanz der Java-Kontrollkomponente verwaltet und angeboten, welche den konkreten Simulator (inkl. dessen Verhaltensmodell) in FERAL repräsentiert. Für den Austausch von Daten mit anderen Simulationskomponenten können entweder die bereits von FERAL bereitgestellten Nachrichten- und Nachrichtendatentypen verwendet oder eigene Implementierungen der jeweiligen Schnittstellen (`Message` bzw. `MessageData`) erstellt werden, welche speziell auf die Anforderungen des Simulators zugeschnitten sind.

Beispiele für eigene Nachrichtentypen sind die beiden Klassen `PTP_RX` und `PTP_TX` (Abbildung 8.2), diese erweitern das bestehende Nachrichtentypsystem von FERAL durch die Implementierung der `Message`-Schnittstelle (vgl. Abbildung 7.2, Kapitel 7.1.1). Diese Klassen repräsentieren Sende- und Empfangsereignisse eines spezifischen Simulators für verzögerungsbehaftete Punkt-zu-Punkt-Verbindung. Ebenfalls in Abbildung 8.2 dargestellt ist der spezielle Nachrichtendatentyp `CANMessage` für die Repräsentation von CAN-Nachrichten, welcher Bestandteil der Integration des CAN-Simulators ist (vgl. Kapitel 9.7.1). Eigene Nachrichtendatentypen müssen die Schnittstelle `MessageData` implementieren, welche als Basistyp aller Nachrichtendatentypen in FERAL dient (vgl. Kapitel 7.1).

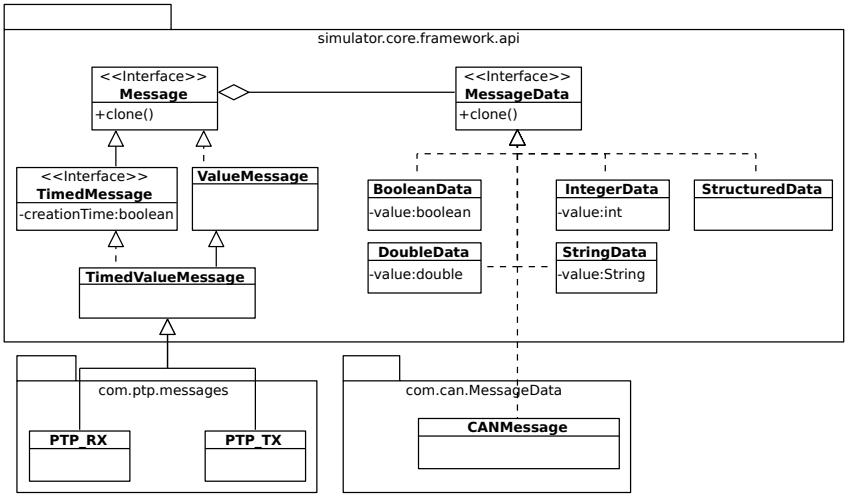


Abbildung 8.2: UML-Diagramm des Nachrichtentypsystems von FERAL.

Der dritte und letzte Schritt der Adaption besteht in der Instanziierung der Simulationskomponenten und deren Simulatoren zur Abbildung eines konkreten Szenarios. Ausgangspunkt hierfür ist ein *abstraktes Simulationssystem*, welches lediglich Struktur und beteiligte Komponenten (sowie deren Rollen und Interaktionen) des Simulationsszenarios beschreibt. Dann werden für die Komponenten Simulatoren ausgewählt, die dazu geeignet sind, das geforderte Verhalten zu beschreiben. Innerhalb von FERAL wird jede Komponente bzw. der für die Komponenten gewählte Simulator durch eine eigene Instanz der jeweiligen Java-Kontrollkomponenten repräsentiert, über die FERAL die Ausführung steuert. Das Verhalten der einzelnen Komponenten liegt in Form von Verhaltensmodellen (z. B. als Matlab Simulink-

[3] Allgemein bedeutet hierbei, dass es sich sowohl um eine konkrete Liste von Ports handeln kann als auch um Vorschriften, wie automatisiert die Datenschnittstellen eines auszuführenden Verhaltensmodells auf Ports in FERAL abzubilden sind.

Modell oder SDL-Spezifikation) vor, welches durch den jeweiligen Simulator interpretiert wird. Bei CSCs, welche Kommunikationssysteme simulieren, ist das Verhaltensmodell bereits durch die Kommunikationstechnologie festgelegt. Dafür benötigt eine Instanz jedoch konkrete Konfigurationsparameter, wie z. B. Slotreservierungen, Ablaufpläne oder die Übertragungsrate. Im Kontext von CSCs übernehmen diese Konfigurationen die Rolle der Modelle.

Das Ergebnis dieser Festlegungen (Simulatoren und Modelle) ergibt das *konkrete Simulationssystem*, welches mit FERAL simuliert werden kann. Eine detaillierte Beschreibung zur Erstellung eines Simulationssystems durch die Definition abstrakter und konkreter Simulationssysteme liefert das Kapitel 10.1. Während dieser letzte Schritt fester Bestandteil bei der Erstellung jedes neuen Simulationsszenarios ist, sind die Schritte 1 und 2 nur einmalig für die Anbindung eines neuen Simulators notwendig.

8.2 Native Simulationskomponenten

Ein besonders schneller Weg zu einer einfachen FSC[4] führt über die native Implementierung. Hierbei wird die gewünschte Funktionalität bzw. das jeweilige Verhalten der FSC direkt in Java realisiert. Die Interaktion mit FERAL erfolgt mittels der hierfür bereitgestellten Schnittstellen. Die Implementierung der `SimulationComponent`-Schnittstelle ermöglicht die Realisierung von Simulationskomponenten für beliebige MOCCs und erlaubt deren Ausführung durch FERALs Direktoren. Um den Aufwand bei der Umsetzung einer nativen Simulationskomponente zu reduzieren, bietet es sich an, eine der abstrakten Klassen (`BaseSimulationComponent`, `TimeTriggeredSimulationComponent` oder `EventProcessingSimulationComponent`) als Ausgangspunkt für die Implementierung zu verwenden. Diese kann dann entsprechend verfeinert und um das gewünschte Verhalten erweitert werden (durch Implementierungen innerhalb der `fire`, `timeStep` bzw. `messageReceived`-Methode).

Zusätzlich (Schritt 2) muss die nach außen hin sichtbare Schnittstelle der Simulationskomponente definiert werden. Dies erfolgt durch die Festlegung der Input- und OutputPorts. Auch bei der nativen Implementierung steht es dem Entwickler wieder frei, eigene Nachrichten- und Nachrichtendatentypen zu implementieren. Im Idealfall genügen jedoch die von FERAL bereitgestellten Nachrichten- und Nachrichtendatentypen für die Modellierung der Schnittstelle sowie die Interaktion zwischen den Simulationskomponenten. Auch bei nativen Simulationskomponenten kann für die Anbindung mit CSCs auf die Konzepte der *Bridges* zurückgegriffen werden. Diese ermöglichen es, das Kommunikationsverhalten deklarativ zu beschreiben, ohne für die jeweilige Kommunikationstechnologie spezifischen Code entwickeln und warten zu müssen (vgl. Kapitel 10). Auf diese Weise ist auch ein Wechsel der Kommunikationstechnologie im Laufe des Entwicklungsprozesses oder bei der Evaluation von Designalternativen mit geringen Aufwänden verbunden.

Der Einsatz nativer Simulationskomponenten bietet sich vor allem für einfache Funktionalitäten innerhalb des Simulationsszenarios an, die nicht im Fokus der Analyse oder Entwicklung stehen, sowie für den frühen Entwurf von Komponenten (Prototypen), die dann im Laufe des Entwicklungsprozesses sukzessive verfeinert werden. Ein Beispiel hierfür wäre eine Simulationskomponente, die einen einfachen Sensor repräsentiert, der innerhalb der Simulation lediglich eine vorgegebene Abfolge von Werten ausgibt, die z. B. im Rahmen einer Testfahrt aufgezeichnet wurden.

Die Nutzung funktionaler Prototypen in frühen Entwicklungsphasen erlaubt von Anfang an die Konstruktion eines vollständigen und ausführbaren Simulationssystems. Mit den Prototypen können schon zu Beginn der Entwicklung die benötigten Schnittstellen und Interaktionsmuster entworfen und im Rahmen von Simulationen validiert werden, um Fehler, Missverständnisse und Inkompatibilitäten frühzeitig zu

[4] Da sich mit Hilfe nativer Simulationskomponenten beliebiges Verhalten umsetzen lässt, können diese auch verwendet werden, um eine CSC zu implementieren. Da CSCs jedoch sehr schnell in ihrer Realisierung, dem Verhalten sowie den Schnittstellen (Input- und OutputPorts, Nachrichtentypen, etc.) sehr komplex werden, stellt die Entwicklung dedizierter Simulatoren für FERAL oder die Integration existierender Simulatoren den (aus unserer Sicht) flexibleren Weg dar eine CSC umzusetzen (vgl. Kapitel 9).

entdecken. Hierbei beschränkt sich die Realisierung der Prototypen häufig auf ein vereinfachtes Modell des nach außen hin sichtbaren Verhaltens. In den weiteren Entwicklungsphasen können die Prototypen dann – sofern erforderlich – verfeinert und sukzessive durch deren finale Versionen ersetzt werden. Dadurch, dass FERAL in einem Simulationssystem den Einsatz von Simulationskomponenten mit unterschiedlichem Abstraktionsgrad unterstützt, bleibt auch bei der schrittweisen Verfeinerung das Simulationssystem jederzeit ausführbar. Auf diese Weise werden kontinuierlich Integrationstests über alle Entwicklungsphasen hinweg ermöglicht.

In Bezug auf das Adaptionsschema aus Kapitel 8.1 fällt auf, dass bei nativen Simulationskomponenten die beschriebene Unterscheidung zwischen Simulator und dem durch diesen Simulator ausgeführten Verhaltensmodell nicht vollständig gegeben ist. Zwar bietet FERAL mit den abstrakten Klassen `TimeTriggeredSimulationComponent` bzw. `EventProcessingSimulationComponent` einen Rahmen für die Implementierung der nativen Simulationskomponenten, diesen als Simulator zu bezeichnen wäre jedoch übertrieben, zumal eine klare Trennung zu dem Code, welcher das Verhalten realisiert, fehlt. Dies ist jedoch nicht problematisch, da die typischen Verhaltensmodelle nativer Simulationskomponenten aufgrund des Prototypen-Charakters in der Regel sehr einfach gestaltet und genau auf ein Szenario zugeschnitten sind. Das heißt, eine Wiederverwendung ist unwahrscheinlich, sodass bei nativen Simulationskomponenten ohnehin stets alle drei Adaptionsschritte anfallen. Aber auch diesbezüglich zeigen die Erfahrungen aus der Praxis, dass die Schnittstellen und abstrakten Klassen von FERAL eine einfache und schnelle Implementierung nativer Simulationskomponenten ermöglichen. Dies hängt natürlich auch damit zusammen, dass, im Gegensatz zu der Integration eines Simulators, lediglich das Verhalten direkt in Java zu implementieren ist und kein aufwändiger generischer Glue-Code für die Ansteuerung (Synchronisation) eines (externen) Simulators und dessen Verhaltensmodell benötigt wird.

8.3 Integration von Matlab Simulink

Mathworks Matlab Simulink [Mat12] ist ein in Industrie und Forschung weitverbreitetes Softwarepaket für die modellbasierte Entwicklung und Simulation dynamischer sowie eingebetteter Systeme [NL02]. Besonders in der Automobilindustrie, aber auch in vielen anderen Bereichen, ist Simulink der de facto Standard in Bezug auf modellbasierte Entwicklung sowie den Entwurf mathematischer und physikalischer Modelle und Regelungssysteme.

8.3.1 Überblick über Matlab Simulink

Simulink unterstützt, neben der Modellierung zeitdiskreter und kontinuierlicher Regelungsalgorithmen sowie dynamischer Systeme, auch die Entwicklung und Beschreibung zustandsbasierter Systeme mittels Stateflow. Die Modellierung erfolgt graphisch mittels hierarchischer Blockdiagramme, welche das Verhalten des Systems spezifizieren. Die einzelnen Blöcke können ihrerseits wieder durch ein Blockdiagramm verfeinert werden oder stammen aus einer Blockbibliothek und weisen ein vordefiniertes Verhalten auf. Alternativ kann die Übertragungsfunktion eines Blockes auch unter Verwendung von C-Code oder mittels der in Matlab integrierten proprietären Sprache spezifiziert werden. Die Schnittstellen der Blöcke bestehen aus einer Menge von unidirektionalen, benannten ein- und ausgehenden Ports; der Datenfluss zwischen den Blöcken wird durch Verbindungslinien zwischen deren Ports beschrieben. Neben dem graphischen Editor enthält Simulink auch einen Simulator und Debugger, um die Fehlersuche zu erleichtern. Der Einsatz von Matlab Simulink ist nicht an eine spezielle Entwicklungsphase oder ein spezifisches Abstraktionsniveau gebunden. So kann Simulink für die Entwicklung von (frühen) Prototypen bis hin zur fertigen Funktionalität – inklusive automatischer Codegenerierung – genutzt werden.

Eine Vielzahl von existierenden Toolboxes[5] erweitert die Funktionalitäten von Simulink. Diese Toolboxes stellen Blöcke für spezielle Anwendungsdomänen, wie z. B. die Regelungstechnik oder Luft- und Raumfahrt, zur Verfügung oder erweitern auf andere Weise die Funktionalitäten von Simulink. So ermöglicht der Mathworks Embedded Coder die automatische Generierung von C-Code aus Simulink-Modellen für unterschiedliche Zielplattformen. Andere Toolboxes konzentrieren sich auf die Bereitstellung komplexer physikalischer Modelle der Umgebung (z. B. CarMaker, CarSim, DRIVE und TruckSim). Diese unterstützen die Simulation physikalischer Eigenschaften von PKWs oder LKWs, sodass neu entworfene Funktionalitäten direkt in einer möglichst realistisch simulierten Umgebung getestet werden können. Die Anbindung zwischen Toolbox und Simulink-Modell, welches die Funktionalität beschreibt, erfolgt ebenfalls auf Blockebene, d.h., die Toolbox stellt spezielle Blöcke mit Ports bereit, über die auf das Modell der Umgebung zugegriffen und diese beeinflusst werden kann.

Die Integration von Matlab Simulink in FERAL erlaubt somit nicht nur die Simulation eigener bzw. existierender Simulink-Modelle, sondern gestattet (ohne weiteren Integrationsaufwand) auch den Zugriff auf die Funktionalitäten solcher Toolboxes und deren Blockbibliotheken. Dies erleichtert, je nach Verfügbarkeit und Anwendungsgebiet, die schnelle Konstruktion realistischer Simulationsumgebungen für die Entwicklung komplexer eingebetteter (verteilter) Systeme.

8.3.2 Anwendungsszenarien für Matlab Simulink und FERAL

Wir bieten in Bezug auf Matlab Simulink zwei unterschiedliche Arten der Integration in FERAL an, die unterschiedliche Anwendungsszenarien adressieren: Liegt der Fokus auf der Entwicklung sowie dem Debugging einer einzelnen – auf einem Simulink-Modell beruhenden – Funktionalität, so kann FERAL verwendet werden, um eine realistische Umgebung zu simulieren. Hierzu gehört sowohl die Simulation des Verhaltens anderer funktionaler Komponenten des Gesamtsystems, aber auch die Simulation von Kommunikation und den damit verbundenen Verzögerungen sowie deren Auswirkungen und ggf. der physikalischen Umwelt, mit entsprechenden Stimuli und Interaktionsmöglichkeiten. In diesem Fall wird FERAL – und mit ihm alle von FERAL simulierten Komponenten des Simulationsszenarios – unter der Kontrolle des in Simulink integrierten Simulators bzw. Debuggers ausgeführt. Bei diesem Ansatz stehen dem Entwickler sämtliche Werkzeuge von Simulink zur Verfügung, um sein Modell zu optimieren und Fehler aufzuspüren, wohingegen FERAL sich auf die Simulation der Systemumgebung beschränkt. Um die Synchronität zwischen dem Simulink-Modell und der Umgebungssimulation zu gewährleisten, kontrolliert Simulink das Voranschreiten der Simulationszeit für das gesamte Simulationssystem, inklusive aller von FERAL simulierten Komponenten. Hierdurch wird auch eine schrittweise Ausführung des Simulink-Modells samt dessen Umgebungssimulation innerhalb des Simulink Debuggers ermöglicht.

Um FERAL auf diese Weise in Simulink integrieren zu können, entwickelte Thomas Forster im Rahmen der Kooperation mit dem Fraunhofer IESE einen speziellen Simulink-Direktor entwickelt. Dieser muss für dieses Anwendungsszenario als oberster Direktor für das Simulationssystem verwendet werden. Der Simulink-Direktor wird über TCP/IP von Simulink gesteuert und mit ihm auch alle von FERAL ausgeführten Simulationskomponenten.

Das zweite Anwendungsszenario sieht die Rolle von Simulink in der Entwicklung einzelner Komponenten eines Simulationssystems und erlaubt daher die Integration von Simulink-Modellen in Form eigenständiger konkreter Simulationskomponenten (FSCs). Hierbei wird das Verhalten der Simulationskomponenten mit Hilfe eines Simulink-Modells spezifiziert, welches dann unter der Kontrolle von FERAL in einer speziellen Laufzeitumgebung ausgeführt werden. Die Rolle von Simulink als Werkzeug beschränkt sich hierbei im Wesentlichen auf den eines graphischen Editors.

Unsere Realisierung nutzt Mathworks Embedded-Coder, um die Simulink-Modelle zunächst in C-Code zu überführen. Der generierte Code wird dann unter Verwendung des GCC zusammen mit der

[5] In Simulink werden Erweiterungen als Toolboxes bezeichnet. Eine kleine und unvollständige Übersicht der verfügbaren Erweiterungen ist unter http://www.mathtools-net/MATLAB/ zu finden.

Laufzeitumgebung in eine Bibliothek übersetzt, die dann in FERAL geladen und ausgeführt werden kann. Für die Interaktion zwischen Java (der Java-Kontrollkomponente) und dem C-Code wird JNI verwendet. Dieses Vorgehen hat gleich mehrere Vorteile:

1. Für die Simulation wird (nach der Codegenerierung) weder Simulink noch eine entsprechende Lizenz benötigt.
2. Durch die Verwendung des Embedded-Coders wird im Simulator exakt der gleiche Code ausgeführt, welcher auch auf der späteren Zielplattform zum Einsatz kommt.
3. Die Bibliothek, welche das Verhalten der jeweiligen Komponente kapselt, kann an Industriepartner herausgegeben werden, sodass diese ihrerseits Simulationen und Tests durchführen können, ohne Einblick in das eigentliche Simulink-Modell gewähren zu müssen.

8.3.3 Integration von Matlab Simulink in FERAL

An dieser Stelle wollen wir uns auf das zweite Anwendungsszenario in Bezug auf die Integration von Matlab Simulink in FERAL und dessen Umsetzung beschränken, die im Rahmen dieser Arbeit entstanden ist. Im Folgenden beschreiben wir das Vorgehen anhand der drei vorgestellten Adaptionsschritte aus Kapitel 8.1. Bei der hier vorgestellten Realisierung bedienen wir uns keines Matlab Simulink Simulators im klassischen Sinne, der ein Verhaltensmodell *interpretiert*; stattdessen haben wir eine spezielle Laufzeitumgebung entwickelt, welche zusammen mit dem aus dem Simulink-Modell generierten C-Code, direkt von FERAL geladen und über eine spezielle Kontrollkomponente ausgeführt werden kann. Nach der Übersetzung bzw. zur Laufzeit existiert somit keine klassische Trennung zwischen Simulator und Verhaltensmodell.

Der erste Schritt besteht in der Entwicklung einer generischen Java-Kontrollkomponente, welche es FERAL gestattet, die Ausführung eines beliebigen Simulink-Modells zu steuern. Da die Ausführungssemantik von Simulink exakt der Semantik zeitgetriggerter Simulationskomponenten in FERAL entspricht, implementieren wir die Java-Kontrollkomponente als Spezialisierung der `TimeTriggered-SimulationComponent`-Klasse. Die Kontrollkomponente für Simulink bezeichnen wir im Folgenden als *FERAL-Simulink-CC*. Jede Instanz der FERAL-Simulink-CC kontrolliert die Ausführung eines konkreten Simulink-Modells, welches bei der Instanziierung als Parameter übergeben wird. Die Schnittstelle zwischen FERAL-Simulink-CC und dem eigentlichen Simulink-Modell wird durch die Klasse `SimulinkInterface` definiert. Diese definiert Methoden zur Ausführung des Simulink-Modells (für einen Zeitschritt fester Länge) sowie zum Auslesen und Setzen von Werten der ein- und ausgehenden Ports des Simulink-Modells[6].

Abbildung 8.3 zeigt die FERAL-Simulink-CC, die `SimulinkInterface`-Schnittstelle und deren Implementierung für ein konkretes Simulink-Modell mit der Bezeichnung A. Zu beachten ist, dass jedes Simulink-Modell eine eigene Implementierung der `SimulinkInterface`-Schnittstelle benötigt. Die Implementierung ist minimalistisch und beschränkt sich auf das Anlegen einer Klasse mit eindeutigem Namen sowie einem Konstruktor, welcher die C-Bibliothek, die das Simulink-Modell enthält, lädt. Zusätzlich werden alle in der `SimulinkInterface`-Schnittstelle definierten Methoden als `native` deklariert. Hierdurch werden alle Aufrufe dieser Methoden per JNI an die gleichnamigen Funktionen[7] der zuvor geladenen C-Bibliothek weitergereicht. Daher bezeichnen wir eine solche Implementierung des `SimulinkInterface` als *Simulator Glue-Code*.

Der von Matlab mit Hilfe des Embedded-Coder generierte C-Code implementiert ausschließlich das Verhalten des spezifizierten Modells und benötigt noch eine zusätzliche Laufzeitumgebung, um ausführbar zu sein. Unsere Laufzeitumgebung für FERAL bezeichnen wir als Matlab-Runtime. Diese wird zusammen mit dem generierten Code übersetzt und zu einer Bibliothek gelinkt. Bestandteil der Matlab-Runtime sind

[6] Bei diesen Ports handelt es sich um die in Matlab angelegten Ports des Simulink-Modells und nicht um die Input- und OutputPorts von FERAL.

[7] Gleichnamig im Sinne der JNI-Namenskonvention, siehe [Ora].

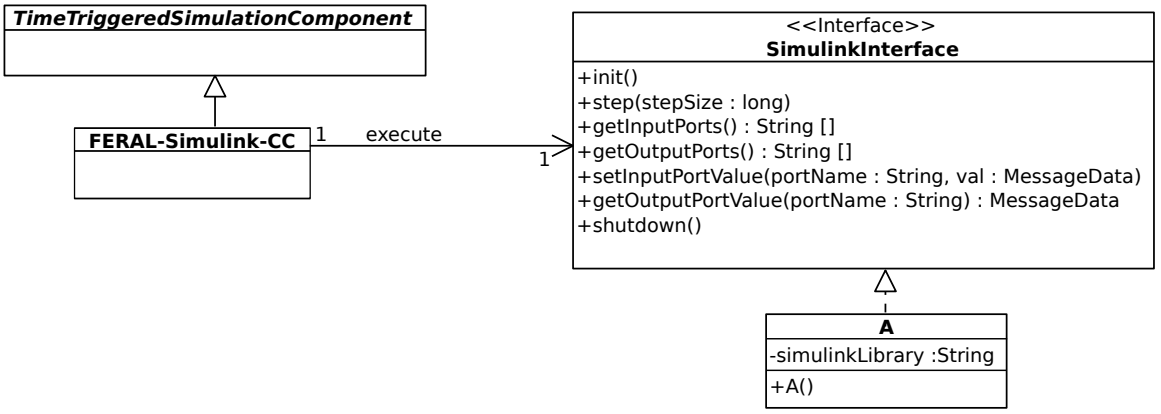


Abbildung 8.3: UML Klassendiagramm der FERAL-Simulink-CC.

die Implementierungen für die innerhalb der `SimulinkInterface`-Schnittstelle definierten Methoden, mit denen die Java-Kontrollkomponente die Ausführung des Simulink-Modells steuert, sowie Funktionen zur Konvertierung zwischen den FERAL-Datentypen und den Simulink-Datentypen.

Da JNI für die aufzurufenden Funktionen innerhalb des C-Codes ein spezielles Namensschema erzwingt, welches den Namen der `native`-deklarierten Java-Methode sowie den Klassennamen enthält, ist ein direkter Aufruf der Methoden der Matlab-Runtime nicht möglich, ohne diese für jedes Simulink-Modell anzupassen. Daher generieren wir auf der C-Seite zusätzlichen Glue-Code – den Matlab-Runtime Glue-Code –, welcher die Aufrufe aus dem Simulator Glue-Code an die Matlab-Runtime delegiert. Abbildung 8.4 zeigt dies anhand eines konkreten Beispiels. Der Aufruf der Methode `step` des Simulink-Modells A durch die Java-Kontrollkomponente (per JNI) wird an die Prozedur `Java_scenario_system_A_step` weitergeleitet. Diese wird von dem generierten Matlab-Runtime Glue-Code bereitgestellt und delegiert den Aufruf an die entsprechende Prozedur der Matlab Runtime, welche die eigentliche Funktionalität implementiert.

Der zweite Schritt der Integration besteht in der Definition einer geeigneten Schnittstelle zur Interaktion und für den Austausch von Nachrichten zwischen der Simulink FSC (bzw. dem ausgeführten Simulink-Modell) und anderen Simulationskomponenten. In FERAL besteht eine solche Schnittstelle aus einer Menge von Input- und OutputPorts sowie den verwendeten Nachrichten- und Nachrichtendatentypen für den Austausch von Werten. Hinsichtlich der Input- und OutputPorts nutzen wir die semantischen Ähnlichkeiten zwischen Simulink und FERAL: Da Simulink-Modelle ebenfalls unidirektionale Ports mit eindeutigen Bezeichnern zur Definition der Schnittstelle verwenden, übernehmen wir diese einfach für die FERAL-Simulink-CC. Das heißt, für jeden InputPort des Simulink-Modells verfügt die jeweilige

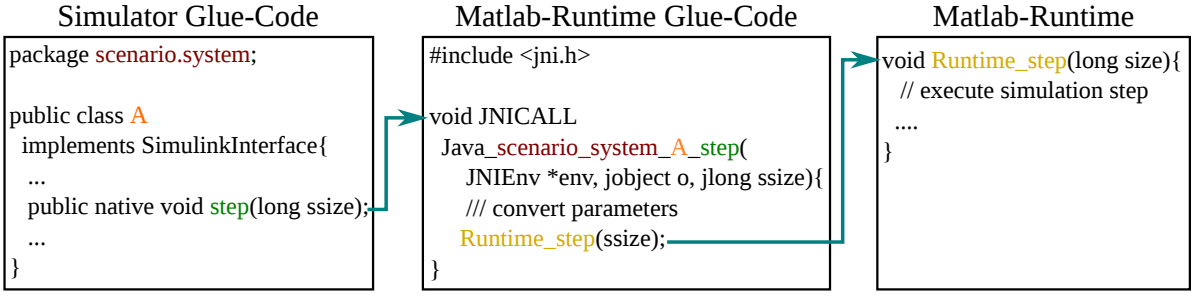


Abbildung 8.4: Glue-Code für die Anbindung der FERAL-Simulink-CC an die Matlab-Runtime.

Instanz der FERAL-Simulink-CC über einen gleichnamigen InputPort; ebenso verhält es sich mit den OutputPorts. Ein- bzw. ausgehende Nachrichten werden dann in beide Richtungen an die gleichnamigen Ports weitergereicht. Die unveränderte Schnittstelle zwischen Simulink-Modellen und deren FSCs erlaubt eine transparente Nutzung der Simulink-Modelle in FERAL.

Die Ports eines Simulink-Modells werden im erzeugten C-Code auf Variablen und einfache C-Datentypen abgebildet. Daher genügen die von FERAL bereitgestellten Nachrichtendatentypen für deren Repräsentation (vgl. Abbildung 8.2) und die Einführung eigener Nachrichtendatentypen ist nicht notwendig. Die Konvertierung zwischen den Nachrichtendatentypen von FERAL und den C-Datentypen des Simulink-Modells erfolgt durch Hilfsfunktionen der Matlab-Runtime. Als Nachrichtendatentyp für die Übergabe von Werten zwischen Simulationskomponenten werden Objekte der Klasse `ValueMessage` verwendet.

Um die Ports eines Simulink-Modells beim Laden der Bibliothek automatisch ermitteln zu können, wird ein Port-Mapping benötigt, da C – anders als Java – nicht über Mechanismen für Reflection verfügt. Das Mapping wird ebenfalls generiert und ist Bestandteil des Matlab-Runtime Glue-Code. Es beinhaltet neben den Namen der Input- und OutputPorts auch eine Abbildung zwischen Portnamen sowie den Speicheradressen der dazugehörenden Variablen innerhalb des C-Codes des Simulink-Modells. Zusätzliche Typinformationen erlauben die automatische Konvertierung zwischen den Typen von FERAL und den C-Datentypen. Anhand dieser Informationen kann die Matlab-Runtime bei einem Aufruf der Methode `setInputPortValue` bzw. `getOutputPortValue` die für einen Port verantwortliche Variable ermitteln, um diese entweder zu lesen oder zu schreiben. Das Port-Mapping wird beim Erzeugen einer Simulink FSC abgefragt, um bei der Instanziierung der FERAL-Simulink-CC die entsprechenden Pendants der Ports des Simulink-Modells anlegen zu können. Listing 8.1 zeigt einen Ausschnitt eines solchen Port-Mapping; der erste Eintrag der Struktur `PORTSIGNAL_T` enthält die Größe des C-Datentyps in Bits, der zweite verweist auf die Speicheradresse der Variablen (Zeilen 7-9). Die Einträge drei und vier identifizieren den Typ der Variablen in Matlab sowie deren nativen C-Datentyp.

Listing 8.1: Beispiel für ein Port-Mapping.

```
1  /** InputPorts */
   extern ExternalInputs_Car Car_U;
3
   /** Identified inPorts */
5  uint8_t inPortsLen = 3;
   PORTSIGNAL_T inPorts[] = {
7    {16, (void*) &(Car_U.Stellgroesse), MLT_int16_T, CT_int16_T, "Stellgroesse"},
     {64, (void*) &(Car_U.Stoergroesse), MLT_real_T, CT_double, "Stoergroesse"},
9    {64, (void*) &(Car_U.Notbremsung), MLT_real_T, CT_double, "Notbremsung"},
   };
11
   /** OutputPorts */
13 ...
```

Der dritte und letzte Schritt besteht in der Instanziierung des Simulators unter Angabe des auszuführenden Verhaltensmodells. In diesem konkreten Fall genügt die einfache Auswahl eines Simulink-Modells und die Instanziierung der FERAL-Simulink-CC allein nicht, da wie bereits beschrieben, an verschiedenen Stellen speziell an das Simulink-Modell angepasster Code benötigt wird. Dies betrifft den Simulator Glue-Code für den FERAL-Simulink-CC, welcher das `SimulinkInterface` implementiert, den Matlab-Runtime Glue-Code für die Delegation der Methodenaufrufe des Simulator Glue-Codes an die Matlab-Runtime sowie das Port-Mapping. Um eine schnelle und fehlerfreie Integration von Simulink-Modellen zu gewährleisten, müssen diese Artefakte automatisiert erzeugt werden.

Die Abbildung 8.5 zeigt die benötigten Artefakte sowie die Werkzeuge und Prozesse zu deren Erzeugung. Ausgangspunkt ist das Simulink-Modell, welches zunächst, unter Verwendung des Embedded-Coders in

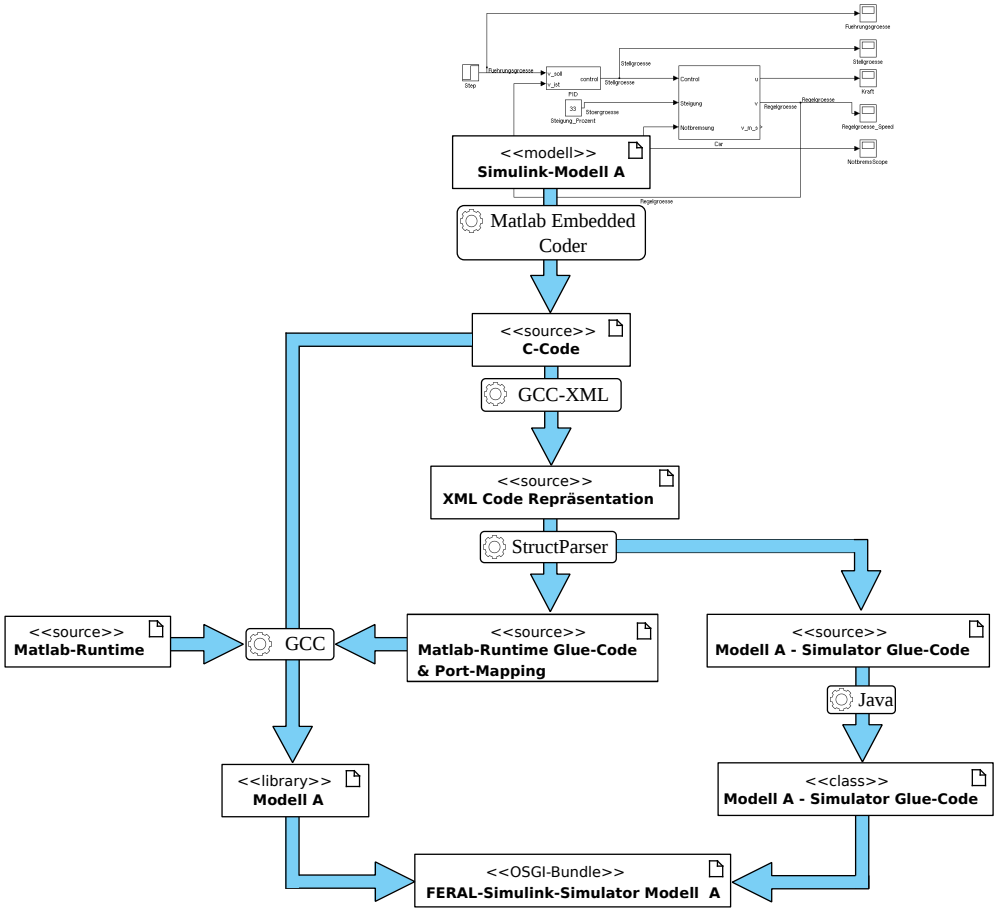


Abbildung 8.5: Ablauf bei der Erstellung einer Simulationskomponente aus einem Simulink-Modell.

C-Code umgewandelt wird. Der generierte C-Code wird dann automatisch analysiert[8]. Hierfür wird dieser zunächst mit *GCCXML* in eine XML-Repräsentation transformiert, welche dann durch unseren *StructParser* weiterverarbeitet wird. Der *StructParser* extrahiert das Port-Mapping (vgl. Listing 8.1) und erzeugt den benötigten Matlab-Runtime Glue-Code ebenso wie den Simulator Glue-Code (vgl. Abbildung 8.4), welcher das `SimulinkInterface` implementiert.

Der aus dem Simulink-Modell generierte C-Code wird zusammen mit der Matlab-Runtime, dem generierten Matlab-Runtime Glue-Code sowie dem Port-Mapping zu einer Bibliothek übersetzt[9]. Diese bildet zusammen mit dem Simulator Glue-Code des Simulink-Modells ein in sich abgeschlossenes OSGI-Bundle, welches sowohl einfach in Java geladen als auch archiviert werden kann. Dieses Bundle beinhaltet sowohl die Realisierung des in Simulink beschriebenen Verhaltensmodells als auch die für die Ausführung durch FERAL notwendige Laufzeitumgebung. Um das Simulink-Modell in einer Simulation zu verwenden, genügt es, in der Szenariobeschreibung eine Instanz der FERAL-Simulink-CC zu erzeugen und dieser den Namen der Klasse des generierten Simulator Glue-Code zu übergeben. Anhand des Klassennamens wird das verantwortliche OSGi-Bundle identifiziert und automatisch geladen.

Die Abbildung 8.6 zeigt anhand eines Beispiels die einzelnen Schichten und deren logisches Zusam-

[8] Wir haben uns dazu entschieden, direkt den generierten C-Code anstelle des Simulink-Modells zu analysieren, da so auch die gewählten Optionen bei der Codegenerierung (z. B. Vorgaben zur Abbildung auf C-Datentypen) berücksichtigt werden können.

[9] Der gesamte Ablauf aus Abbildung 8.5 wird durch ein spezielles Makefile unterstützt, welches nahezu alle Schritte automatisiert.

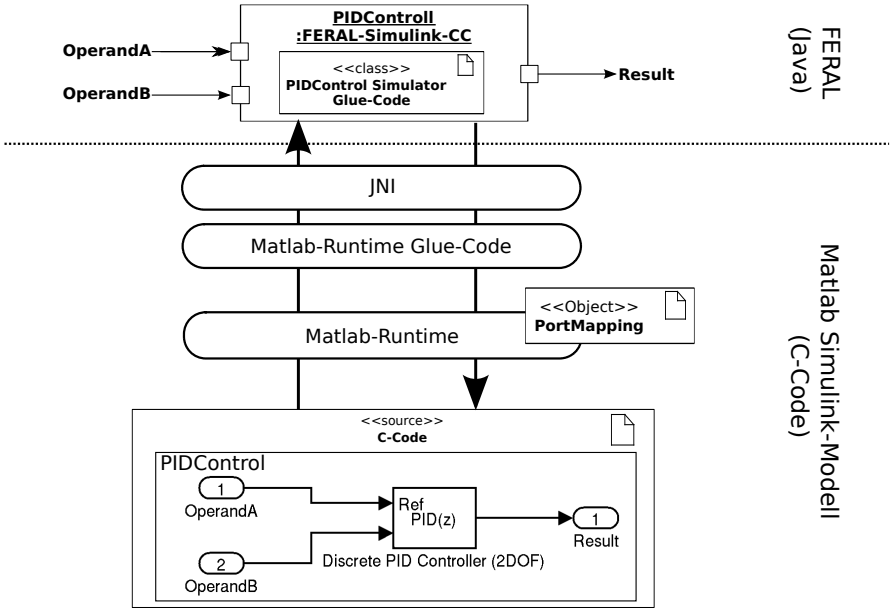


Abbildung 8.6: Instanzierung eines Matlab Simulink-Modells in FERAL.

menwirken, um die Interaktion zwischen FERAL und einem Simulink-Modell zu ermöglichen. Das in Abbildung 8.6 verwendete Simulink-Modell definiert zwei Eingänge (`OperandA` und `OperandB`) und einen Ausgang sowie eine Übertragungsfunktion, welche aus den beiden Eingabewerten einen Ausgabewert (`Result`) ermittelt. Wie in der Abbildung gezeigt, verfügt die Instanz `PIDControl` der FERAL-Simulink-CC (Java-Ebene) für das Simulink-Modell `PIDControl` für jeden Ein- und Ausgangsport des Simulink-Modells über jeweils einen gleichnamigen FERAL-Input- bzw. FERAL-OutputPort. Über diese Ports erfolgt die Interaktion sowie der Austausch von Daten zwischen den verschiedenen konkreten Simulationskomponenten des Simulationssystems.

Die Ansteuerung bzw. Übergabe von Nachrichten von bzw. an die Instanz der FERAL-Simulink-CC an das Simulink-Modell erfolgt unter Verwendung des `PIDControl`-Simulator Glue-Codes. Dieser beinhaltet eine Implementierung des `SimulinkInterface` für das `PIDControl` Simulink-Modell. Über JNI werden die Methodenaufrufe auf dem `SimulinkInterface` zunächst an die Prozeduren des generierten Matlab-Runtime Glue-Code und dann weiter an die Prozeduren der Matlab-Runtime delegiert. Die Matlab-Runtime wiederum führt den aus dem Simulink-Modell generierten und übersetzen Code aus, welcher das eigentliche Verhalten des Modells implementiert. Zusätzlich übernimmt die Matlab-Runtime die Konvertierung zwischen den Nachrichtendatentypen von FERAL sowie den C-Datentypen innerhalb des Simulink-Modells unter Verwendung des generierten Port-Mappings.

8.4 SDL Integration in FERAL

Neben Matlab Simulink unterstützen wir mit SDL (ITU-T's Specification and Description Language [Int12], vgl. Kapitel 5.5) einen weiteren Ansatz zur modellbasierten Entwicklung und Spezifikation von Verhaltensmodellen für FSCs. Da SDL für die modellbasierte Entwicklung reaktiver und verteilter Echtzeitsysteme geeignet ist, ergänzt es in idealer Weise die bereits integrierten Methodiken. SDL verwendet asynchron kommunizierende erweiterte endliche Automaten zur Spezifikation des Verhaltens und verfolgt somit einen zu Simulink komplementären Ansatz[10]. Aufgrund der formalen Syntax und Semantik von SDL können die Spezifikationen automatisiert in Code überführt werden. SDL benötigt für die Ausführung eine Laufzeitumgebung. Diese ist für die Interaktion mit der Zielplattform sowie Ausführung des SDL-Systems (Auswahl von Transitionen, Weiterleiten von SDL-Signalen, etc.) verantwortlich. Um eine neue Zielplattform zu unterstützen genügt es, diese Laufzeitumgebung einmal an die neue Plattform anzupassen. Eine Anpassung der Verhaltensspezifikation des SDL-Systems ist nicht notwendig, um das System auf unterschiedlichen Zielplattformen auszuführen. FERAL ist somit aus Sicht von SDL lediglich eine weitere unterstützte Zielplattform, welche sich von anderen Zielplattformen lediglich durch eine angepasste Laufzeitumgebung unterscheidet. Der aus der Verhaltensspezifikation generierte Code ist hingegen identisch, sodass Simulationen das funktionale Verhalten korrekt widerspiegeln können, sofern nicht-funktionale Eigenschaften dieses nicht zu stark beeinflussen (z. B. Ausführungsverzögerungen einer konkreten Plattform, welche bei FERAL nicht auftreten).

Werkzeuge, die sowohl eine Umgebung für die Spezifikation von SDL-Systemen als auch Codegeneratoren integrieren, sind unter anderem, die IBM Rational SDL Suite [IBMar] oder PragmaDev Real-time Developer Studio (RTDS, [Praar]). FERAL unterstützt für den Entwurf des Verhaltens von FSCs die Rational SDL Suite und verwendet für die Generierung von Code den in der Arbeitsgruppe *Vernetzte Systeme* entwickelten Transpiler ConTraST [FGW06]. Für die Generierung von Code wird die SDL-Spezifikation zunächst in SDL-PR, eine textuelle Repräsentation der graphischen SDL-Spezifikation, umgewandelt und anschließend mit ConTraST in C++-Code überführt. Dieser wird dann zusammen mit der Laufzeitumgebung *SDL Runtime Environment (SdlRE)* und dem *SDL Environment Environment Framework (SEnF)* [FGW06, FGJ+05] zu einem ausführbaren System übersetzt. Bei SdlRE handelt es sich um eine Implementierung der *SDL Virtual Machine (SVM)*, deren Aufgabe die Ausführung eines SDL-Systems gemäß der SDL Semantik ist. Hierzu gehört das Scheduling der SDL Prozesse, der Signalaustausch sowie die Ausführung der entsprechenden Transitionen innerhalb der SDL-Prozesse. SEnF bietet die notwendigen Funktionen an, damit das SDL-System mit der jeweiligen Zielplattform und dessen Umgebung interagieren kann. Bestandteil von SEnF sind sowohl plattformspezifischer Code für die Initialisierung als auch Treiber für die vorhandene Peripherie. Insbesondere kapselt SEnF die plattformspezifischen Funktionalitäten und stellt diese dem SDL-System und SdlRE zur Verfügung. Im Ergebnis wird hierdurch erreicht, dass sowohl SdlRE als auch der aus der SDL-Spezifikation generierte Code unabhängig von der Zielplattform ist.

Bei der BiPS-Integration[11] haben wir RTDS den Vorzug gegeben, da der generierte Code sehr schlank und daher optimal auf eingebettete Systeme zugeschnitten ist. Ein Nachteil des kommerziellen RTDS ist jedoch, dass dessen Quellcode nicht verfügbar ist und sich deshalb Erweiterungen von SDL, wie sie in [2, CBG11, CG12, CBG13] vorgenommen wurden, nicht umsetzen lassen. Hier liegen die Stärken der Kombination von ConTraST, SdlRE und SEnF, da diese Werkzeuge vollständig im Quellcode vorliegen. Da ein Schwerpunkt auch die Evaluation der Erweiterung von SDL mit FERAL bildete, haben wir uns für die Kombination, bestehend aus der IBM Rational SDL Suite, ConTraST, SdlRE und SEnF, entschieden. Basierend auf unseren Erfahrungen mit RTDS [BCGM14] lässt sich jedoch sagen, dass eine Integration in FERAL in ähnlicher Weise umsetzbar wäre.

[10] Zwar erlaubt Simulink mit Stateflow auch die Modellierung mit Hilfe von Automaten, jedoch nur innerhalb des von Simulink definierten zeit- und datenflussorientierten Kontextes.

[11] BiPS (*Black burst-Integrated Protocol Stack*) ist ein in der Arbeitsgruppe *Vernetzte Systeme* entwickelter speziell auf eingebet-

8.4.1 Integration des SDL Simulators in FERAL

Die Integration eines Simulators für die Ausführung von SDL-Systemen erfolgte im Wesentlichen von Dennis Christmann [BCG+13, Chr15], orientiert sich jedoch stark an der Integration von Matlab Simulink aus Kapitel 8.3. Daher beschränken wir uns an dieser Stelle auf eine kurze Zusammenfassung, ohne zu sehr ins Detail zu gehen. Auch bei dieser Integration bedienen wir uns keines klassischen Simulators, welcher eine SDL-Spezifikation *interpretiert*, sondern erzeugen eine direkt von FERAL ausführbare Bibliothek, welche das Verhaltensmodell als generierten C-Code, die Laufzeitumgebung (SdlRe, SEnF) sowie den benötigten Glue-Code enthält. FERAL steuert die Ausführung eines SDL-Systems bzw. der aus dem System erzeugten Bibliothek, über eine spezielle Java-Kontrollkomponente für SDL, vergleichbar mit der FERAL-Simulink-CC für Simulink-Modelle.

Der erste Schritt bei der Integration besteht wieder in der Entwicklung einer Java-Kontrollkomponente. In diesem konkreten Fall setzt sich die Laufzeitumgebung aus SdlRE und SEnF zusammen, welche gemeinsam eine plattformspezifische SVM (PSVM) bilden. Da die PSVM in C++ realisiert ist, wird wieder JNI als Schnittstelle zur Java-Kontrollkomponente verwendet. Wie bei der Integration von Matlab Simulink benötigten wir sowohl auf Java-Seite *Simulator Glue-Code* für die Java-Kontrollkomponente als auch auf der C-Seite entsprechenden *PSVM Glue-Code*, um die Aufrufe an die Funktionen innerhalb der PSVM zu delegieren (vgl. Abbildung 8.4). SEnF implementiert die plattformspezifischen Codeanteile der Laufzeitumgebung und muss entsprechend erweitert werden, damit FERAL als neue Zielplattform unterstützt wird. Nach unseren Anpassungen stellt SEnF[12] vergleichbare Funktionalitäten wie die Matlab-Runtime bereit:

- Bereitstellung einer Schnittstelle, welche die Ausführung des SDL-Systems durch FERAL ermöglicht.
- Speziell angepasste SEnF-Treiber, welche Ports (Input- und OutputPorts) bereitstellen.
- Funktionen zum Zugriff auf die Ports der SEnF-Treiber aus FERAL.
- Konvertierungsfunktionen zwischen SDL-Datentypen und dem Typsystem von FERAL.

Für SDL unterstützen wir sowohl ein zeit- als auch ein ereignisgetriggertes MOCC. Daher existieren zwei unterschiedliche Java-Kontrollkomponenten, um diese beiden Semantiken in FERAL nativ abbilden zu können. In Analogie zu dem vorherigen Kapitel bezeichnen wir die Java-Kontrollkomponenten für SDL als FERAL-SDL-CC. Auch hier kontrolliert eine Instanz der FERAL-SDL-CC die Ausführung eines konkreten SDL-Systems, welches bei der Instanziierung als Parameter übergeben wird.

Der zweite Schritt betrifft die Nachrichtenschnittstelle und besteht sowohl aus der Festlegung der Input- und OutputPorts der Java-Kontrollkomponente als auch den für die Interaktion notwendigen Nachrichten- und Nachrichtendatentypen. Da FERAL aus Sicht des SDL-Systems die Umgebung (Environment) simuliert, in der dieses ausgeführt wird, beginnen wir zunächst mit der Betrachtung der Konzepte von SDL für die Interaktion mit seiner Umgebung. Innerhalb eines SDL-Systems erfolgt die Interaktion mit der Umgebung mittels Signalen an das bzw. von dem Environment. Hierbei handelt es sich um einen speziellen SDL-Prozess, welcher bei unserer Laufzeitumgebung durch SEnF implementiert wird. SEnF selbst besteht aus dedizierten Treibern, welche die SDL-Signale verarbeiten und über diese z. B. die Ansteuerung der Peripherie der Zielplattform ermöglichen. Ein solcher SEnF-Treiber besteht aus zwei Teilen: Einer Menge von SDL-Signaldefinitionen, über die das SDL-System mit dem Treiber interagieren kann, sowie eine Implementierung der Treiberfunktionalität für die jeweilige Plattform.

Bei unserem Konzept definiert jeder SEnF-Treiber seine eigene Schnittstelle in Form von Input- und OutputPorts, über die er mit dem Rest des Simulationssystems interagiert. Jeder SEnF-Treiber kann, sofern erforderlich, für die Interaktion eigene Nachrichten- und Nachrichtendatentypen definieren, muss

tete Systeme zugeschnittener Protokollstack, mit dem Ziel echtzeitfähige, deterministische Kommunikationsprotokolle für drahtlose (Sensor-)Netzwerke bereitzustellen (vgl. Kapitel 5).

[12] Die nachfolgend beschriebenen Erweiterungen decken sämtliche Anpassungen ab, nicht nur diejenigen, die für die Umsetzung des ersten Integrationsschrittes notwendig sind.

jedoch entsprechende Konvertierungsfunktionen bereitstellen, um die eigenen SDL-Signale auf spezifische FERAL-Nachrichten und umgekehrt abzubilden.

Beim Instanziieren der FERAL-SDL-CC mit einem Verhaltensmodell (SDL-System) werden zunächst die aktiven SEnF-Treiber innerhalb des SDL-Systems bestimmt. Dann werden mittels der Funktionen der PSVM die von diesen Treibern definierten Input- und OutputPorts ermittelt. Die Instanz der FERAL-SDL-CC legt für jeden ermittelten Input- bzw. OutputPort einen gleichnamigen FERAL-Input- bzw. FERAL-OutputPort an. Ein- bzw. ausgehende Nachrichten werden zwischen den gleichnamigen Ports der SEnF-Treiber bzw. der Instanz der FERAL-SDL-CC ausgetauscht. Über die Ports der FERAL-SDL-CC erfolgt über Links die eigentliche Interaktion mit den anderen Simulationskomponenten.

Der dritte Schritt der Integration betrifft die Instanziierung der FERAL-SDL-CC innerhalb eines Simulationssystems sowie der damit verbundenen Zuweisung des Verhaltensmodells (SDL-Systems). Hierzu wird der mittels ConTraST generierte C++-Code des SDL-Systems zusammen mit den Quellen von SDLRe, SEnF sowie dem generierten PSVM Glue-Code übersetzt und zu einer Bibliothek gelinkt. Diese wird beim Instanziieren der FERAL-SDL-CC als Parameter übergeben und geladen. Die Instanz der FERAL-SDL-CC repräsentiert das SDL-System innerhalb von FERAL und gestattet dem Direktor die Kontrolle über die Ausführung des SDL-Systems. Die anderen Simulationskomponenten können über die Input- und OutputPort der instanziierten FERAL-SDL-CC mit dem SDL-System interagieren.

Die Erzeugung der benötigten Glue-Codes sowohl auf Java- (Simulator Glue-Code) wie auch C-Seite (PSVM Glue-Code) wurde ebenso wie die Erstellung der Bibliothek automatisiert und liefert als Ergebnis ein OSGi-Bundle. Dieses enthält das vollständige Verhaltensmodell (Bibliothek und Simulator Glue-Code). Das Zusammenwirken der verschiedenen Schichten ist in Abbildung 8.7 anhand eines Beispiels dargestellt.

8.4.2 Portierte SEnF-Treiber für FERAL

Die Schnittstelle zwischen dem SDL-System und anderen konkreten Simulationskomponenten wird durch die aktiven SEnF-Treiber bereitgestellt. Diese definieren Input- und OutputPorts sowie geeignete Nachrichten- und Nachrichtendatentypen. Wir haben SEnF um spezielle Treiber für FERAL erweitert, welche die Interaktion mit den für FERAL entwickelten CSCs (CAN, FlexRay sowie dem Simulator für das abstrakte Kommunikationsmodell (vgl. Kapitel 9) erlauben. Des Weiteren stellen wir einen generischen SEnF-Treiber zur Verfügung, welcher den generischen Datenaustausch sowie die Interaktion zwischen SDL-Systemen und anderen konkreten Simulationskomponenten ermöglicht. Die Entwicklung dieser SEnF-Treiber erfolgte wieder im Kontext dieser Arbeit.

Eine Anwendung der Treiber für das abstrakte Kommunikationsmodell, CAN sowie FlexRay, findet sich in unserem Adaptive Cruise Control System in Kapitel 11. Der Treiber für die direkte Interaktion kommt in unserem Simulationssystem für das inverse Pendel zum Einsatz (vgl. Kapitel 12.3). Weitere SEnF-Treiber existieren für die Anbindung von SDL-Systemen an den ns-3 sowie für die Simulation des CC2420-Transceivers, welche ebenfalls Bestandteile des ns-3 sind und von Anuschka Igel realisiert wurden [Ige15, Gro12, IG13, BCG+13].

8.4.2.1 SEnF-Treiber für FlexRay

Der *FlexRay-SEnF-Treiber* für FERAL ermöglicht die Verwendung des FlexRay-Simulators aus SDL heraus. Er stellt hierfür gegenüber dem SDL-System die gleiche Schnittstelle (Signaldefinitionen) wie der von Matthias Wiebel entwickelte FlexRay-Treiber [BGW10] für die Ansteuerung eines realen FlexRay Kommunikationscontrollers zur Verfügung. Dies gestattet es, ein SDL-System, welches FlexRay verwendet, ohne Modifikationen der Spezifikation sowohl mittels FERAL zu simulieren als auch auf der realen Hardwareplattform auszuführen.

Der SEnF-Treiber beschränkt sich auf die wichtigsten SDL-Signale (z. B. für Übertragung bzw. Empfang eines Rahmens) sowie die Zustandssignalisierung (Verlassen des Startup-Zustandes, Beginn eines neuen Kommunikationszyklus, ...) des CC und bildet diese auf die von dem FlexRay-Simulator bereitgestellten Nachrichten- und Nachrichtendatentypen ab. Ein SDL-System, welches den FlexRay-SEnF-Treiber verwendet, verfügt über zwei Ports (je ein Input- und ein OutputPort) für den Austausch der FlexRay spezifischen Nachrichtentypen mit dem FlexRay-Simulator – `flexray_tx` und `flexray_rx`.

SEnF-Treiber für CAN

Der SEnF-Treiber für CAN existiert nur für die Plattform FERAL und ermöglicht nur die Anbindung eines SDL-Systems an den CAN-Simulator, nicht aber die Nutzung realer Hardware. Daher ist die Schnittstelle zwischen SDL-System und SEnF-Treiber minimalistisch gestaltet und besteht nur aus den SDL-Signalen `CAN_RX` und `CAN_TX` sowie Datenstrukturen zur Repräsentation des CAN-Rahmens innerhalb von SDL.

Die beiden SDL-Signale werden durch den *CAN-SEnF-Treiber* für FERAL auf die entsprechenden Nachrichtentypen des CAN-Simulators zum Senden bzw. Empfangen von CAN-Rahmen abgebildet. Die Kodierung des CAN-Rahmens erfolgt mit den speziellen Nachrichtendatentypen des CAN-Simulators. Der CAN-SEnF-Treiber verwendet die beiden Ports `can_tx` und `can_rx` für den Austausch von Nachrichten mit dem CAN-Simulator.

SEnF-Treiber für abstrakte Kommunikationsmodelle

Die Anbindung eines SDL-Systems an den Simulator für das abstrakte Kommunikationsmodell (ACOM-Simulator) erfolgt über den *ACOM-SEnF-Treiber*. Auch dieser ist nur für FERAL verfügbar und nahezu identisch mit dem CAN-SEnF-Treiber. Der ACOM-SEnF-Treiber verwendet die beiden Ports `acom_tx` und `acom_rx` für die Interaktion mit dem ACOM-Simulator.

SEnF-Treiber für die direkte Interaktion mit anderen FERAL Simulationskomponenten

Die bislang vorgestellten SEnF-Treiber für FERAL sind auf die Interaktion mit bestimmten, in FERAL eingebundenen Simulatoren beschränkt. Eine direkte Interaktion (durch den Austausch von beliebigen Nachrichten über Links) mit beliebigen konkreten Simulationskomponenten wird durch diese nicht unterstützt.

Der *ForwardSignals-SEnF-Treiber* schließt diese Lücke und gestattet den direkten Austausch von Nachrichten und Daten zwischen einem SDL-System und beliebigen anderen konkreten Simulationskomponenten (z. B. FSCs auf Basis von Simulink-Modellen) sowie die Verwendung von Bridges zur indirekten Interaktion mit CSCs (vgl. Kapitel 10.2). Für die Interaktion und den Datenaustausch verwenden wir Nachrichten des Typs `ValueMessage`, welcher fester Bestandteil von FERAL ist. Innerhalb der Nachricht wird jeweils nur ein Datum übermittelt, wobei wir uns für dessen Repräsentation auf das von FERAL bereitgestellte Typsystem beschränken. Auf diese Weise wird sichergestellt, dass eine Interaktion mit beliebigen Simulationskomponenten möglich ist, da nur die Standardtypen von FERAL zum Einsatz kommen.

Als Schnittstelle zu anderen Simulationskomponenten stellt der ForwardSignals-SEnF-Treiber eine feste (aber konfigurierbare) Anzahl von Input- und OutputPorts zur Verfügung, die innerhalb des SDL-Systems beliebig für den Austausch von Nachrichten genutzt werden können. Hierfür sind die entsprechenden Ports (der FERAL-SDL-CC) ihrerseits über Links mit den Ports anderer Simulationskomponenten verbunden, sodass eine Interaktion ermöglicht wird. Da der ForwardSignals-SEnF-Treiber nicht auf eine bestimmte Anwendung zugeschnitten ist, hängt die Semantik der über die Ports ausgetauschten Nachrichten und Daten von dem konkreten SDL-System ab. Aus diesem Grund verwendet der ForwardSignals-SEnF-Treiber ein generisches Namensschema für seine Ports, um deren Verwendung innerhalb von FERAL

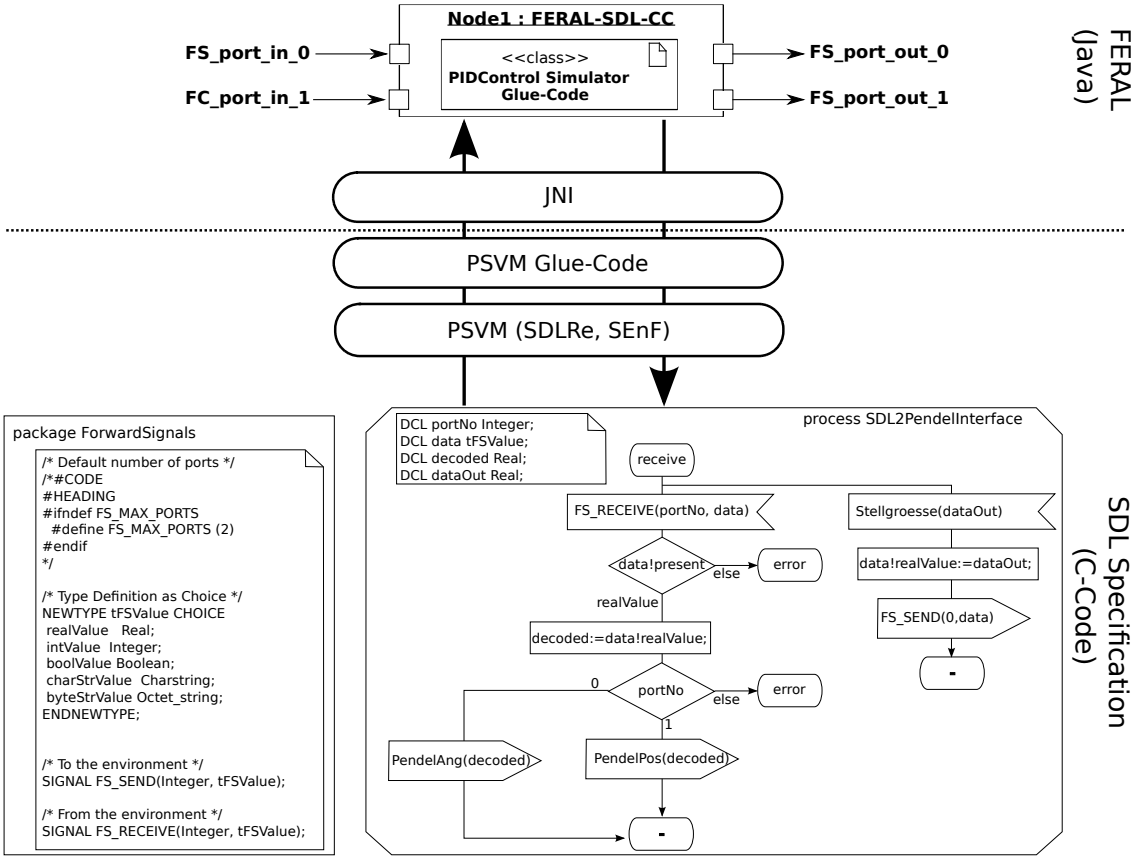


Abbildung 8.7: Verwendung des ForwardSignals-SEnF-Treibers in SDL.

zu erleichtern: Die Input- und OutputPorts werden fortlaufend nummeriert und tragen die Bezeichnung `FS_port_in_X` bzw. `FS_port_out_X`, wobei `X` die Nummer des jeweiligen Ports ist und `in` bzw. `out` die Kommunikationsrichtung aus Sicht des SDL-Systems angibt.

Die Schnittstelle zwischen dem SDL-System und dem ForwardSignals-SEnF-Treiber bilden die beiden SDL-Signale `FS_SEND` und `FS_RECEIVE`. Diese verfügen über zwei Parameter; der erste gibt jeweils die Nummer des Input- bzw. OutputPorts an, über die der im zweiten Parameter übergebene Wert übertragen werden soll bzw. empfangen wurde. Unterstützt werden die SDL-Datentypen `Real`, `Integer`, `Boolean`, `Charstring` und `Octet_string` für den Datenaustausch mit FERAL, welche auf die entsprechenden Typen des FERAL Typsystems abgebildet werden.

Die Abbildung 8.7 illustriert den Einsatz des ForwardSignals-SEnF-Treibers innerhalb eines SDL-Systems sowie die Definition der SDL-Signale `FS_SEND` und `FS_RECEIVE` des Treibers. Zusätzlich ist die Beziehung zwischen dem für das konkrete SDL-System instanziierten FERAL-SDL-CC und dessen Input- und OutputPorts in Relation zu den aktiven Treibern von SEnF dargestellt. Hier wird lediglich der ForwardSignals-SEnF-Treiber verwendet, sodass die Instanz der FERAL-SDL-CC nur über jeweils zwei Input- und OutputPorts verfügt. In dem gezeigten Ausschnitt des SDL-Systems ist die Interaktion mit einer mittels Matlab Simulink realisierten Simulationskomponente für eine Regelstrecke eines inversen Pendels dargestellt. Wir zeigen exemplarisch den SDL-Prozess `SDL2PendelInterface`, welcher für den Austausch von Sensor- und Stellwerten mittels des ForwardSignals-SEnF-Treibers mit der Regelstrecke des inversen Pendels verantwortlich ist. Dieser Prozess ist Bestandteil der SDL-Spezifikation unseres Reglers für das inverse Pendel. Eine detaillierte Beschreibung des inversen Pendels sowie der anderen Komponenten des Simulationssystems ist in Kapitel 12.3 zu finden.

9 Communication Simulation Components

Von zentraler Bedeutung bei der Entwicklung verteilter (Echtzeit-)Systeme ist die Wahl einer geeigneten Kommunikationstechnologie für die Vernetzung der einzelnen Knoten bzw. Funktionalitäten. Da die Interaktion der Knoten auf der Kommunikation basiert, können sich nichtfunktionale Eigenschaften der Kommunikationslösung (Verzögerungen, Verfälschungen oder der Verlust von Nachrichten) in komplexer Form auf die Funktionalität und das Verhalten des Gesamtsystems auswirken. So können stark verzögerte Nachrichten z. B. bei einem verteilten Regelungssystem dazu führen, dass die Regelgüte abnimmt oder die Regelung komplett versagt.

Um hieraus resultierendes unerwünschtes Verhalten oder Fehler zu vermeiden, müssen die spezifischen Eigenschaften der gewählten Kommunikationslösung bereits in den frühen Phasen der Entwicklung berücksichtigt werden. Auch im weiteren Entwicklungsprozess helfen Simulationen, die Eignung der gewählten Lösung zu bewerten. Aus diesem Grund stellt FERAL Communication Simulation Components (CSCs) für verschiedene Kommunikationstechnologien zur Verfügung, mit denen zusätzlich zum funktionalen Verhalten der FSCs auch deren Kommunikation simuliert werden kann. Der Begriff CSCs subsumiert hierbei alle Arten von Simulatoren, die in der Lage sind, Kommunikationstechnologien innerhalb von FERAL zu simulieren, um die verteilte Interaktion von FSCs über diese möglichst realistisch widerzuspiegeln.

Beim Einsatz von CSCs beschreibt das abstrakte Simulationssystem nicht nur die direkte Interaktion des FSCs, sondern auch die logische Kommunikationstopologie, d.h., welche FSCs über einen bestimmten Bus miteinander verbunden sind. Das konkrete Simulationssystem legt, neben den zu verwendenden Simulatoren für die FSCs sowie deren Verhaltensmodellen, auch die Kommunikationstechnologie für die verwendeten CSCs (vgl. Kapitel 8.1) fest. Auf diese Weise können die Eigenschaften und Auswirkungen von Kommunikationslösungen bereits zu Beginn der Entwicklung der FSCs berücksichtigt werden. Dies erlaubt die kontinuierliche Evaluation mittels Simulationen während der gesamten fortschreitenden Entwicklung.

FERAL unterstützt den Entwickler auch bei der Evaluation von Designalternativen hinsichtlich geeigneter Kommunikationslösungen. Voraussetzung hierfür ist ein möglichst einfacher Austausch der verwendeten Kommunikationslösung gegen eine andere. Dies wird in FERAL durch die Verwendung von speziellen *Bridge*-Komponenten erreicht, die FSCs und CSCs miteinander verbinden. Mit deren Hilfe kann das Kommunikationsverhalten von FSCs spezifiziert werden, ohne deren Verhaltensmodelle anpassen oder diese mit spezifischem Wissen in Bezug auf die verwendeten Kommunikationstechnologien anreichern zu müssen. Die Bridges sind in FERAL als eigenständige Simulationskomponenten realisiert und mit den FSCs sowie mit den CSCs über Links verbunden; eine direkte Verbindung zwischen den FSCs und den CSCs (z. B. direkte Links in FERAL) gibt es in diesem Fall nicht. Generische *Gateway*-Komponenten leisten die gleiche Funktionalität für die Verbindung zweier CSCs miteinander. Diese ermöglichen die Spezifikation des Weiterleitungsverhaltens, ohne dass der Benutzer die Funktionalität selbst implementieren muss. Die indirekte Anbindung von FSCs und CSCs über Bridges resp. Gateways erlaubt es bei der Erstellung des konkreten Simulationssystems, aus dem abstrakten Simulationssystem beliebige Kommunikationstechnologien für die definierten CSCs zu wählen, ohne dass die Verhaltensmodelle der FSCs angepasst werden müssen. Dementsprechend gering ist der Aufwand für die Evaluation von Designentscheidungen hinsichtlich einer geeigneten Kommunikationstechnologie. Würde man, entgegen des hier vorgestellten Lösungsansatzes, bei den Modellen der FSCs nicht sauber zwischen rein funktionalem Verhalten und Kommunikationsverhalten trennen, so müssten die Verhaltensmodelle der FSCs für jede

neu zu evaluierende CSC (Kommunikationstechnologie) modifiziert werden. Entsprechend hoch wäre der Aufwand für die Evaluation von Designalternativen.

Voraussetzung für den einfachen Austausch der CSCs ist das Vorhandensein einer einheitlichen gemeinsamen Schnittstelle. Nur so werden auch die erforderlichen Anpassungen des Java-Codes, welcher das konkrete Simulationssystem in einer von FERAL ausführbaren Art beschreibt, bei dem Austausch einer CSC möglichst klein gehalten. Aus diesem Grund sind die CSCs nach dem Abstract Factory Pattern aufgebaut, d.h., die gesamte Architektur der CSCs ist auf Austauschbarkeit ausgelegt. Das Abstract Factory Pattern beschränkt sich hierbei nicht nur auf einfache Factory Methoden und abstrakte Schnittstellen, sondern umfasst ein gemeinsames Konfigurationskonzept, Fehlermodell sowie eine einheitliche Schnittstelle zur Protokollierung und Visualisierung der simulierten Kommunikation.

Insgesamt stehen dem Entwickler aktuell drei verschiedene Simulatoren für CSCs zur Verfügung, die speziell für FERAL entwickelt wurden. Diese Simulatoren stellen die Kommunikationstechnologien CAN, FlexRay (Version 2.1a sowie Version 3.0.1) sowie ein abstraktes Kommunikationsmodell für die Simulation bereit. Das abstrakte Kommunikationsmodell kann als Platzhalter für frühe Entwürfe und Entwicklungsphasen benutzt werden bzw., um die Auswahl einer konkreten Kommunikationstechnologie hinauszuzögern. Auf diese Weise können dennoch Effekte, wie Übertragungsverzögerungen oder zu hohe Last, auf dem virtuellen Medium in der Simulation abgebildet werden. Zusätzlich wurde der Network-Simulator 3 (ns-3, [ns3ar]) in FERAL integriert. Dieser ist auf IP-basierte Kommunikationsansätze, wie IEEE 802.11 (WLAN) oder IEEE 802.3 (Ethernet) [Ins12], spezialisiert, jedoch nicht darauf beschränkt. Die Integration von ns-3 in FERAL erfolgte durch Anuschka Igel [BCG+13]. Des Weiteren wurde die Simulation des CC2420-Transceivers für IEEE 802.15.4 [IEE03] in ns-3 integriert, sodass eine Simulation des Kommunikationsverhaltens der *Imote 2*-Plattform ermöglicht wird [IG13]. Um auch die durch den ns-3 bereitgestellten Kommunikationstechnologien auf die gleiche Weise austauschbar zu gestalten, wie dies bei den bereits genannten Simulatoren der Fall ist, wurde ein entsprechender Wrapper entwickelt, der unter anderem auch die Nutzung von Bridges und Gateways erlaubt.

Die ersten Kapitel dieses Abschnitts beschäftigen sich mit der Vorstellung der CSC-übergreifenden Konzepte und stellen die getroffenen Architektur- und Designentscheidungen vor, die bei der Entwicklung der drei nativen (speziell für FERAL entwickelten) CSCs angewendet wurden, um deren Austauschbarkeit zu gewährleisten. Erläutert und illustriert werden diese jeweils anhand ihrer Umsetzung für die CAN-CSC. Kapitel 9.1 widmet sich der Fragestellung der Austauschbarkeit und stellt die grundlegenden Konzepte vor. Die Nutzung und Rolle des *Abstract Factory Pattern* in diesem Kontext wird in Kapitel 9.2 beschrieben. Anschließend widmen wir uns der Datenschnittstelle zwischen CSCs und FERAL (Kapitel 9.3) und betrachten dann in Kapitel 9.4 den internen Aufbau der CSCs. Zusätzliche Erweiterungen für die Simulation von Übertragungsfehlern und die Protokollierung von Abläufen werden in Kapitel 9.5 und 9.6 diskutiert. Danach (Kapitel 9.7) widmen wir uns den einzelnen CSCs noch einmal im Detail und betrachten abschließend die Integration von ns-3 in FERAL mit Fokus auf dem Wrapper, welcher die Interoperatiblität und Austauschbarkeit mit den anderen CSCs verbessert.

Die Ergebnisse dieses Kapitel wurden in [5, 10] publiziert.

9.1 Architektur der CSCs

Die Austauschbarkeit der einzelnen CSCs wird durch die Definition einer gemeinsamen Schnittstelle erreicht, die nach dem Vorbild des *Abstract Factory*-Entwurfsmusters gestaltet wurde. Die Abbildung 9.1 aus [GHJV09] zeigt die allgemeine Struktur dieses Entwurfsmusters. Um eine konkrete CSC durch eine andere CSC (z. B. die CAN-CSC durch die FR-CSC[1]) innerhalb des Simulationsszenarios zu ersetzen genügt es, die für die Instanziierung verwendete *ConcreteFactory* auszutauschen. An allen anderen Stellen im Code werden die konkreten Produkte über deren gemeinsamen abstrakten Schnittstellen referenziert

[1] FR-CSC steht für FlexRay-CSC und simuliert einen FlexRay-Bus.

(vgl. `Client` in Abbildung 9.1), sodass keine Modifikationen erforderlich sind. In dem genannten Beispiel würde es daher genügen, anstelle der `ConcreteFactory` der CAN-CSC die `Concrete-Factory` der FR-CSC für die Erzeugung der konkreten Simulationskomponenten der CSC sowie deren untergeordneten Komponenten zu instanziieren.

Die alleinige Instanziierung des Simulators ist jedoch nicht ausreichend, zusätzlich werden Konfigurationsparameter benötigt, welche für die Kommunikationstechnologie spezifisch sind (beispielsweise bei CAN die Übertragungsrate oder bei FlexRay der Aufbau des Kommunikationszyklus). Daher werden sowohl die für die Konfiguration sowie das Fehlermodell notwendigen Klassen ebenso wie die Nachrichtentypen im Abstract Factory-Pattern der CSCs berücksichtigt. Da die Definition eines konkreten Simulationsszenarios für FERAL in Form von Java-Code erfolgt, verzichten wir darauf, die einzelnen Konfigurationsparameter oder auch das Fehlermodell direkt innerhalb des Java-Codes mit Werten zu initialisieren. Dies würde nur den Code aufblähen und den Austausch einer CSC unnötig aufwändig gestalten. Daher lagern wir die Konfigurationen der CSCs in XML-Dateien aus, die automatisiert eingelesen und ausgewertet werden (Details hierzu in Kapitel 10.5). Die definierten Klassen, welche die Konfiguration betreffen, dienen lediglich der Repräsentation der geladenen Konfigurationen zur Laufzeit.

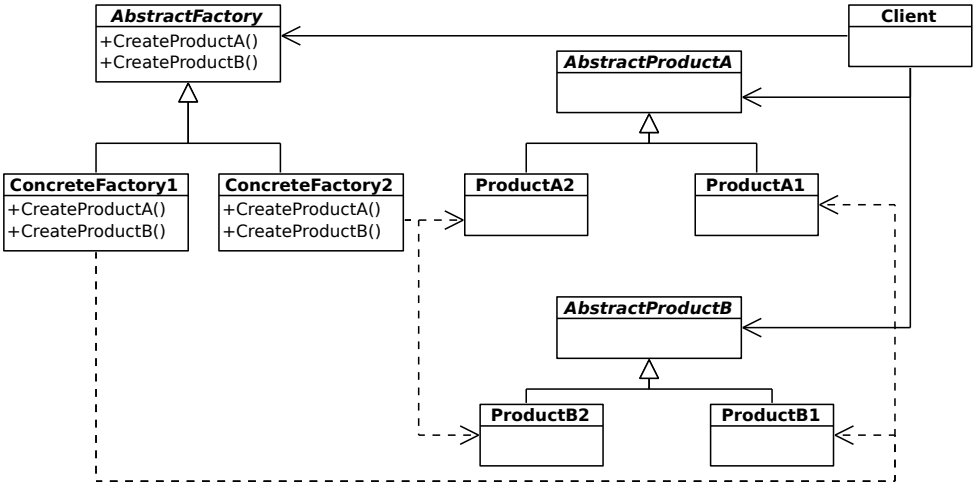


Abbildung 9.1: Aufbau des Abstract Factory-Pattern [GHJV09].

9.2 Anwendung des Abstract Factory-Pattern für CSCs

Die Definition der gemeinsamen Schnittstellen und abstrakten Klassen der CSC, die zum Abstract Factory-Pattern gehören, erfolgt innerhalb des separaten OSGi-Bundle `COM_Generic`. In diesem Bundle sind auch gemeinsame Funktionalitäten der CSCs hinterlegt. Die Implementierung der definierten Schnittstellen für die jeweiligen Simulatoren erfolgt dann in eigenen OSGi-Bundles. Hierdurch ist die Implementierung jeder CSC in einem eigenen OSGi-Bundle gekapselt, welches nur von dem `COM_Generic`-Bundle und FERAL abhängig ist.

Die Abbildung 9.2 skizziert den Aufbau des Abstrakt Factory-Pattern für die CSCs, wobei wir uns hier auf die Darstellung der wichtigsten Klassen beschränken, welche die nach außen hin sichtbare Schnittstelle des Bundles (der CSC) bilden. Deren Funktion und näheren Zusammenhänge beleuchten wir in den folgenden Teilkapiteln. Da es zunächst nur um einen ersten Überblick geht, zeigt die Abbildung 9.2 nur die konkreten Produkte der FlexRay- und CAN-CSC. Äquivalente Klassen, entsprechend diesem

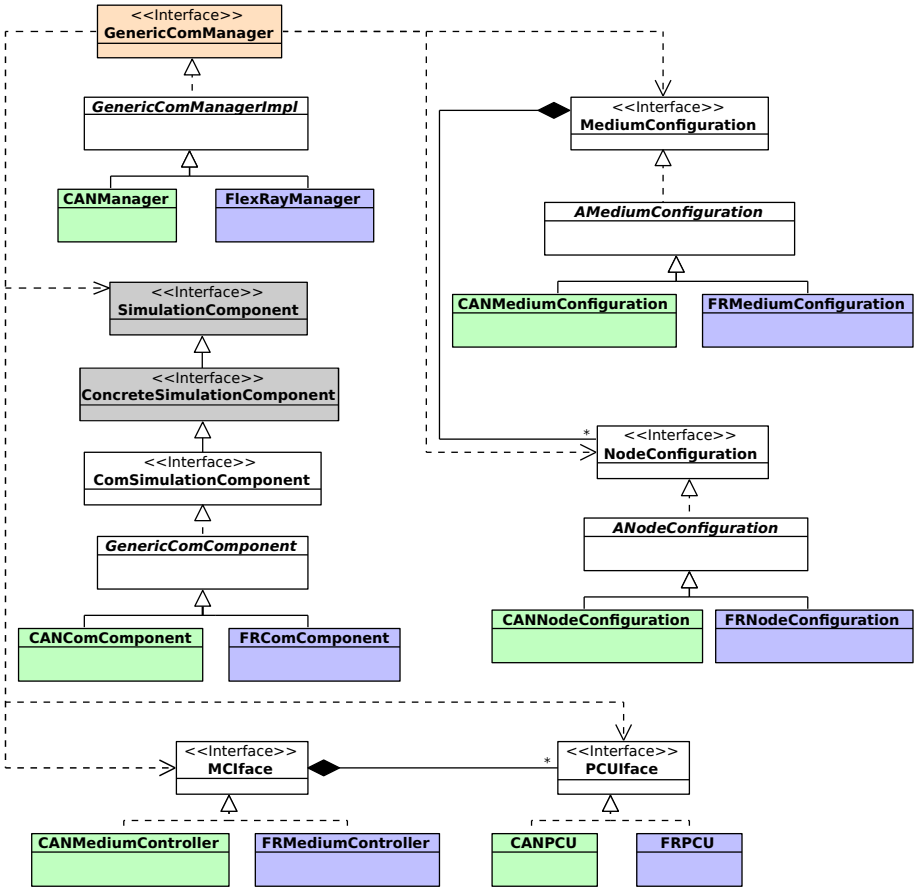


Abbildung 9.2: UML-Klassendiagramm der Architektur der CSCs.

Namensmuster[2], existieren sowohl für das abstrakte Kommunikationsmodell `ACOM` als auch den ns-3-Wrapper.

Das UML-Diagramm 9.2 zeigt Elemente (Klassen und Interfaces) aus insgesamt vier verschiedenen OSGi-Bundles. Die weiß bzw. orange eingefärbten Elemente gehören zu dem `COM_Generic`-Bundle und definieren die abstrakten Produkte (weiß), im Sinne der Abstract Factory bzw. die abstrakte Fabrik selbst (orange). Die grünen Elemente sind Bestandteil der CAN-CSC (`COM_CAN`-Bundle), wohingegen die violetten der FR-CSC (Bundle `COM_FR`) angehören. Die grau hinterlegten Elemente stammen direkt aus FERAL. Die Beschreibung der Schnittstelle der abstrakten Fabrik (`GenericComManager` – orange) sowie der von ihr erzeugten *abstrakten Produkte* erfolgt mit Hilfe von Java-Interfaces. Soweit möglich, wird nur gegen die Interfaces programmiert, sodass eine Abhängigkeit zu konkreten Implementierungen vermieden wird. Dies gilt auch insbesondere in Bezug auf die Signatur der Factory-Methoden, z. B. innerhalb des `GenericComManager`.

Die Schnittstelle `GenericComManager` wird von der abstrakten Klasse `GenericComManagerImpl` implementiert. Diese stellt gemeinsame, von allen CSCs benötigte Funktionalitäten bereit und dient als Basisklasse für die konkreten Fabriken der CSCs (hier `CANManager` bzw. `FlexRayManager`). Beide implementieren die in `GenericComManager` definierten Fabrikmethoden zur Instanziierung der in ihrem Bundle enthaltenen Implementierungen der *abstrakten Produkte*. Als Basis für Java-Kontrollkomponenten für die CSCs dient die abstrakte Klasse `GenericComComponent`. Jede CSC

[2]Jede CSC verfügt über einen eigenen Namensraum, beginnend mit `com.X.`, wobei *X* für die simulierte Kommunikationstechnologie bzw. den Namen des Simulators steht.

bietet eine eigene Implementierung einer Java-Kontrollkomponente an, welche die abstrakte Klasse `GenericComComponent` verfeinert und um Protokoll- bzw. CSC-spezifisches Verhalten erweitert.

Das grundlegende Konzept, ein abstraktes Produkt zunächst anhand einer Schnittstelle (Java-Interface) zu definieren und anschließend gemeinsames Verhalten in einer abstrakten Klasse bereitzustellen, welches dann von den CSCs verfeinert wird, behalten wir auch für die Repräsentation der Konfiguration bei. Objekte des Typs `MediumConfiguration` und `NodeConfiguration` bilden die Konfiguration des jeweiligen Busses oder eines Knotens zur Laufzeit ab. Eine `MediumConfiguration` kapselt die Konfiguration, welche das Medium direkt betrifft und gilt für alle an einen Bus angebundenen Knoten (z. B. die Übertragungsrate). In der `NodeConfiguration` sind individuelle Konfigurationsparameter der einzelnen Knoten hinterlegt (wie z. B. bei FlexRay die Konfiguration der Nachrichtenpuffer).

In Analogie zu der Konfiguration wird auch bei der Implementierung des Verhaltens zwischen dem eines einzelnen Knotens sowie dem Zusammenspiel der Knoten über ein gemeinsames Medium unterschieden. Die Schnittstelle `PCUIface` (PCU – Protocol Control Unit) und deren Implementierungen simulieren das Verhalten eines einzigen Knotens, während der Medium-Controller (Schnittstelle `MCIface`) das sich aus der Protokollspezifikation sowie dem geteilten Medium ergebende, knotenübergreifende Verhalten nachbildet. Hierzu gehört etwa bei CAN, der Ablauf der Arbitrierung oder die Simulation von Kollisionen bei FlexRay, um nur einen kleinen Ausschnitt zu nennen.

9.3 Schnittstelle zwischen CSC und FERAL

Bei der Implementierung der CSCs wird zwischen dem Verhalten eines einzelnen Knotens und dem Verhalten, welches sich aus dem Zusammenwirken der Knoten über das gemeinsame Medium ergibt, unterschieden. Ziel ist es, die reale Struktur eines verteilten (eingebetteten) Systems sowie dessen Verhalten möglichst exakt simulieren zu können. Bei einem realen eingebetteten System, welches über einen Bus kommuniziert, wird auf dem Knoten – bestehend aus einem Mikrocontroller oder Steuergerät – eine Anwendung ausgeführt, welche das knotenlokale Verhalten realisiert. Die Anbindung dieses Knotens an das jeweilige Kommunikationssystem erfolgt über einen Kommunikationscontroller, welcher die komplette Protokolllogik implementiert und entweder direkt in den Mikrocontroller integriert oder extern angebunden ist. Neben dem durch das Protokoll vorgegebenen Verhalten implementiert der Kommunikationscontroller auch noch eine Schnittstelle für die Anwendung, deren exakte Ausgestaltung sowohl von der Protokollspezifikation als auch von dem Hersteller beeinflusst wird. Dadurch unterscheiden sich die Bausteine in Funktionsumfang, Ansteuerung und teilweise dem Verhalten. Hinzu kommt, dass unterschiedlich gewählte Konfigurationsparameter für die Kommunikationscontroller der Knoten ebenfalls Auswirkungen auf das Verhalten haben[3].

Durch die Unterscheidung zwischen Medium-Controller und PCU besteht die Möglichkeit, genau diesen Aspekt abbilden zu können. Jede instanziierte PCU simuliert das Verhalten eines einem bestimmten Knoten zugeordneten Kommunikationscontrollers. Abhängig vom Abstraktionsgrad der Simulation kann so z. B. zwischen verschiedenen Implementierungen gewählt werden, welche das Verhalten und die Eigenarten unterschiedlicher Kommunikationscontroller imitieren. Hierzu wird die zu instanziierende PCU-Implementierung in der `NodeConfiguration` (Abbildung 9.2) festgelegt ebenso wie evtl. weitere Konfigurationsparameter, welche den Kommunikationscontroller betreffen.

Der Medium-Controller simuliert das Ergebnis des Zusammenwirkens und der Interaktion der einzelnen Knoten über das gemeinsame Medium gemäß der Protokollspezifikation. Bei der Realisierung kann der Medium-Controller sein globales Wissen nutzen, um die Simulation zu beschleunigen, da anders als in der Realität der Zeitpunkt, zu dem jede PCU auf das Medium zugreift, sowie die ausgeführten Aktionen bekannt sind. Somit ist das Resultat der Interaktion vorhersagbar, ohne die Abläufe auf dem gemeinsamen Medium auf physikalischer Ebene nachzubilden. Stattdessen genügt in der Regel eine Simulation auf

[3] Ein einfaches Beispiel ist die Festlegung der maximalen Anzahl von Übertragungsversuchen, z. B. bei CAN.

der Ebene der übertragenen Frames. Dies reduziert sowohl den Aufwand bei der Implementierung als auch die Simulationsdauer. Hierbei gestattet das globale Wissen des Medium-Controllers es aber dennoch, die für die Kommunikation relevanten Aspekte wie Kollisionen, Übertragungsverzögerungen etc. korrekt zu bestimmen[4]. Dieser Ansatz wurde bei den drei direkt für FERAL entwickelten Simulatoren für CAN, FlexRay und das abstrakte Kommunikationsmodell umgesetzt. ns-3 verwendet, zumindest in Teilen, ein vereinfachtes physikalisches Modell für die Bestimmung der Signalausbreitung bei drahtloser Kommunikation, ist jedoch intern anders aufgebaut.

Eine PCU-Instanz simuliert ausschließlich das Verhalten eines Kommunikationscontrollers, während die Anwendung, die auf dem Steuergerät/Knoten abläuft bzw. deren Verhalten innerhalb von FERAL durch eine eigenständige FSC simuliert wird. Für jeden an das Kommunikationssystem angebundenen Knoten wird innerhalb der instanziierten CSC wiederum eine PCU-Instanz als Stellvertreter erzeugt. In FERAL wird ein konkreter Bus[5] durch eine Instanz der Java-Kontrollkomponenten der jeweiligen CSC repräsentiert, welche ihrerseits von der Klasse `GenericComComponent` abgeleitet ist. Über die implementierte `SimulationComponent`-Interface steuert FERAL dann die Simulation des Busses. Die beschriebene interne Struktur sowie die Abläufe innerhalb der CSCs (Medium-Controller, PCUs) sind für FERAL nicht sichtbar und vollständig durch die `GenericComComponent` gekapselt.

Der Austausch von Nachrichten zwischen CSCs und FSCs erfolgt über reguläre Input- und OutputPorts. Für jede mit der CSC verbundene FSC stellt die `GenericComComponent` einen eigenen Input- und OutputPort zur Verfügung; die Anzahl der benötigten Ports wird hierbei auf Basis der `MediumConfiguration` und der darin enthaltenen `NodeConfiguration`s bestimmt. Die `NodeConfiguration` legt zusammen mit den individuellen Konfigurationsparametern auch einen für den simulierten Bus eindeutigen Bezeichner[6] fest. Die Input- bzw. OutputPorts, die mit dieser PCU assoziiert sind, tragen den durch den Benutzer vergebenen Bezeichner, ergänzt um einen zusätzlichen Suffix (`_BUS_IN` bzw. `_BUS_OUT`). Die Instanz der `GenericComComponent` leitet eingehende Nachrichten eines Ports direkt an die assoziierte PCU weiter. Umgekehrt werden auch die von einer PCU erzeugten Nachrichten[7] über den entsprechenden OutputPort der `GenericComComponent` an die jeweilige FSC weitergeleitet.

Abbildung 9.3 illustriert diese (Verbindungs-)Struktur sowie die innerhalb von FERAL erzeugten Instanzen bei der Simulation eines CAN-Busses *MCBus*. Über diesen Bus kommunizieren die drei Knoten *Node1*, *Node2* und *Node3* miteinander. Deren Verhalten wird mittels FSCs simuliert, die ebenfalls in der Abbildung dargestellt sind. Die Input- und OutputPorts der instanziierten `CANComComponent` sind direkt mit den Ports der jeweiligen PCUs (in diesem Fall Instanzen der Klasse `CANPCU`) verbunden, sodass die Nachrichten direkt weitergeleitet werden. Während die PCUs das Verhalten der Kommunikationscontroller simulieren, übernimmt der `CANMediumController` die eigentliche Simulation der Interaktion über *den gemeinsamen Bus*. Die in Abbildung 9.3 gezeigten Objekte und Klassen gehören zur CAN-CSC; die gleiche Struktur kommt jedoch auch bei der FR-CSC oder ACOM-CSC zum Einsatz. Der ns-3 verfügt intern über einen anderen Aufbau, die Konzepte sind aber weitgehend identisch hinsichtlich der verwendeten Schnittstelle (Input- und OutputPorts) [BCG+13].

In dem Beispiel aus Abbildung 9.3 interagieren die FSCs direkt mit der CSC, daher beinhalten die Verhaltensmodelle zwangsläufig protokollspezifisches Wissen, wie z. B. das Wissen um CAN-Rahmen und deren Identifier. Ebenso werden die konkreten Nachrichten- und Nachrichtendatentypen der CAN-CSC (vgl. Abbildung 9.4) direkt verwendet (s.u.). Ein Austausch der CAN-CSC gegen eine andere CSC

[4] Übertragungsverzögerungen lassen sich beispielsweise anhand der Konfiguration sowie dem Zustand des simulierten Mediums (Übertragungsrate, aktuelle Belegung etc.) berechnen.

[5] Im Folgenden verwenden wir, der einfacheren Begrifflichkeit wegen, in Bezug auf die CSCs den Begriff Bus, subsumieren hierunter aber auch andere Kommunikationstechnologien, wie drahtlose Netze oder Punkt-zu-Punkt-Verbindungen.

[6] Die Bezeichner sind Strings, die durch den Benutzer gewählt werden und sowohl die Zuordnung als auch die Erstellung der Links sowie die Zuordnung von Logausgaben erleichtern.

[7] Bei den hier angesprochenen Nachrichten handelt es sich um Objekte des Typs `Message` und nicht notwendigerweise um Nachrichten, die über den Bus übertragen werden.

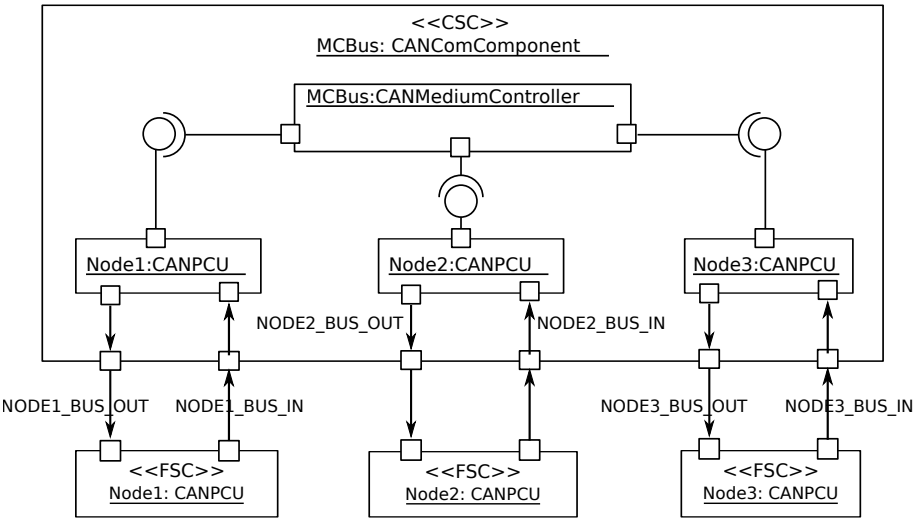


Abbildung 9.3: Verwendung der definierten Input- und OutputPorts zwischen CSCs und FSCs.

erfordert dementsprechend die Anpassung der Verhaltensmodelle der FSCs. In Kapitel 10 werden wir mit den *Bridges* ein Konzept vorstellen, welches speziell diese Problematik adressiert.

Die CSCs definieren eigene Nachrichten- und Nachrichtendatentypen, um Ereignisse mit anderen Simulationskomponenten auszutauschen. Die sogenannten `BusEvents` bilden elementare Ereignisse in Bezug auf die Kommunikation ab, und spezielle Nachrichtendatentypen dienen zur Speicherung und Übergabe der jeweiligen Rahmentypen. Abbildung 9.4 zeigt, wie sich die Nachrichten- und Nachrichtendatentypen in das von FERAL definierte Typsystem integrieren (vgl. Abbildung 8.2). Das UML-Klassendiagramm präsentiert sowohl die allgemeine Typhierarchie, welche von allen CSCs genutzt wird (und Bestandteil des `COM_Generic`-Bundles sind), als auch die konkrete Umsetzung der Nachrichten- und Nachrichtendatentypen durch die CAN-CSC. Die für die anderen CSCs bereitgestellten Typen basieren ebenfalls auf den in `com.generic.busevents` definierten `BusEvents` und sind in den jeweiligen Unterkapiteln beschrieben.

Alle von den CSCs verwendeten Nachrichten werden von dem gemeinsamen Nachrichtentyp `BusEvent` abgeleitet. Weiter verfeinert wird dieser Typ durch die Aufteilung in verschiedene Ereignisklassen: `MessageSendBusEvent` veranlassen eine CSC (bzw. deren PCU) dazu, den enthaltenen Rahmen zu senden. Empfangene Nachrichten werden von der jeweiligen CSC an die verbundenen Simulationskomponenten, als Objekte des Typs `MessageReceiveBusEvent`, übergeben. Ein solches `MessageReceiveBusEvent` enthält neben dem empfangenem Rahmen auch eine Referenz auf den ursprünglichen Auftrag für die Übertragung (das ursprüngliche `MessageSendBusEvent`). Hierdurch lassen sich innerhalb von FERAL sehr leicht die Datenflüsse rekonstruieren und Ende-zu-Ende-Verzögerungen ermitteln, da alle `BusEvents` einen Zeitstempel mit dem Erzeugungszeitpunkt des Ereignisses tragen. Um die Implementierung zu erleichtern, werden die beiden abstrakten Klassen `AbstractBusEvent` und `AbstractReceivingBusEvent` bereitgestellt. Jede CSC bietet Verfeinerungen dieser Klassen an, um die protokollspezifischen Eigenarten abzubilden. Das Beispiel in Abbildung 9.4 zeigt dies anhand der beiden Klassen `CANMessageSend` bzw. `CANMessageReceive`, der CAN-CSC. Für die Speicherung der eigentlichen Rahmen verwenden die CSCs jeweils eigene spezifische Nachrichtendatentypen. Im Beispiel von CAN wird ein CAN-Rahmen durch eine Instanz der Klasse `CANMessage` repräsentiert und deren Header als Instanz der Klasse `CANHeader`.

Neben dem Versand und Empfang von Nachrichten gibt es weitere Ereignisse, wie z. B. der Wechsel in einen anderen Fehler- oder Betriebszustand. Diese Ereignisse nutzen die Typen `IStatusEvent`

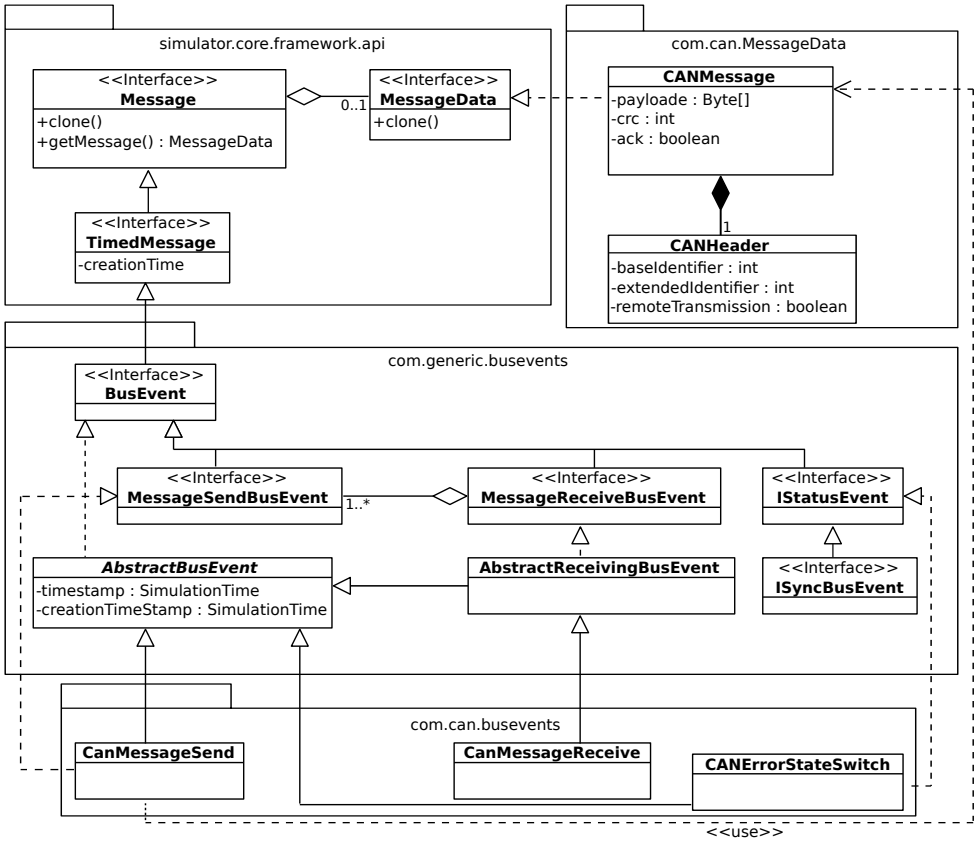


Abbildung 9.4: UML-Klassendiagramm der Nachrichtentypen der CSCs.

und `ISyncBusEvent`. Ersterer signalisiert einen allgemeinen Zustandswechsel, während letzterer ein spezielles Synchronisationsereignis kommuniziert (vgl. Kapitel 9.7.2). Bei CAN dient der Nachrichtentyp `CANErrorStateSwitch` für Benachrichtigungen bezüglich Veränderungen des Fehlerzustandes und implementiert daher die Schnittstelle[8] `IStatusEvent`.

Die beschriebene Definition der Nachrichtentypen, Nachrichtendatentypen sowie der Input- und OutputPorts schließen den Schritt 2 der Adaption von FERAL ab. Es fehlt noch der dritte und letzte Schritt: Die Instanziierung des Simulators mit einem Verhaltensmodell (vgl. Kapitel 8.1). Bei den CSCs besteht dieser Schritt darin, eine Instanz der jeweiligen Java-Kontrollkomponente mittels der konkreten Fabrik der CSC zu erzeugen und diese mit einer Konfiguration zu versehen. Diese beinhaltet, neben allgemeinen Parametern für den jeweiligen konkreten Bus, auch die Anzahl der angebundenen Knoten sowie die Konfigurationen der einzelnen PCUs. Das Verhaltensmodell ist bei den CSCs fest integriert und wird durch das Zusammenwirken der Protokollspezifikation (resp. deren Implementierung innerhalb des Medium-Controllers) sowie der Konfiguration (von Medium-Controller und PCUs) gebildet. Das UML-Klassendiagramm in Abbildung 9.5 zeigt die Klassen, welche die Konfiguration einer CSC und im speziellen der CAN-CSC zur Laufzeit repräsentieren. Die Abbildung zeigt für die CAN-CSC die konkreten Implementierungen der Typen für die medien- bzw. knotenspezifische Konfiguration in Form der Klassen `CANMediumConfiguration` sowie `CANNodeConfiguration`. Auch hier beschränken wir uns bei der Darstellung auf einige wenige repräsentative Attribute und somit auf lediglich einen kleinen Auszug der zulässigen Konfigurationsparameter.

[8] Da CAN ereignisgetriggert ist, existieren dort keine Synchronisationsereignisse. Dementsprechend stellt die CAN-CSC keine Implementierung der Schnittstelle `ISyncBusEvent` bereit.

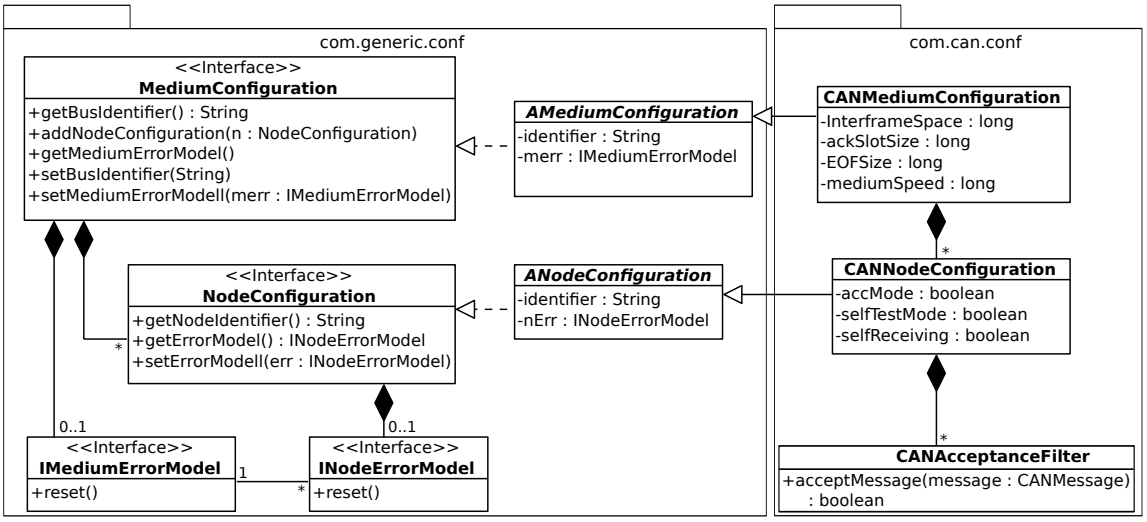


Abbildung 9.5: UML-Klassendiagramm der Konfigurationsschnittstelle.

Ein weiterer Bestandteil der Konfiguration ist ein Fehlermodell, welches es erlaubt, innerhalb der Simulation Übertragungsfehler zu berücksichtigen und deren Auswirkungen zu simulieren (`IMedium-ErrorModel` und `INodeErrorModel`). Dem Fehlermodell widmet sich Kapitel 9.5.

9.4 Realisierung der CSCs

Die speziell für FERAL entwickelten CSCs für die Simulation von CAN, FlexRay sowie des abstrakten Kommunikationsmodells verwenden ein zeitgetriggertes MOCC. Das bedeutet, die Simulationskomponente wird durch ihren – ebenfalls zeitgetriggerten – Direktor jeweils für die Dauer eines festen Zeitschritts ausgeführt. Innerhalb dieses Zeitschritts erfolgt keine Interaktion mit anderen Simulationskomponenten, d.h., es werden keinerlei Nachrichten ausgetauscht und sowohl der Zustand der Input- wie auch der OutputPorts bleibt währenddessen unverändert. Dies ermöglicht die parallele Ausführung aller zeitgetriggerter Simulationskomponenten des entsprechenden Direktors. Gleichzeitig wird durch die seltenere Synchronisation die Geschwindigkeit der Simulation verbessert (vgl. Kapitel 7.1.2). Da innerhalb des Simulationsschrittes keine Interaktionen mit anderen Simulationskomponenten erfolgt, kann die CSC den Zustand des Mediums für den gesamten Zeitschritt vollständig berechnen.

Die Genauigkeit bei der Simulation von Übertragungen wird hierbei durch die Wahl der Länge des Zeitschrittes (Aktivitätsperiode) nicht beeinträchtigt, da die Übertragungszeitpunkte, unabhängig von der Dauer des Zeitschritts, stets exakt berechnet werden. Allerdings – und hier liegt ein Nachteil bei der zeitgetriggerten Semantik – bestimmt die Länge der Aktivitätsperiode darüber, wann und wie oft neue Sendeaufträge von anderen Simulationskomponenten entgegengenommen werden können. Dies gilt natürlich auch in umgekehrter Richtung: Nur am Ende eines Zeitschrittes kann die CSC andere Knoten über Ereignisse, wie den Empfang einer Nachricht oder einen Zustandswechsel, informieren. Die simulierten Ereignisse selbst tragen jedoch einen exakten Zeitstempel, der darüber Auskunft gibt, wann dieses aufgetreten ist.

Daher ist es wichtig, die Dauer eines Simulationsschrittes der CSCs, entsprechend den Zielen der Simulation, sinnvoll zu wählen: Hierbei gilt es, einen geeigneten Tradeoff zwischen einer hohen Genauigkeit (sehr kurzer Aktivitätsperiode) einerseits und einer schnellen Simulationsgeschwindigkeit (lange Aktivitätsperiode) andererseits zu finden. Einen Hinweis für den Entwickler bietet hierbei die Berücksichtigung

des Abstraktionsgrades der anderen Simulationskomponenten[9] sowie die jeweilige Entwicklungsphase, in der die Simulation durchgeführt wird. Auch der Schwerpunkt der Fragestellung, unter der die Simulation durchgeführt wird, ist für die Entscheidung relevant. Liegt dieser auf der Evaluation einer Kommunikationstechnologie, so wird man eine kürzere Periodendauer wählen als bei einer rein funktionalen Evaluation von FSCs.

Da ein zeitgetriggertes MOCC verwendet wird, erbt die abstrakte Basisklasse `GenericComComponent` ihr Verhalten von der allgemeinen Basisklasse für zeitgetriggerte Simulationskomponenten `TimeTriggeredSimulationComponent`. Die konkreten Java-Kontrollkomponenten der einzelnen CSCs für CAN (`CANComComponent`), FlexRay (`FRComComponent`) und dem abstrakten Kommunikationsmodell (`ACOMComComponent`) sind ihrerseits wiederum von `GenericComComponent` abgeleitet. Das UML-Klassendiagramm 9.6 zeigt diesen Zusammenhang exemplarisch für die CAN-CSC.

Die InputPorts der CSCs weichen von dem klassischen MOCC bei zeitgetriggerten Simulationskomponenten ab, indem sie nicht nur eine einzige Nachricht speichern, sondern eine Warteschlange implementieren. Diese Anpassung ist erforderlich, damit beim Zusammenspiel mehrerer Direktoren und Simulationskomponenten mit unterschiedlichen Aktivitätsperioden sichergestellt ist, dass keine Nachrichten überschrieben werden, bevor diese an die zuständige PCU weitergeleitet werden. Jede PCU entscheidet dann auf Basis ihres internen Zustandsmodells selbstverantwortlich, ob ggf. Sendeaufträge verworfen werden, weil z. B. die maximale simulierte Pufferlänge innerhalb des simulierten Controllers überschritten wurde.

Die `GenericComComponent` bzw. die Instanz der konkreten Java-Kontrollkomponente der CSC (z. B. `CANComComponent` bei der CAN-CSC) verfügt über eine Konfiguration für das zu simulierende Medium (z. B. `CANMediumConfiguration`, vgl. Abbildung 9.5), inklusive entsprechenden Knotenkonfigurationen. Die Java-Kontrollkomponente steuert den Ablauf der Simulation des Busses auf Basis der Methoden der `MCIface`-Schnittstelle, entsprechend den Vorgaben des eigenen Direktors. Die CSC-spezifischen Implementierungen dieser Schnittstelle – die Medium-Controller – setzen das Verhalten gemäß der jeweiligen Protokollspezifikation um (z. B. der `CANMediumController` für das CAN-Protokoll [ISO03b, Rob91].).

Wir geben im Folgenden einen kurzen Überblick über die wichtigsten Methoden der `MCIface`- und `PCUIface`-Schnittstellen sowie deren Interaktion, beginnend mit dem `MCIface`. Die beiden Methoden `getPCU` und `getPCUbyID` dienen der Verwaltung und erlauben die Abfrage der instanziierten PCUs. Diese werden bei der Instanziierung der Java-Kontrollkomponente benötigt, um die erforderlichen Ports für die Interaktion mit den anderen Simulationskomponenten anzulegen (vgl. Abbildung 9.3) und die empfangenen `BusEvents` an die PCUs weiterzuleiten. Die `BusEvents` werden den PCUs durch die Methode `enqueueNewEvent` übergeben.

Die Übergabe der `BusEvents` an die PCUs erfolgt jeweils als erster Schritt, nachdem die Ausführungskontrolle an die jeweilige Instanz der Java-Kontrollkomponente übergeben wurde (Aufruf der `step`-Methode durch den Direktor). Danach führt die Kontrollkomponente die `execute`-Methode des Medium-Controllers aus, der als Parameter die Dauer des Simulationsschrittes sowie die aktuelle Simulationszeit übergeben wird. Der Rückgabewert ist ein Zeitstempel, der angibt, zu welchem Zeitpunkt der Medium-Controller erneut ausgeführt werden muss. Eine frühere Ausführung erfolgt nur, falls zwischenzeitlich neue `BusEvents` (von anderen Simulationskomponenten) eingehen.

In der `execute`-Methode sammelt der Medium-Controller zunächst bei jeder PCU das früheste gespeicherte `BusEvent` – welches bereits eingetreten ist – ein (`getNextMediumEvent`). Basierend auf den eingesammelten `MessageSendeEvents`, entscheidet der Medium-Controller unter Berücksichtigung des aktuellen Medienzustands und gemäß der Protokollspezifikation, wie weiter zu verfahren ist. Ist beispielsweise bei CAN das Medium gerade belegt, wird die Übertragung zurückgestellt; bei einem freien

[9] Handelt es sich um einfache Modelle, die lediglich das Verhalten der finalen Komponenten vereinfacht modellieren, so sind im Allgemeinen auch die Anforderungen an die Genauigkeit der Kommunikationssimulation geringer, und die Aktivitätsperiode kann größer gewählt werden.

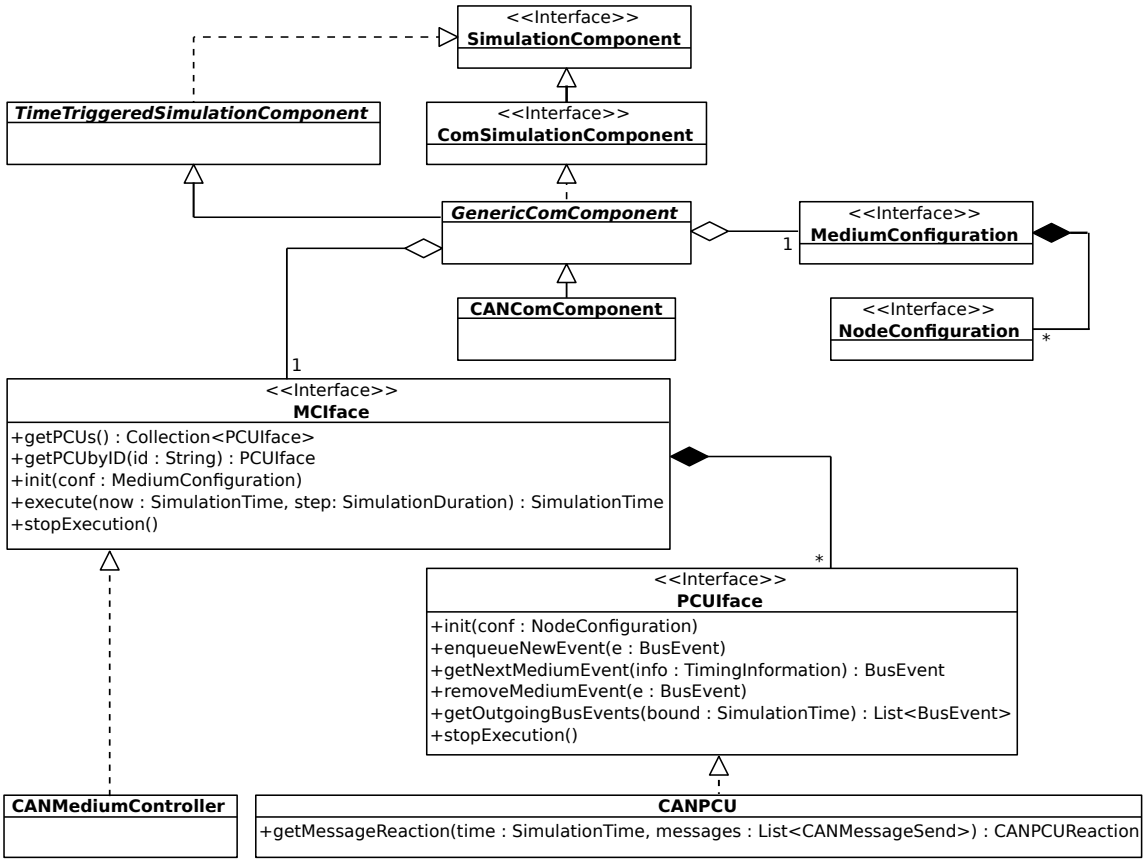


Abbildung 9.6: UML-Klassendiagramm des internen Aufbaus der CSCs.

Medium und mehreren gleichzeitigen `MessageSendeEvents` wird der Konflikt über den Arbitrierungsmechanismus von CAN gelöst. Wurde das Ereignis durch den Medium-Controller oder anderweitig durch die PCU verarbeitet, wird dieses anschließend aus der Warteschlange entfernt (`removeMediumEvent`).

Wird die Übertragung eines Rahmens in der Simulation erfolgreich abgeschlossen, erzeugt der Medium-Controller ein Empfangsereignis (`MessageReceiveEvent`) und liefert dieses an die PCUs aus. Die PCUs speichern die erzeugten Empfangsereignisse zusammen mit internen Ereignissen (wie z. B. einem Zustandswechsel in einen anderen Fehlerzustand) solange, bis diese an FERAL übergeben werden können. Ob und wie viele *empfangene Rahmen* eine PCU vorhalten kann, hängt von deren instanziiertem Verhaltensmodell sowie der gewählten Konfiguration ab. Imitiert die PCU beispielsweise einen konkreten CC, so richtet sich deren Speichermodell und -vermögen natürlich nach dessen spezifischen Eigenarten.

Am Ende eines Simulationsschrittes (als Bestandteil der `postfire`-Methode der Java-Kontrollkomponente) wird geprüft, ob die PCUs Ereignisse gespeichert haben, die an FERAL weitergeleitet werden müssen. Hierzu nutzt die Java-Kontrollkomponente die Methode `getOutputBusEvents`, um die Ereignisse in die den PCUs zugeordneten OutputPorts der Java-Kontrollkomponente zu transferieren[10] und anschließend über die Links weiterzuleiten.

Die CSCs für CAN, FlexRay und das abstrakte Kommunikatonsmodell können zusätzlich auch Übertragungsfehler in der Simulation berücksichtigen. Hierfür wird ein Fehlermodell verwendet, welches vor dem Erzeugen des `MessageReceiveEvent` entscheidet, ob bei diesem Rahmen ein Übertragungsfehler simuliert werden soll. Zusätzlich fragt der Medium-Controller die PCUs, ob diese auf die Übertragung

[10] Die Ereignisse sind entsprechend dem Zeitpunkt ihrer Erzeugung geordnet.

in besonderer Weise reagieren wollen, z. B. in Form einer (ggf. fehlerhaften) Fehlersignalisierung. Dies ermöglicht die Simulation von falschem Verhalten oder Fehlern innerhalb einer einzelnen PCU. Die Reaktion auf einen Übertragungsfehler ist abhängig von dem Protokoll und wird daher bei den CSCs unterschiedlich umgesetzt. Die CAN-CSC reagiert auf Fehler mit einer Fehlersignalisierung (vgl. Kapitel 3.1), in der Simulation wird hierfür die Methode `getMessageReaction` verwendet. Diese ist nicht Bestandteil der `PCUIface`-Schnittstelle, da sie nicht von allen CSCs benötigt wird, z. B. verwirft FlexRay fehlerhafter Rahmen ohne Rückmeldung über den Bus.

9.5 Simulation von Übertragungsfehlern

Gerade bei sicherheitskritischen Anwendungen ist es wichtig, dass ein System, selbst beim Auftreten von Fehlern noch ein sicheres Verhalten aufweist[11]. Auch bei diesbezüglichen Untersuchungen kann FERAL den Entwickler unterstützen. So lassen sich Ausfälle kompletter Funktionalitäten oder Knoten dadurch nachbilden, dass die entsprechenden FSCs ab einem definierten Zeitpunkt nicht mehr durch ihren Direktor ausgeführt werden. Auch fehlerhafte Sensorwerte oder Aktuatoransteuerungen lassen sich relativ einfach in FERAL umsetzen, beispielsweise durch eine Anpassung der Verhaltensmodelle von Sensoren bzw. Aktuatoren. Die Simulation von Fehlern in der Hardwareplattform oder bei der Kommunikation sowie deren Auswirkungen ist hingegen bereits anspruchsvoller.

Um Auswirkungen von Hardwarefehlern (z. B. Speicherfehlern) zu simulieren, werden entsprechende Simulatoren für die Hardwareplattform benötigt, die in der Lage sind, die Hardware und deren Abläufe taktgenau zu simulieren. Simulationen dieser Art lassen sich erst sehr spät in der Entwicklung einsetzen, wenn diese Plattform bereits – z. B. in Form eines bestimmten Steuergerätes – ausgewählt und die Funktionalitäten vollständig entwickelt wurden. Um solche Simulationen durchführen zu können, arbeitet die AG Wehn aktuell an der Integration des Plattform Simulators Synopsis [Rey13] in FERAL.

Wir widmen uns stattdessen der zuletzt genannten Fehlerquelle: Fehler, die bei der Kommunikation innerhalb eines verteilten Systems auftreten bzw. hieraus resultieren. Fehler bei der Übertragung von Rahmen treten in der Praxis indeterministisch auf und können sowohl zum Verlust des Rahmens als auch zu dessen Verfälschung führen. Ob Übertragungsfehler erkannt und behandelt werden, hängt von dem jeweiligen Protokoll, dessen Bitcodierung, Rahmenformat sowie den Fehlerbehandlungsstrategien ab. CAN beispielsweise verwendet eine CRC, um Fehler zu erkennen, und überträgt fehlerhafte Rahmen standardmäßig solange erneut, bis die maximale Anzahl von Versuchen erreicht wurde oder der Knoten in einen Fehlerzustand wechselt (Bus-off). In diesen Fällen wird der Rahmen verworfen und geht verloren.

Wir teilen die Fehler, welche bei der Kommunikation auftreten können, in zwei Fehlerkategorien ein: Fehler, die nur einen einzelnen Knoten betreffen – weil dieser entweder defekt (Hardwaredefekt des Kommunikationscontrollers) oder die Verbindung zum Medium unterbrochen ist – und Fehler, welche sich auf das Übertragungsmedium selbst beziehen und von deren Auswirkungen alle Knoten betroffen sind. Für diese zweite Kategorie von Fehlern können beispielsweise elektromagnetische Einstrahlungen in das Kabel verantwortlich sein, die die Übertragung verfälschen. Auch unser Fehlermodell spiegelt diese zwei Kategorien von Fehlern wider und unterscheidet daher zwischen medium- und knotenspezifischen Fehlern. Diese können in unserem Fehlermodell unabhängig voneinander definiert werden.

Das UML-Diagramm 9.7 zeigt den Aufbau unseres Fehlermodells. Verschiedene Arten von medium- bzw. knotenspezifischen Fehlern werden durch unterschiedliche Implementierungen der Schnittstellen `IMediumErrorModell` bzw. `INodeErrorModel` repräsentiert. Eine über die reine Struktur hinausgehende Definition (in Form einer einheitlichen Schnittstelle mit Methodendefinitionen) hat sich aufgrund der starken Unterschiede bei den Protokollen und den Medium-Controllern sowie der protokollspezifischen Reaktionen auf Fehler hier leider als unpraktikabel erwiesen. Um ein bestimmtes Fehlermodell

[11] Die Festlegung, welches Verhalten in dem jeweiligen Kontext akzeptabel ist, hängt von der Anwendung, der Auftrittswahrscheinlichkeit der Fehler sowie den möglichen Folgen bei einem Systemversagen ab.

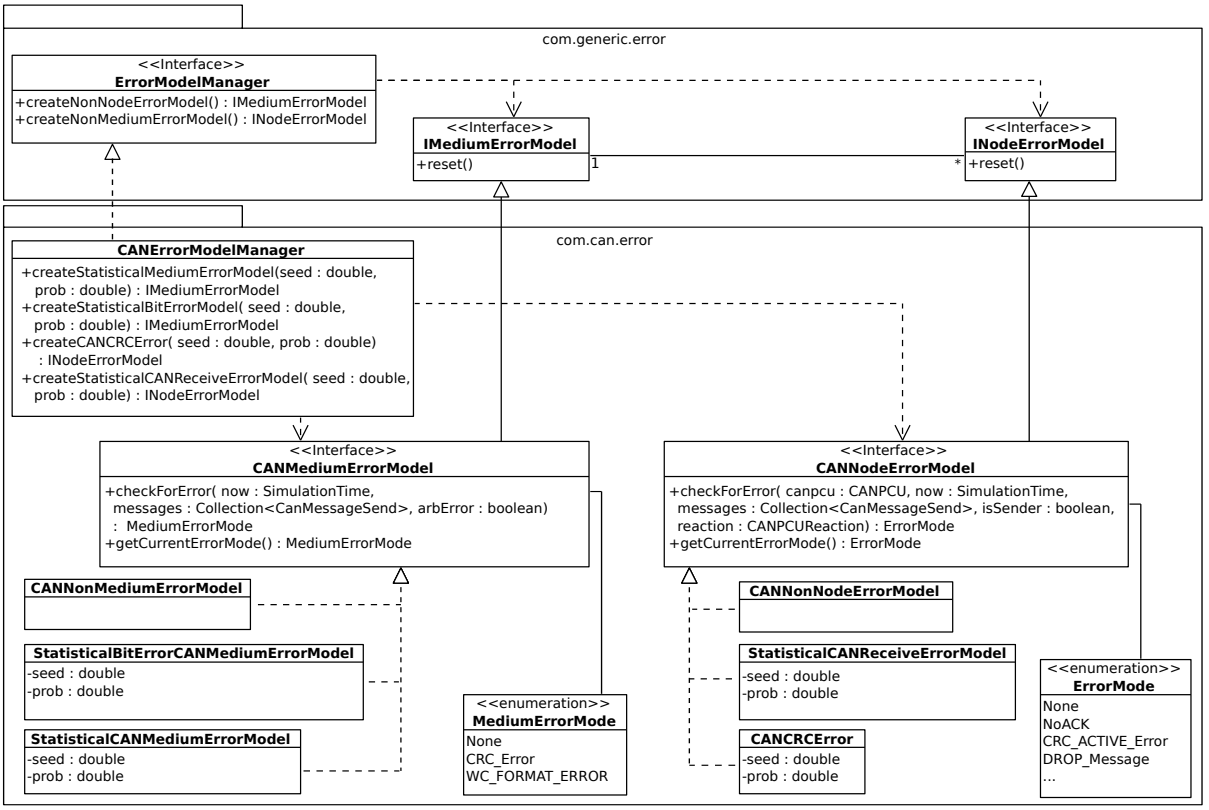


Abbildung 9.7: UML-Klassendiagramm des Fehlermodells.

zu verwenden, wird ein Objekt der entsprechenden Klasse instanziiert und zusammen mit der Medium- bzw. Knotenkonfiguration übergeben. Wie schon in den vorherigen Beispielen zeigen wir, neben der allgemeinen Klassenstruktur, auch wieder direkt die Umsetzung des Fehlermodells für das CAN-CSC.

Der Typ `ErrorModelManager` definiert die Schnittstelle der abstrakten Fabrik für die Erzeugung der konkreten medium- bzw. knotenspezifischen Fehlermodelle. Die angebotenen beiden Fabrikmethoden dienen der Instanziierung von Fehlermodellen, die während der Ausführung keine Fehler erzeugen. Diese kommen bei rein funktionalen Evaluationen des Verhaltens zum Einsatz und können ohne weitere Änderungen des Simulationssystems durch andere Fehlermodelle ersetzt werden, welche spezifische Fehler modellieren. Für die Erzeugung der CSC-spezifischen Fehlermodelle stellen die konkreten Fabriken der CSCs entsprechende Fabrikmethoden bereit. Ein Beispiel hierfür liefert der `CANErrorModelManager`. Innerhalb der CSCs sind die Schnittstellen der Fehlermodelle besser ausgestaltet, wie die Typen `CANMediumErrorModel` bzw. `CANNodeErrorModel` belegen.

Über die Methode `checkForError` der Klasse `CANMediumErrorModel` prüft der `CANMediumController`, ob die aktuelle Übertragung durch einen simulierten Fehler auf dem Medium gestört wird. Der Rückgabewert (`MediumErrorMode`) gibt Aufschluss über die Art des Fehlers. Neben der aktuellen Zeit sowie den anstehenden Übertragungen (identifiziert durch deren Sendeaufträge[12]) gibt der letzte Parameter an, ob bei der Arbitrierung ein Fehler aufgetreten ist, z. B. weil zwei Nachrichten mit gleichem CAN-Identifier, aber unterschiedlicher Payload gleichzeitig übertragen werden sollen.

Analog zu den mediumspezifischen Fehlern nutzen die PCUs ihr knotenspezifisches Fehlermodell, um mit der Methode `checkForError` zu überprüfen, ob die konkrete Übertragung durch einen Fehler

[12] An dieser Stelle wird eine Liste von Sendeaufträgen übergeben, da z. B. identische Datenanforderungsrahmen von mehreren Knoten gleichzeitig ohne Fehler gesendet werden können.

aufgrund einer simulierten Fehlfunktion der PCU[13] gestört werden soll. Die Reaktion der PCU auf die Übertragung wird durch ein Objekt des Typs `CANPCUReaction` zurückgeliefert. So kann eine PCU bei CAN beispielsweise mit dem Aufschalten eines aktiven oder passiven Fehlerflags auf einen empfangenen Rahmen reagieren. Ebenso lassen sich fehlerhafte Reaktionen abbilden (wie z. B. ein Fehlerflag zu senden, obwohl der Rahmen korrekt ist oder sich der Controller im Zustand Bus-off befindet). Die dargestellten Literale führen zu einem Verzicht auf die Sendung einer Quittung (`NoACK`) oder erzwingen die Übertragung eines aktiven Fehlerflags (`CRC_ACTIVE_Error` – unabhängig von dem empfangenen Rahmen und dem eigenen Zustand des Knotens), während bei `DROP_Message` die PCU den Rahmen verwirft.

Sowohl auf Ebene des Mediums als auch auf Knotenebene stellt die CAN-CSC statistische Fehlermodelle bereit. Diese basieren auf uniformen Wahrscheinlichkeitsverteilungen, die mit einer bestimmten (konfigurierbaren) Wahrscheinlichkeit einen Übertragungsfehler erzeugen. Die beiden dargestellten statistischen Fehlermodelle auf Mediumebene erlauben die Angabe einer Wahrscheinlichkeit für das Auftreten eines Bitfehlers resp. für die Verfälschung eines Rahmens. Die statistischen Fehlermodelle auf Knotenebene ermöglichen es, die Wahrscheinlichkeit für einen detektierten Übertragungsfehler vorzugeben (unabhängig davon, ob der Rahmen auf dem Medium verfälscht wurde oder nicht). Hierbei erzeugt das `CANCRCError`-Modell stets ein aktives Fehlerflag, unabhängig vom Zustand des Knotens, während sich das andere Modell korrekt verhält und den Fehlerzähler der PCU berücksichtigt. So lassen sich Fehler simulieren, die entweder nur für eine PCU sichtbar sind oder durch einen Fehler in der PCU selbst verursacht werden.

Die Aufgabe des `CANMediumController` besteht darin, die unterschiedlichen Reaktionen der PCUs, unter Berücksichtigung des mediumspezifischen Fehlermodells, korrekt zu komponieren. Über dieses zweistufige Fehlermodell lassen sich komplexe Fehlerbilder zusammenstellen und zudem, durch die Implementierung der beiden vorgestellten Schnittstellen, leicht weitere Fehlerarten integrieren. Erweiterungen könnten, z. B. Übertragungen von Rahmen mit bestimmten CAN-Identifier stören, um ein bestimmtes Fehlerszenario bzw. Verhalten gezielt zu evaluieren.

Die Fehlersimulation bei unserem CAN-CSC beschränkt sich momentan auf eine Erkennung von Verfälschungen eines Rahmens bei der CRC-Prüfung. Die Fehlersignalisierung selbst erfolgt entsprechend des CAN-Standards und dem Bestätigungsfeld. Alternativ kann ein Fehler zum spätestmöglichen Zeitpunkt eingeschleust werden (innerhalb des EOF-Delimiters, vgl. Definition 3.10, Kapitel 3.2.2). Beide Ansätze dienen dazu sicherzustellen, dass die Zeitspanne bis der Bus wieder für eine normale Übertragung genutzt werden kann maximal ist und auf diese Weise stets der Worst-Case bei der Simulation berücksichtigt wird.

9.6 Protokollierung von Simulationsabläufen

Ebenso wichtig wie die Simulation selbst ist die Möglichkeit, diese nachvollziehen und (automatisch) auswerten zu können. Hierfür implementieren alle CSCs – mit Ausnahme von ns-3 – eine gemeinsame Debug-Schnittstelle, über die alle relevanten ausgeführten Aktionen gespeichert und, falls gewünscht, direkt ausgegeben werden können. Deshalb wurde in Zusammenarbeit mit dem IESE ein OSGi-Dienst entworfen, der die Debug-Nachrichten aller Simulationskomponenten zentral sammelt und diese für andere Softwarekomponenten (wie z. B. Visualisierungen oder Speicherdiensten) zur Verfügung stellt.

Zu diesem Zweck wurde der Typ `DebugMessage` eingeführt, welcher von allen Simulationskomponenten für die Meldung von Aktionen genutzt werden kann. Eine solche Debug-Nachricht enthält neben einer Referenz auf die Simulationskomponente samt Verhaltensmodell (Konfiguration bei den CSCs), auch eine Tabelle aus Schlüssel-Wert-Paaren, welche die Aktion charakterisieren. Der OSGi-Dienst

[13] Jede PCU verfügt über eine eigene Instanz eines knotenspezifischen Fehlermodells. Welches Fehlermodell verwendet wird, kann für jede PCU individuell festgelegt werden.

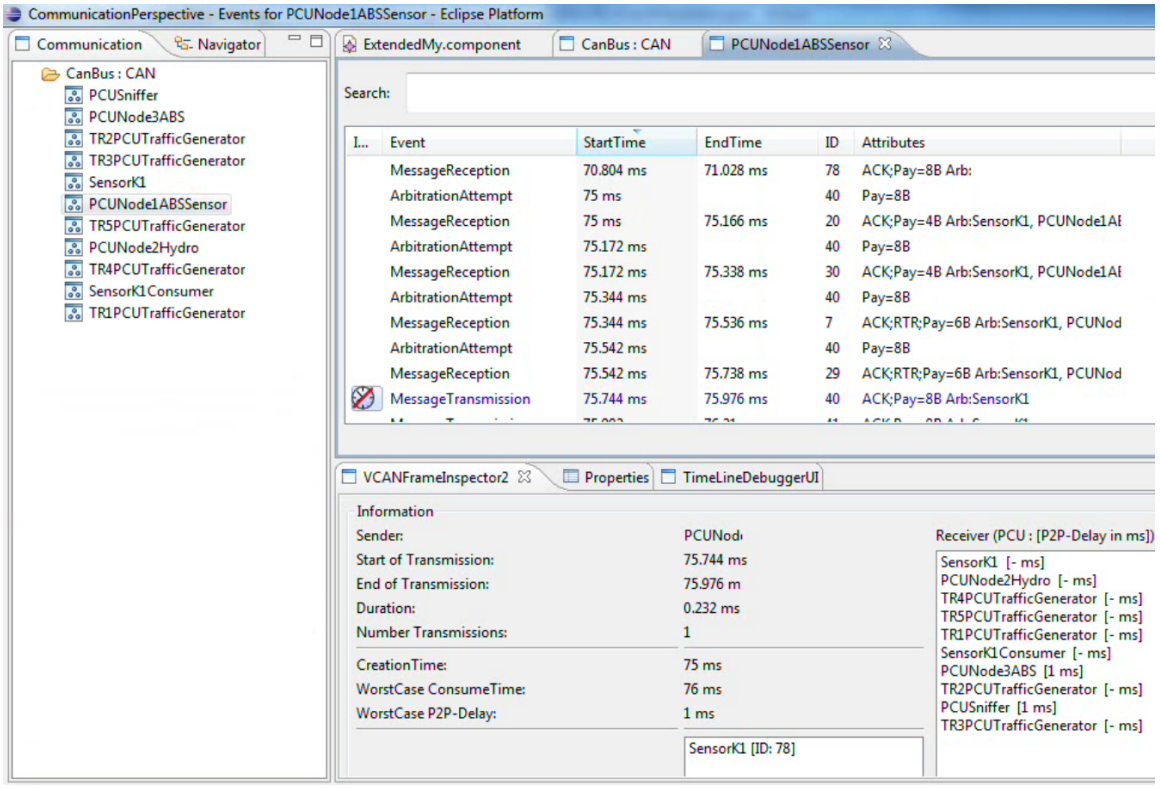


Abbildung 9.8: Visualisierung der Kommunikation in einem CAN-Bus.

bietet auch anderen Softwarekomponenten oder Visualisierungen die Möglichkeit, sich als Empfänger für die gesammelten Debug-Nachrichten zu registrieren (Observer-Pattern). Dies wurde genutzt, um verschiedene Visualisierungen der Aktionen zu realisieren sowie die erzeugten Debug-Nachrichten in einer SQL-Datenbank (für eine spätere Auswertung) zu speichern. Des Weiteren wurden mit Hilfe der in den Debug-Nachrichten übertragenen Informationen automatische Constraint-Checks implementiert. Diese prüfen zur Laufzeit, ob Constraints eingehalten werden. Hierzu gehört beispielsweise, ob vorgegebene bestimmte Ende-zu-Ende-Verzögerungen bei der Kommunikation eingehalten werden oder Werte (Sensorwerte etc.) innerhalb eines spezifizierten Wertbereichs liegen.

Über die Debug-Schnittstelle ist auch eine Visualisierung direkt zur Laufzeit möglich. Die Abbildung 9.8 zeigt eine solche Visualisierung für die Kommunikation eines ABS-Systems über einen CAN-Bus. Auf der linken Seite sind die Knoten des CAN-Busses dargestellt, während im rechten Teil Details zur Kommunikation angezeigt werden. Die rechte obere Hälfte zeigt eine Übersicht der Nachrichtenereignisse (`BusEvents`), während der untere Teil Detailinformationen zu einem ausgewählten Ereignis anzeigt. Bei einem übertragenen Rahmen sind dies Start- und Endzeitpunkt, die Dauer der Übertragung sowie weitere Details. Die durchgestrichene Uhr weist den Entwickler darauf hin, dass die für diese Nachricht konfigurierte, maximal zulässige Übertragungsverzögerung verletzt wurde und stellt damit direkt die Ergebnisse des Constraint-Checks dar. Die Funktionsweise von FERAL, insbesondere im Kontext der interaktiven Evaluation, haben wir im Rahmen des Audits des Innovationszentrums 2012 präsentiert. Ein entsprechender Screencast ist unter http://agvs.cs.uni-kl.de/FERAL/ verfügbar.

9.7 Übersicht der bereitgestellten CSCs

Nachdem wir die allgemeine Architektur sowie die Schnittstelle der implementierten CSCs betrachtet haben, legen wir nun den Schwerpunkt auf die im Rahmen der Arbeit implementierten CSCs für CAN (CAN-CSC), FlexRay (FR-CSC), das abstrakte Kommunikationsmodell (ACOM-CSC) sowie der Integration bzw. dem Wrapper für den ns-3 (ns-3-CSC). Da der CAN-CSC bereits als Beispiel für die Illustration der Architektur herangezogen wurde und die anderen CSCs die gleiche Architektur aufweisen, beschränken wir uns im Folgenden auf eine Beschreibung des Leistungsumfangs der einzelnen CSCs sowie ausgewählter Aspekte der Umsetzung.

9.7.1 Simulator für das Controller Area Network (CAN)

Das Controller Area Network (CAN) ist ein in der Automobilindustrie und Automatisierung weit verbreiteter Feldbus, der auf einem prioritätsbasierten Medienzugriff beruht (vgl. Kapitel 3). Da CAN eine etablierte Basistechnologie ist, erschließt die Integration in FERAL ein sehr breites Markt- und Anwendungssegment. Insbesondere die Kombination mit Matlab Simulink macht FERAL zu einem potentiell interessanten Werkzeug für Entwicklungen im Umfeld der Automobilindustrie.

Die entwickelte CAN-CSC setzt die CAN Specification 2.0 [CiA14a, Rob91, ISO03b] um und unterstützt sowohl das einfache (11-Bit Identifier) als auch das erweiterte Rahmenformat (29-Bit Identifier). Die Simulation erfolgt auf Rahmenebene, d.h., zu jedem Zeitpunkt, zu dem der simulierte Bus nicht belegt ist, werden die aktuell anliegenden Sendeaufträge der einzelnen PCUs gesammelt und auf Basis der CAN-Identifier der- oder diejenigen mit der höchsten Priorität ermittelt. Eine physikalisch korrekte Simulation der Abläufe des Arbitrierungsprozesses oder der Übertragung auf elektrischer Ebene ist nicht notwendig, da ein einfacher Wertevergleich der Identifier aufgrund des vorhandenen globalen Wissens für die Realisierung ausreicht. Unterstützt werden sowohl Datenrahmen als auch Datenanforderungsrahmen, insbesondere bei letzteren wird auch die gleichzeitige Übertragung identischer Rahmen durch mehrere verschiedene Knoten korrekt simuliert. Zusätzliche Mechanismen helfen bei der Fehlersuche; so meldet der Simulator einen Konfigurationsfehler, falls verschiedene Knoten gleichzeitig Rahmen mit identischem CAN-Identifier, aber unterschiedlicher Payload, zu übertragen versuchen.

Die Übertragungsdauer eines CAN-Rahmens wird entsprechend dem gewählten Rahmenformat (vgl. Abbildung 3.1), der Anzahl übertragener Bits auf dem Medium (inkl. Berücksichtigung des Bit-Stuffing) sowie der Übertragungsrate berechnet. Der berechnete Endzeitpunkt der Übertragung wird als Empfangszeitpunkt für die aktiven PCUs verwendet, die den empfangenen Rahmen dann im Anschluss weiterverarbeiten. Um die Korrektheit der Implementierung zu prüfen, wurde auf das im ersten Teil dieser Arbeit bereits verwendete STM32F4-Discovery-Board zurückgegriffen, um die Simulationsergebnisse (ermittelte Übertragungszeiten, den Berechnungen zugrunde liegende Bitsequenzen) mit gemessenen Werten[14] bei real übertragenen Rahmen (durch den CAN-Controller des STM32F4) zu vergleichen. Ebenso wurden mehrere CAN-Rahmen sowohl in der Realität als auch in der Simulation in direkter Abfolge übertragen, um die korrekte Umsetzung des Inter-Frame-Spacing und der End-of-Frame Signalisierung zu verifizieren. Für rein funktionale Tests der eigentlichen Implementierung einzelner Komponenten innerhalb der CAN-CSC kamen JUnit-Tests zum Einsatz.

In Bezug auf die Simulation von Übertragungsfehlern unterstützt CAN-CSC das bereits vorgestellte zweistufige Fehlermodell. Da die Übertragung auf Rahmenebene simuliert wird, erfolgt die Fehlerbehandlung stets so, als ob ein simulierter Fehler erst bei der Prüfung der CRC entdeckt würde. Hierbei handelt es sich um eine konzeptionelle Einschränkung, deren Behebung eine Simulation der Übertragung auf Bitebene erfordert. Eine entsprechende Rückfallebene wurde bereits vorgesehen, jedoch noch nicht implementiert. Die aktuelle Simulation deckt den Worst-Case ab, d.h., es wird die maximale Dauer simuliert,

[14] Hierbei wurden Messungen und Vergleiche für ausgewählte Rahmentypen mit Hilfe eines digitalen Speicheroszilloskops durchgeführt.

bis der Bus sich regeneriert und wieder für die Übertragung regulärer Rahmen zur Verfügung steht. Die Fehlersignalisierung beginnt mit dem Start des EOF-Delimiters und erfolgt gemäß den Vorgaben des CAN-Standards. Hierbei wird sowohl die Überlappung von Error-Frames als auch die Unterscheidung zwischen primärem und sekundärem Fehlerflag unterstützt. Diese wird beispielsweise für die korrekte Simulation von Knoten benötigt, die fälschlicherweise Fehler signalisieren. Die PCUs verfügen jeweils über eigene Fehlerzähler und wechseln den Fehlerzustand gemäß den Vorgaben des Standards. Der Fehlerzustand selbst beeinflusst die Art des gesendeten Fehlerflags und das Kommunikationsverhalten des Knotens[15]. Neben der Fehlersignalisierung wird auch der Acknowlegdement-Mechanismus des CAN-Protokolls nachgebildet.

Die implementierten PCUs orientieren sich an der Funktionalität realer CAN-Controller und bieten die Möglichkeit, über *Acceptance Filtering* eingehende Nachrichten anhand ihrer CAN-Identifier zu selektieren und zu filtern. Die Größe der Nachrichtenpuffer sowohl für die empfangenen als auch die noch zu sendenden Nachrichten lässt sich sich beliebig konfigurieren, um auf diese Weise relevante Eigenschaften der später eingesetzten realen Hardware abbilden zu können.

9.7.2 Simulator für das FlexRay-Protokoll

FlexRay ist ein deterministisches zeitgetriggertes Kommunikationsprotokoll, das speziell für die Anforderungen der Automobilindustrie und für die Realisierung sicherheitskritischer Anwendungen entwickelt wurde. FlexRay unterteilt die Zeit in Kommunikationszyklen, die ihrerseits wieder in ein statisches und ein dynamisches Segment (optional) konfigurierbarer Länge unterteilt werden (vgl. Kapitel 6.1). Das obligatorische statische Segment ist in eine feste Anzahl statischer Slots unterteilt, von denen jeder maximal einem Knoten (exklusiv) zugeordnet wird. Innerhalb des optionalen dynamischen Segments erfolgt die Zuteilung der verfügbaren Bandbreite in Abhängigkeit von den Nachrichtenprioritäten (FTDMA). Um die hohen Anforderungen hinsichtlich der Übertragungssicherheit zu gewährleisten, verwendet FlexRay zwei voneinander getrennte physikalische Kanäle mit einer Übertragungsrate von 10 MBit/s, die auch zur redundanten Übertragung genutzt werden können (statisches Segment).

FlexRay unterstützt eine Vielzahl unterschiedlichster Topologien (z. B. unterschiedliche Topologien der Kanäle, Anbindung von Knoten an lediglich einen einzigen Kanal etc.). Unser FlexRay Simulator (FR-CSC) verzichtet auf die Abbildung der Topologie und setzt implizit voraus, dass alle Knoten stets über beide Kanäle miteinander verbunden sind. Möchte der Entwickler nur über einen Kanal kommunizieren, kann er dies realisieren, indem er die Nachrichten jeweils nur auf dem jeweiligen Kanal sendet oder empfängt. Hierfür müssen lediglich die Ein- bzw. Ausgangspuffer der betroffenen Knoten entsprechend konfiguriert werden.

9.7.2.1 Aufbau der FR-CSC und Integration in FERAL

Auch das FR-CSC ist gemäß dem Abstract Factory-Pattern des `COM_Generic`-Bundle aufgebaut; die Schnittstelle zwischen den FSCs und der FR-CSC wird, wie bei der CAN-CSC, über Input- und OutputPorts mit einem festen Namensschema realisiert (vgl. Kapitel 9.3). Die Java-Kontrollkomponente des FR-CSC (`FRComComponent`), über die FERAL mit dem FlexRay Medium-Controller interagiert, erbt von der Klasse `GenericComComponent` und verwendet eine zeitgetriggerte Ausführungssemantik (vgl. die UML-Klassendiagramme in Abbildung 9.2 und 9.6). Die Verwendung der zeitgetriggerten Ausführungssemantik, zusammen mit einer Simulation auf Rahmenebene (wie bei CAN), erlaubt eine sehr schnelle und effiziente Simulation mit einer hinreichend hohen Genauigkeit[16].

[15] Der Zustand `bus-off` erlaubt keine Teilnahme an der Kommunikation, während im Zustand `error-passive` ein Knoten nach einer erfolgreichen Übertragung eine bestimmte Zeitspanne abwarten muss, bis er erneut senden darf.

[16] Alle relevanten Aspekte der Vernetzung eingebetteter Systeme lassen sich evaluieren. Hierzu gehören Übertragungsverzögerungen, Zugriffskonflikte (z. B. bei Fehlkonfigurationen) sowie die Simulation von Übertragungsfehlern und Nachrichtenverlusten.

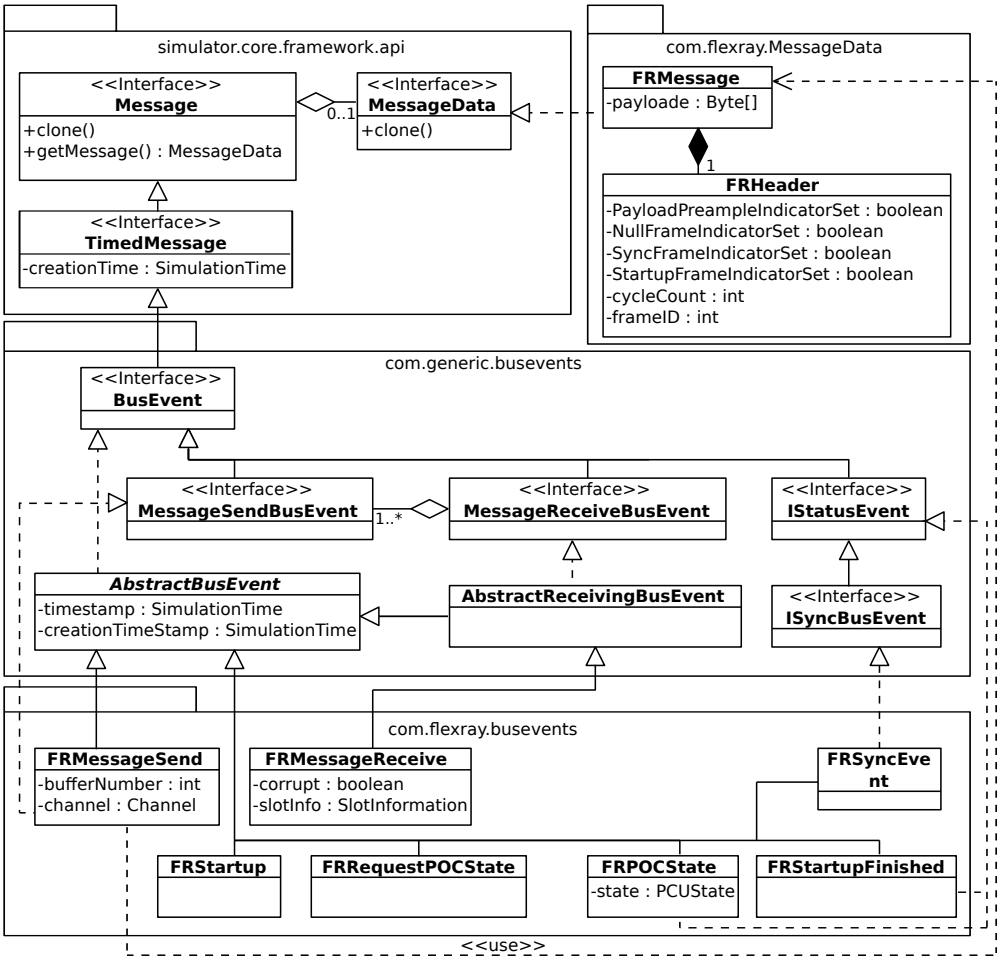


Abbildung 9.9: UML-Klassendiagramm der Nachrichtentypen der FR-CSC.

Neben der Java-Kontrollkomponente werden auch geeignete Nachrichtentypen für die Interaktion zwischen den durch die PCUs simulierten FlexRay-Kommunikationscontrollern (CC) sowie den FSCs benötigt. In Abbildung 9.9 sind die durch die FR-CSC bereitgestellten und verwendeten Nachrichten- und Nachrichtendatentypen und deren Zusammenhang mit dem `COM_Generic`-Bundle dargestellt. Der Nachrichtendatentyp `FRMessage` dient der Repräsentation von FlexRay-Rahmen (bestehend aus einer Payload und einem Header). Um einen Sendeauftrag zu erteilen, erstellt eine FCS eine Nachricht des Typs `FRMessageSend` und leitet diese an die Instanz der `FRComComponent` weiter, welche den jeweiligen Bus repräsentiert; empfangene FlexRay-Rahmen werden über Nachrichten des Typs `FR-MessageReceive` an die empfangende FSC übermittelt. Auch die `FRMessageReceive`-Nachricht enthält, neben dem eigentlichen FlexRay-Rahmen, eine Referenz auf den ursprünglichen Sendeauftrag.

Die Nachrichtentypen `FRStartup` und `FRRequestPOCState` werden von der FSC an die FR-CSC gesendet und initiieren Aktionen des simulierten CC, wie z. B. Start[17] oder Integration in einen Cluster (vgl. [Fle05a, Fle10b]) sowie die Abfrage des aktuellen Zustandes. Eine erfolgreiche Integration

Über Schrittweite und Aktivitätsperioden der Direktoren lässt sich die Genauigkeit zudem an die Anforderungen der Simulation anpassen.

[17] An dieser Stelle sei darauf hingewiesen, dass auch der Start eines FlexRay-Clusters sowie die Integration eines Knotens entsprechend der Spezifikation simuliert wird und dementsprechend zunächst erfolgreich absolviert werden muss, bevor eine Kommunikation erfolgen kann.

oder ein erfolgreicher Start eines Clusters wird mit der Nachricht `FRStartupFinished` bestätigt, Informationen zum aktuellen Zustand des CC mittels `FRPOCState` übermittelt.

Der Beginn jedes Kommunikationszyklus wird durch eine `FRSyncEvent`-Nachricht signalisiert. Diese ermöglicht den FSCs (bzw. deren Verhaltensmodellen) eine Synchronisation mit dem Kommunikationszyklus und erlaubt beispielsweise eine abgestimmte Übermittlung von Rahmen, sodass die Ende-zu-Ende-Verzögerung minimiert werden kann (vgl. Kapitel 11 für ein entsprechendes Anwendungsbeispiel). Dies führt im Gegenzug jedoch zu einer engen temporalen Kopplung zwischen Anwendung und Kommunikation und kann in der Praxis zu Fehlern führen (z. B. durch verpasste Deadlines). Zudem wird die Austauschbarkeit des Kommunikationssystems erschwert, da das Verhaltensmodell entsprechend zugeschnitten sein muss, um aus der Synchronisation Vorteile ziehen zu können.

9.7.2.2 Unterstützte Protokollversionen und Erweiterungen

Die FR-CSC unterstützt sowohl den FlexRay 2.1a Standard [Fle05a], der in bereits bestehenden Systemen verwendet wird, als auch den FlexRay 3.0.1 Standard [Fle10b], welcher die Grundlage des ISO 17458 Standards bildet. Das protokollspezifische Verhalten der CC wird durch die PCUs implementiert, daher werden verschiedene PCU-Implementierungen für FlexRay 2.1a und 3.0.1 angeboten (vgl. Kapitel 6.1 sowie [Fle05a, Fle10b]). Hierbei erlaubt unser Simulator auch die Kombination von PCUs für FlexRay 2.1a und 3.0.1 innerhalb eines simulierten Clusters sowie die Evaluation der hieraus unter Umständen resultierenden Fehler, bei falscher Konfiguration.

Die FR-CSC implementiert auch eine erweiterte Variante von FlexRay, die es – entgegen der Standards – erlaubt, die Länge der einzelnen statischen Slots innerhalb des statischen Segments individuell zu konfigurieren. Diese nicht standardkonforme Erweiterung von FlexRay bezeichnen wir als *Extended-FlexRay-Protocol (EFR)*. EFR dient primär als Grundlage für die Evaluation von Protokollerweiterungen und unterstützt daher auch die modusbasierte Kommunikation durch modusbasierte statische Slots (MSSs), wie in Kapitel 6.3 beschrieben.

Diese Freiheitsgrade spiegeln sich natürlich auch in den – gegenüber dem CAN-CSC – deutlich komplexeren Klassenstrukturen für die Konfiguration der FR-CSC wider. Die hierfür notwendigen Strukturen sind auszugsweise in dem UML-Klassendiagramm in Abbildung 9.10 dargestellt.

Die linke Hälfte des Klassendiagramms betrifft (allgemeine) Konfigurationsoptionen in Bezug auf den simulierten Bus. Hierbei definiert die abstrakte Klasse `FRMediumConfiguration` die allgemeine Struktur des Kommunikationszyklus: Länge der einzelnen Segmente, die Anzahl und Länge der statischen Slots sowie die entsprechenden Werte des dynamischen Segments. Die Bezeichner der jeweiligen Konfigurationsparameter entsprechen den innerhalb des FlexRay-Standards verwendeten Konfigurationsparametern (wobei wir uns in der Darstellung auf ausgewählte Attribute beschränken). Die Konfigurationsparameter entsprechen denen realer CC, sodass diese aus der Simulation direkt für den realen Betrieb übernommen werden können.

Abhängig davon, welche Implementierung der `FRMediumConfiguration` instanziiert wird, erhält man ein Simulationsmodell für einen reinen FlexRay 2.1a Bus (`FRMediumConfiguration2`), welcher lediglich PCUs des FlexRay 2.1a Standards akzeptiert oder einen zum FlexRay 3.0.1 Standard konformen Bus (`FRMediumConfiguration3`). Dieser erlaubt den gemeinsamen Betrieb von CC (PCUs), die den FlexRay-Standards 2.1a oder 3.0.1 genügen, innerhalb des gleichen Clusters. Soll EFR verwendet werden, ist eine Instanz der Klasse `EFRMediumConfiguration3` erforderlich. Diese ermöglicht die individuelle Konfiguration (`StaticSlotConfiguration`) jedes Slots des statischen Segments (z. B. Länge und ActionPointOffset), während das dynamische Segment der FlexRay 3.0.1 Spezifikation entspricht. Um einen MSS zu erzeugen genügt es, bei der Slotkonfiguration den Typ `MBSlotConfiguration` zu verwenden. EFR erlaubt es, für jeden MSS individuell die Länge der einzelnen Backoffslots sowie den TSP innerhalb der Backoffslots festzulegen. Über die Anzahl der Backoffslots wird gleichzeitig die maximale Anzahl von konkurrierenden Modes pro MSS festgelegt. Die Definition der beim *Mode-*

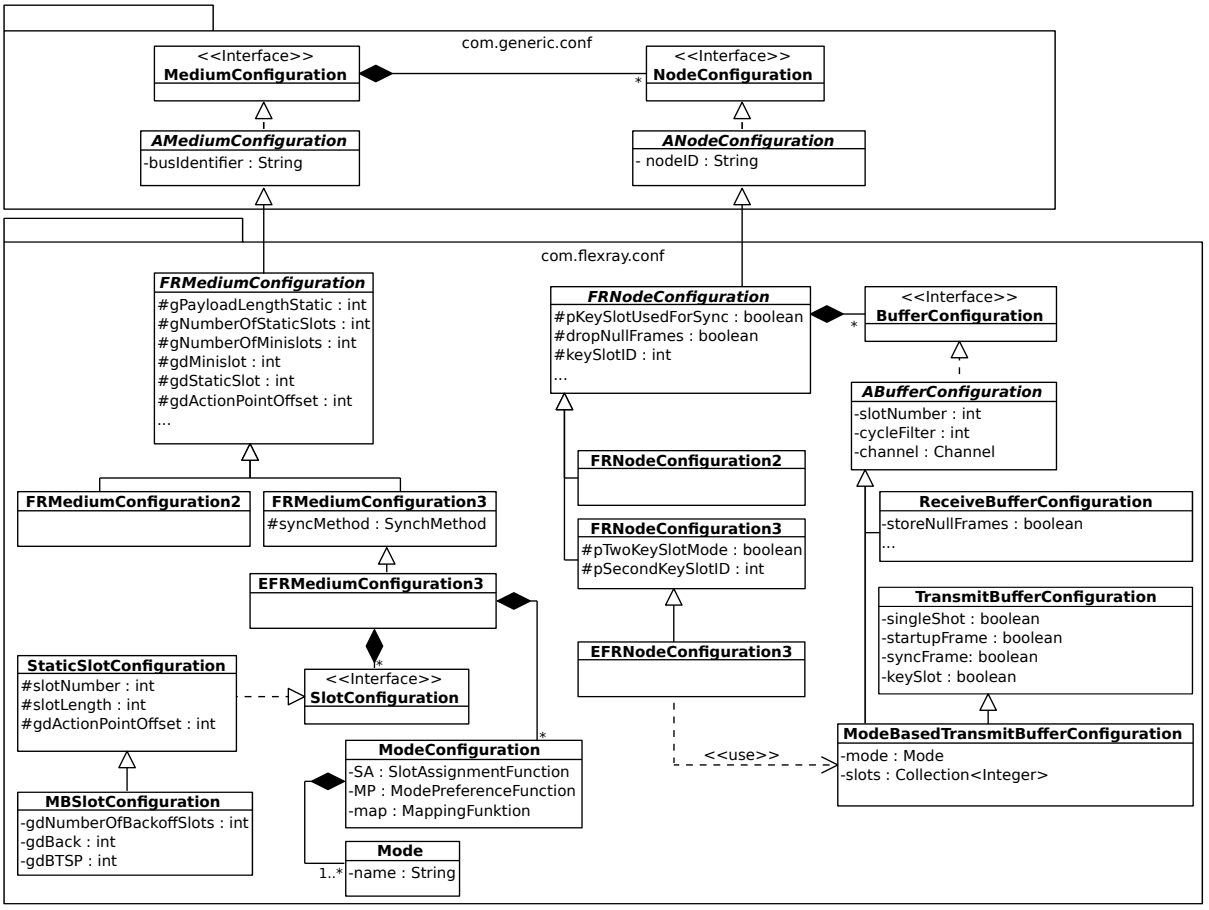


Abbildung 9.10: UML-Klassendiagramm der Konfiguration der FR-CSC.

Based Scheduling zur Verfügung stehenden Modes, deren *Modus-Präferenz* sowie die Abbildung der *Modus-Präferenz* auf die Backoffslots erfolgt durch die Klassen `ModeConfiguration` und `Mode` sowie weiterer Hilfsklassen.

Für die Konfiguration der einzelnen PCUs stehen die Implementierungen der abstrakten Klasse `FR-NodeConfiguration` bereit. Abhängig von der gewählten konkreten Klasse wird die dazu passende Implementierung der PCUs instanziiert (z. B. sorgt `EFRNodeConfiguration3` für die Instanziierung einer PCU mit EFR-Erweiterung). Die Konfiguration der CC (PCU) umfasst auch die *Receive* und *Transmit Message Buffer* sowie deren Zuordnung zu den Slots und physikalischen Kanälen. Die Receive Message Buffer speichern die in den ihnen zugeordneten Slots übertragenen Rahmen zwischen, bevor diese an die FSCs weitergeleitet werden. Die Eigenschaften eines *Receive Message Buffer* werden mittels des Typs `ReceiveBufferConfiguration` spezifiziert[18]. *Transmit Message Buffer* sind jeweils fest mit einem Slot assoziiert und speichern FlexRay-Rahmen solange zwischen, bis diese versendet werden können. Für die EFR-Erweiterung werden die in Kapitel 6.3 beschriebenen und speziell für die modusbasierte Kommunikation angepassten *modusbasierten Transmit Message Buffer* bereitgestellt. Diese sind zusätzlich fest mit einem Mode assoziiert, können dafür aber mit beliebig vielen statischen Slots verknüpft werden. Der für die Übertragung zu verwendende Backoffslot wird, anhand der `Mode-Configuration` in Abhängigkeit von dem jeweiligen Slot, automatisch ermittelt. Die Konfiguration erfolgt über den Typ `ModeBasedTransmitBufferConfiguration`.

[18] Die FlexRay 2.1a und 3.0.1 Standards unterscheiden sich hier nicht.

9.7.2.3 Fehlermodelle

Auch die FR-CSC implementiert das beschriebene, zweistufige Fehlermodell, welches zwischen medium- und knotenspezifischen Fehlern unterscheidet. Erstere betreffen alle Knoten des Busses – wie etwa elektromagnetische Störungen innerhalb des physikalischen Übertragungsmediums –, während letztere der Abbildung von Fehlern innerhalb eines CC (PCU) dienen. Das UML-Klassendiagramm (Abbildung 9.11) zeigt die in der FR-CSC bereits integrierten Fehlermodelle sowie deren Schnittstellen `FlexRayMediumErrorModel` und `FlexRayNodeErrorModel`, welche die Grundlage für eigene Fehlermodelle bilden.

Die Abbildung 9.11 zeigt ferner drei in der FR-CSC enthaltene mediumspezifische Fehlermodelle: `FlexRayNoneMediumErrorModel`, `FlexRayPermanentChannelErrorModel` und `FlexRayStatisticalMediumErrorModel`. Das Fehlermodell `FlexRayNoneMediumErrorModel` modelliert ein Medium ohne Fehler, während `FlexRayPermanentChannelErrorModel` als Fehlerbild die permanente Unterbrechung[19] eines oder beider physikalischer Kanäle nachbildet. Das `FlexRayStatisticalMediumErrorModel` streut mit einer bestimmten Wahrscheinlichkeit Übertragungsfehler auf den jeweiligen Kanälen ein, welche durch die CRC-Prüfung erkannt werden. Die fehlerhaften Rahmen werden, je nach gewählter Konfiguration, entweder verworfen oder die angebundenen FSCs über den Fehler informiert.

In Bezug auf die knotenspezifischen Fehler werden zwei Modelle angeboten: Das `FlexRayNodeErrorModel` simuliert einen fehlerfrei arbeitenden CC, während das `FlexRayStatisticalNodeErrorModel` mit einer bestimmten Wahrscheinlichkeit eine Fehlfunktion des CC (PCU) simuliert. Hierbei kann getrennt festgelegt werden, inwiefern die Sende- bzw. Empfangsfunktionalität betroffen ist.

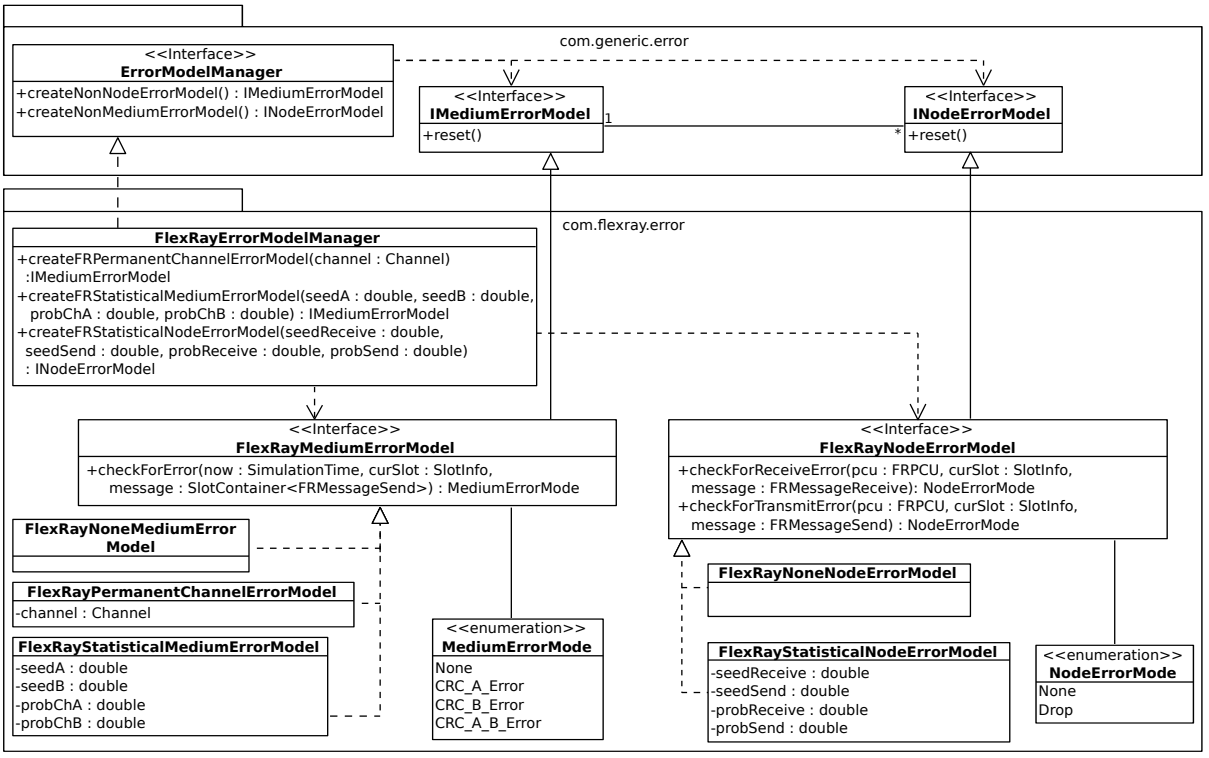


Abbildung 9.11: UML-Klassendiagramm des Fehlermodells von FR-CSC.

[19] Eine weitere Variante erlaubt auch die Modellierung einer temporären Unterbrechung von Kanälen.

Über die Methode `checkForError` des mediumspezifischen Fehlermodells prüft der FlexRay-Medium-Controller für jeden Rahmen, ob ein Fehler simuliert werden soll. Hierzu werden der Methode neben dem Sendeauftrag auch der aktuelle Zeitstempel sowie Informationen über den Slot (sowie Kanal und Nummer des Kommunikationszyklus) übergeben. Die generierten Fehler beschränken sich aktuell auf CRC Fehler, die jeweils auf einem oder beiden Kanälen (siehe `MediumErrorMode`) auftreten können. Analog werden die Methoden `checkForReceiveError` bzw. `checkForSendError` bei dem knotenspezifischen Fehlermodell verwendet, um zu prüfen, ob bei der aktuellen Übertragung ein Fehler der PCU simuliert werden soll (z. B. Verlust des Sendeauftrags oder Verwerfen eines eigentlich korrekt empfangenen Rahmens). Sendet mehr als ein CC innerhalb eines Slots einen Rahmen (z. B. durch eine fehlerhafte Konfiguration), so werden die Rahmen bei der Übertragung verfälscht, d.h., die PCUs empfangen einen Rahmen mit zerstörter CRC und behandeln diesen entsprechend ihren Vorgaben. Die FR-CSC kann zur Zeit nicht für die Simulation der Cliquenbildung bzw. allgemein zur Simulation fehlerhafter Synchronisationen zwischen den einzelnen CCs sowie deren Auswirkungen verwendet werden. Hierfür müssten zunächst die PCUs die internen Zustände der CCs noch genauer abbilden und auch der Synchronisationsvorgang im Detail simulieren.

Auch die FR-CSC unterstützt die allgemeine Debug-Schnittstelle und überträgt spezielle Debug-Nachrichten zur Protokollierung der Ereignisse innerhalb des FlexRay-Clusters. Die in Kapitel 9.6 vorgestellte Visualisierung unterstützt die Darstellung dieser Debug-Nachrichten bereits und kann darüber hinaus auch gleichzeitig die Kommunikation auf mehreren Bussen mit verschiedenen Kommunikationstechnologien veranschaulichen.

9.7.3 Simulator für ein abstraktes Kommunikationsmodell

Die beiden bereits vorgestellten CSCs simulieren die realen Kommunikationstechnologien CAN und FlexRay. Deren Verwendung setzt jedoch Detailkenntnisse und Designentscheidungen hinsichtlich der jeweiligen Kommunikationstechnologie (und deren Konfiguration) voraus. So müssen bei CAN die Identifier festgelegt und die maximale Länge der Payload berücksichtigt werden; bei FlexRay gestaltet sich die Konfiguration des Kommunikationszyklus noch aufwändiger.

Dieser Aufwand ist gerechtfertigt, sofern die Auswahl der konkreten Kommunikationstechnologie bereits abgeschlossen und daher die Erstellung der Konfiguration ohnehin notwendig ist. Ist dies jedoch nicht der Fall, z. B. weil die Anforderungen an die Kommunikationstechnologie noch nicht klar sind, so ist eine zu frühe Festlegung auf eine konkrete Kommunikationstechnologie (samt der damit verbundenen Aufwände) selten sinnvoll. Aus diesem Grund haben wir eine CSC für ein abstraktes Kommunikationsmodell (ACOM-CSC) entwickelt, welches sich mit sehr geringem Aufwand nutzen lässt und in späteren Entwicklungsphasen sehr einfach gegen eine konkrete Kommunikationstechnologie (bzw. deren Simulator) austauschen lässt (vgl. Kapitel 10).

9.7.3.1 Anwendung und Kommunikationsverhalten

Um die Anwendung des abstrakten Kommunikationsmodells in den frühen Entwicklungsphasen attraktiv zu gestalten, beschreibt das Modell ein bewusst minimalistisches Kommunikationsverhalten. Dementsprechend wurden auch die Konfigurationsmöglichkeiten bewusst eingeschränkt, sodass der Aufwand für dessen Verwendung gering bleibt. Dies stellt keine Einschränkung in der Nutzung dar, weil das Kommunikationsmodell in den späteren Entwicklungsphasen ohnehin durch eine konkrete Kommunikationstechnologie ersetzt wird. Böte das abstrakte Kommunikationsmodell zu viele Konfigurationsoptionen an (wie z. B. unterschiedliche Medienzugriffsverfahren, Bitkodierungen, Fehlerbehandlungsmechanismen), so würde dies den eigentlichen Zweck verfehlen, da der Konfigurationsaufwand und Grad der Festlegung dem einer konkreten Kommunikationstechnologie gleich käme.

Das abstrakte Kommunikationsmodell modelliert einen gemeinsamen Übertragungskanal, auf den alle

angeschlossenen Knoten wahlfreien Zugriff haben. Der Kanal verfügt über eine konfigurierbare, konstante Bandbreite, die in Form einer Übertragungsrate spezifiziert wird. Zusätzlich kann der für Übertragungen maximal nutzbare, prozentuale Anteil festgelegt werden. Unser Modell unterstützt drei unterschiedliche Medienzugriffsverfahren oder Semantiken für die Übertragung bzw. den Übertragungskanal, zwischen denen gewählt werden kann:

- Der erste Ansatz dient dem Entwickler als grobe Hilfe, um sicherzustellen, dass die von ihm entworfenen Verhaltensmodelle korrekt mit auftretenden Verzögerungen umgehen können. Bei diesem sehr abstrakten Modell wird lediglich die Übertragungsverzögerung einer Nachricht auf Grundlage der Übertragungsrate berücksichtigt. Es erlaubt, dass mehrere Knoten gleichzeitig einen Rahmen senden, ohne dass es zu Kollisionen oder Verfälschungen kommt. Die Bandbreite wird hierbei nicht auf die einzelnen Übertragungen aufgeteilt, sondern alle Knoten senden mit der vollen konfigurierten Übertragungsrate.
- Der zweite Ansatz basiert auf der Idee eines klassischen CSMA/CA-Verfahrens. Hierbei prüft jeder Knoten vor einer Übertragung (im Simulator übernimmt diese Aufgabe der Medium-Controller in Zusammenarbeit mit den PCUs) zunächst, ob das Medium frei ist. Ist es frei, beginnt er sofort mit dem Senden seines Rahmens, ist das Medium hingegen belegt, wartet der Knoten bis zum Abschluss der laufenden Übertragung. Wurden mehrere Übertragungen für den gleichen Zeitpunkt eingeplant, sequentialisiert der Medium-Controller diese, unter Verwendung einer FIFO-Warteschlange entsprechend dem Zeitpunkt der Erstellung der Sendeaufträge, sodass keine gleichzeitigen Übertragungen stattfinden. Die Rahmen werden dann entsprechend der dort festgelegten Reihenfolge übertragen.
- Die dritte Variante greift die grundlegende Idee des ersten Ansatzes noch einmal auf: Das simulierte Medium gestattet beliebig viele gleichzeitige Übertragungen, ohne dass es zu Interferenzen oder Verfälschungen kommt. Im Gegensatz zur ersten Variante wird die verfügbare Bandbreite jedoch gleichmäßig auf die simultan laufenden Übertragungen aufgeteilt. Dies führt dazu, dass die Verzögerung bei mehreren parallelen Übertragungen, im Vergleich zur Übertragung einer einzelnen Nachricht, steigt. Sobald eine Übertragung abgeschlossen wurde, wird deren Bandbreite wieder auf die noch laufenden Übertragungen aufgeteilt. Analog reduziert sich die Bandbreite der einzelnen Übertragungen, sobald eine zusätzliche Nachrichtenübertragung hinzukommt.

Die erste Variante liefert nur bei sehr geringer Nachrichtendichte ein realistisches Bild der Übertragungsverzögerung, da lediglich die Übertragungsrate, aber nicht das Nachrichtenaufkommen und dessen Verteilung, berücksichtigt wird. Die beiden anderen Verfahren hingegen bestrafen das Auftreten gleichzeitiger Übertragungen; Variante zwei, indem diese eine Serialisierung der Sendeaufträge vornimmt, während Variante drei die verfügbare Bandbreite auf die parallel laufenden Übertragungen aufteilt. Ansatz drei ist insofern interessant, da alle Rahmen, die gleichzeitig übertragen werden, gleichermaßen von höheren Verzögerungen betroffen sind. Variante drei bestraft eine auftretende Überlastung des Mediums proportional, zur Anzahl der gleichzeitig zu übertragenden Rahmen. Da keine Informationen über das System und die Wichtigkeit der Nachrichten für das Verhalten bekannt sind, sollen auf diese Weise die Auswirkungen auf das Gesamtverhalten bei einer von der Auslastung abhängenden erhöhten mittleren Verzögerung nachgebildet werden. Bei einer Serialisierung, wie im zweiten Ansatz, besteht aufgrund mangelnder Informationen stets die Möglichkeit, dass die für das System relevanten Rahmen aufgrund der FIFO-Semantik nicht oder kaum von Verzögerungen betroffen sind.

9.7.3.2 Aufbau der ACOM-CSC und Integration in FERAL

Die Umsetzung des abstrakten Kommunikationsmodells erfolgt durch das OSGi-Bundle `COM_ACOM`. Die Implementierung ist weitestgehend identisch mit dem CAN-CSC und dem FR-CSC und folgt der bereits vorgestellten Architektur. So erbt die Java-Kontrollkomponente `ACOMComComponent` der ACOM-CSC von der allgemeinen `GenericComComponent` (vgl. Abbildung 9.2) und verwendet ebenfalls eine zeitgetriggerte Ausführungssemantik.

Die Nachrichtentypen der ACOM-CSC sind Bestandteil des Java-Package `com.acom.busevents`; für die Repräsentation der Sendeaufträge bzw. der empfangenen Rahmen dienen die beiden Nachrichtentypen `ACOMMessageSend` und `ACOMMessageReceive`. Der Parameter `PCUInternalMessage-Priority` erlaubt die Festlegung einer internen Nachrichtenpriorität. Diese Information wird nur von der PCU benutzt, um die auf eine Übertragung wartenden Rahmen zu ordnen. Wird hierauf verzichtet, so werden die Sendeaufträge durch die PCU in einer FIFO-Warteschlange verwaltet. Die Kodierung der Rahmen erfolgt mittels des Nachrichtendatentyps `ACOMMessage`, welcher die Schnittstelle `MessageData` implementiert. Die Identifikation der einzelnen Rahmentypen erfolgt anhand von IDs (siehe `ACOMHeader`). Die sich hieraus ergebende Klassenstruktur ist weitgehend mit der für die CAN- und FR-CSCs identisch, sodass an dieser Stelle auf eine Darstellung verzichtet wird.

Wie in der Zielsetzung beschrieben, sind die Konfigurationsmöglichkeiten des abstrakten Kommunikationsmodells minimalistisch; die Abbildung 9.12 zeigt das vollständige UML-Klassendiagramm aller für die Konfiguration benötigten Klassen und Attribute. Die Enumeration `ACOMTransmissionMode` erlaubt es, zwischen den drei beschriebenen Simulationsvarianten zu wählen. Die Übertragungsrate wird über das Attribut `speed` in Bit/s konfiguriert. Über das Attribut `useableCapacity` kann der für die Übertragung der Nachrichten nutzbare Anteil festgelegt werden. Mit dem Attribut `overhead` kann jedem Rahmen ein zusätzlicher Overhead (z. B. für Header oder CRC) zugeordnet werden. Dieses Attribut gibt eine feste Anzahl von Bits an, aus der sich zusammen mit den Nutzdaten die vollständige Rahmenlänge ergibt.

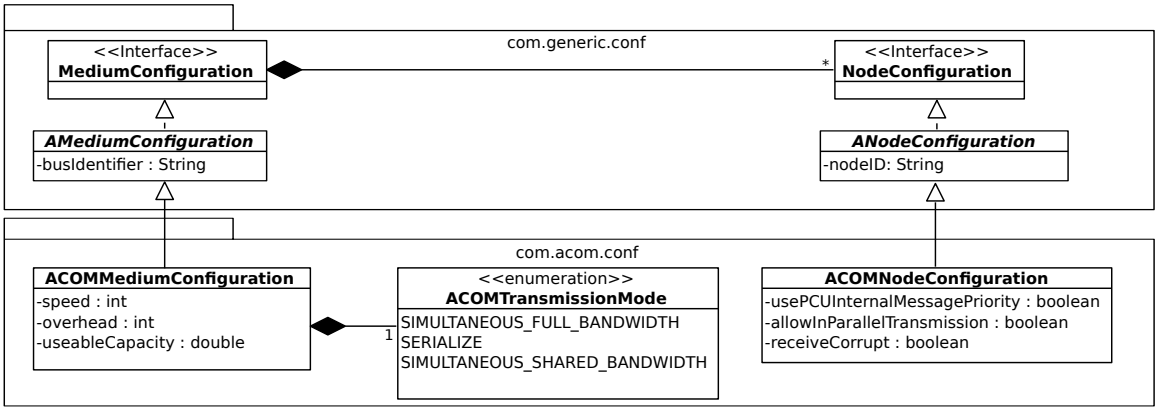


Abbildung 9.12: UML-Klassendiagramm der Konfiguration der ACOM-CSC.

Die Klasse `ACOMNodeConfiguration` konfiguriert das Verhalten der PCU. So kann über das Attribut `usePCUInternalMessagePriotity` die Unterstützung für Rahmenprioritäten innerhalb der PCU aktiviert bzw. die parallele Übertragung von Rahmen durch einen Knoten gestattet werden (allowParallelTransmission – wird nur bei den Varianten 1 und 3 berücksichtigt).

9.7.3.3 Fehlermodelle

Bei der Definition von Fehlermodellen für das abstrakte Kommunikationsmodell ergeben sich ähnliche Überlegungen wie bei der Entwicklung des abstrakten Kommunikationsmodells selbst: Welche Freiheitsgrade sind sinnvoll und mit vertretbarem Aufwand nutzbar und nützlich? Hinzu kommt, dass das abstrakte Kommunikationsmodell bewusst auf die vollständige Definition eines eigenen Kommunikationsprotokolls verzichtet. Insbesondere ist nicht definiert, wie Fehlererkennung, Signalisierung und Behandlung aussehen bzw. ob diese überhaupt unterstützt werden.

Wir haben uns für ein Fehlermodell ähnlich dem von FlexRay entschieden, d.h., Übertragungsfehler werden stets erkannt und fehlerhafte Rahmen entweder der jeweiligen FSC gemeldet oder direkt von

der PCU verworfen (Konfigurationsparameter `receiveCorrupt`, Abbildung 9.12). Der Sender selbst erkennt seine eigene fehlerhafte Übertragung nicht. Wie bei der FR-CSC stellen wir medium- und knotenspezifische Fehlermodelle für die fehlerfreie Übertragung der Rahmen (`ACOMNonMediumErrorModel` und `ACOMNoneNodeErrorModel`) bereit. Diese implementieren die allgemeinen Schnittstellen für die ACOM-CSC spezifischen Fehlermodelle (`ACOMMediumErrorModel` und `ACOMNodeErrorModel`). Auch ein stochastisches Fehlermodell, welches Fehler mit einer konfigurierbaren Wahrscheinlichkeit erzeugt, wurde implementiert. Dieses simuliert die Verfälschung eines Rahmens, welche von den Empfangsknoten detektiert wird. Das stochastische knotenspezifische Fehlermodell ist mit dem entsprechenden Fehlermodell der FR-CSC identisch und erlaubt die Unterscheidung nach Sende- und Empfangsfehler. Auch die ACOM-CSC unterstützt die Debug-Schnittstelle und überträgt Debug-Nachrichten, um die Abläufe bei der Kommunikation zu protokollieren.

9.7.4 Integration des Network Simulator 3 in FERAL

Um neben den speziell für FERAL entwickelten CSCs für CAN und FlexRay auch andere Kommunikationstechnologien zu erschließen, wurde der Network Simulator 3 (ns-3 [ns3ar]) integriert. Dieser ist ein, primär auf Internet Systeme ausgerichteter ereignisbasierter Netzwerk Simulator mit vollständigem TCP/IP- bzw. UDP/IP-Stack sowie Routing-Protokollen und Nachfolger des häufig eingesetzten ns-2 [USCar]. ns-3 unterstützt die Simulation von Ethernet, Switched Ethernet sowie drahtlosen Netzwerken, allen voran WLAN. Ausbreitungsmodelle für Funkwellen und Positionierungen der Knoten erlauben die Berücksichtigung von Mobilität in drahtlosen Netzwerken.

Die Integration des ns-3 in FERAL erfolgte weitestgehend durch Anuschka Igel [BCG+13, BCG+14, Ige15] ebenso wie die Erweiterung des ns-3 für die Simulation drahtloser Netzwerke auf Basis von IEEE 802.15.4. Hierbei wurden insbesondere auch die hardwarespezifischen Eigenarten (Verzögerungen, Zustandsmodell) des CC2420-Transceiver-Transceivers umgesetzt, welcher Bestandteil der *Imote 2*-Plattform ist. Durch diese Ergänzung kann der ns-3 in Verbindung mit der SDL-Integration in FERAL (vgl. Abschnitt 8.4) auch für die modellgetriebene Entwicklung und Evaluation neuer Protokolle auf Basis dieses Transceivers eingesetzt werden. Ein entsprechendes Beispiel für die Verwendung der CC2420-Transceiver-Erweiterung zusammen mit FERAL stellen wir in Kapitel 12 vor. Details der CC2420-Transceiver-Erweiterung sind in [IG13, Gro12, Ige15] zu finden; bezüglich der Integration des ns-3 beschränken wir uns auf eine kurze konzeptionelle Beschreibung[20] der Adaption von FERAL.

Um einen einfachen Austausch zwischen den bereits existierenden CSCs und dem ns-3 innerhalb eines Simulationssystems zu ermöglichen, wurde bei der ns-3-Integration darauf geachtet, die nach außen hin sichtbare Schnittstelle der Java-Kontrollkomponente (Input- und OutputPorts, Nachrichtentypen) so zu gestalten, dass diese mit den Schnittstellen der bereits existierenden CSCs kompatibel ist. Ein zusätzlicher Wrapper sorgt dafür, dass Bridges und Gateways in Verbindung mit dem ns-3 als CSC eingesetzt werden können. Die Implementierung der ns-3-Integration erfolgte durch Anuschka Igel, wobei der Schwerpunkt hier auf der Anbindung des in C++ realisierten ns-3 an FERAL und dessen Steuerung der Ausführung des ns-3 lag. Die Schnittstelle zu FERAL wurde dabei nach dem Vorbild der bereits durch den Autor entwickelten CSCs gestaltet und anschließend durch einen speziellen Wrapper, die für die Kompatibilität mit den Bridges und Gateways erforderliche Funktionalität, ergänzt. Aufgrund der gemeinsamen Schnittstelle des ns-3 mit den anderen CSCs bezeichnen wir den in FERAL integrierten ns-3 zusammen mit dessen Wrapper im Folgenden als ns-3-CSC.

Da der ns-3 in C++ realisiert wurde, kommt, wie auch schon bei der Integration von SDL und Simulink-Modellen, JNI zum Einsatz. Die ns-3-Integration selbst besteht aus einem Java-Teil, welcher die Schnittstelle zu FERAL in Form einer Java-Kontrollkomponente bereitstellt, sowie einem C++-Teil, der für die Interaktion mit dem ns-3-Kern verantwortlich ist (vergleichbar mit der Matlab-Runtime bei der Matlab Integration). Die Java-Kontrollkomponente implementiert die `SimulationComponent`-Schnittstelle

[20] Details zur Schnittstelle zwischen FERAL und ns-3 und deren Implementierung finden sich in der Dissertation von Frau Igel.

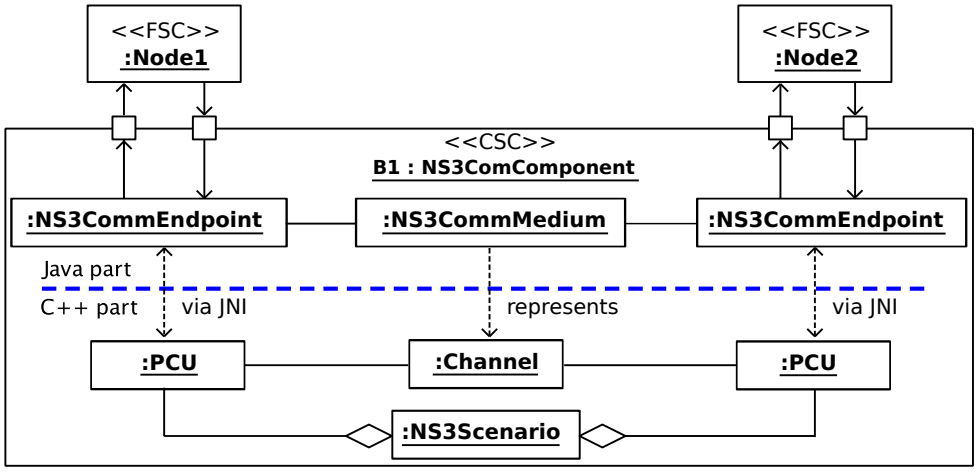


Abbildung 9.13: Verwendung der definierten Input- und OutputPorts zwischen CSCs und FSCs bei der ns-3-CSC gemäß [BCG+13].

und erlaubt FERAL so, die Kontrolle über die Ausführung des ns-3 und ermöglicht den Austausch von Nachrichten mit anderen Simulationskomponenten.

Der erste Schritt der Adaption von FERAL besteht in der Bereitstellung der Java-Kontrollkomponente sowie des C++-Teils, welcher FERAL die Kontrolle des ns-3 ermöglicht. Obwohl der ns-3 ein ereignisbasiertes MOCC implementiert, nutzt die Java-Kontrollkomponente `NS3TimeTriggeredComponent` eine zeitgetriggerte Semantik und basiert auf der `TimeTriggeredSimulationComponent`. Dieses Vorgehen bietet sich an, da der ns-3-Scheduler eine externe Steuerung zulässt, bei der der ns-3 jeweils für eine feste Zeitspanne ausgeführt wird. Auf diese Weise kann FERAL die Zeitbasis des ns-3 kontrollieren und eine semantisch korrekte Simulation gewährleisten, ohne den ns-3 hierfür selbst modifizieren zu müssen.

Die Schnittstelle zwischen der Java-Kontrollkomponente des ns-3 sowie den FSCs orientiert sich an dem Design der drei bereits vorgestellten CSCs bzgl. der Struktur der Input- und OutputPorts. Auch die Idee, jede FSC durch eine PCU zu repräsentieren, welche den Netzwerkzugriff realisiert (vgl. Abbildung 9.3), wurde übernommen. Der interne Aufbau fällt allerdings aufgrund der notwendigen Kopplung zwischen Java und C++ und notwendigen Anpassungen an die Gegebenheiten des ns-3 aufwändiger aus, wie in Abbildung 9.13 illustriert.

In Abbildung 9.13 dargestellt ist die Struktur der ns-3-Integration bei der Simulation eines Kommunikationsmediums mit zwei FSCs. Das Medium wird auf Java-Ebene durch ein Objekt der Klasse `NS3CommMedium` repräsentiert. Die Kommunikationsendpunkte (`NS3CommEndpoint`) fungieren als Schnittstelle zwischen FSC und dessen PCU. Die Verbindung zwischen den FSCs und deren Kommunikationsendpunkt erfolgt über die Links von FERAL, über je einen Input- und OutputPort pro PCU/-Kommunikationsendpunkt. Auf der C++-Seite entspricht jeder Kommunikationsendpunkt einer PCU, welche das protokollspezifische Verhalten für die Anbindung an das simulierte Medium implementiert. Das Verhalten der PCUs ergibt sich im Wesentlichen durch die Instanziierung und Verknüpfung des von ns-3 bereitgestellten modularen (simulierten) Netzwerkstacks.

ns-3-CSC stellt verschiedene PCUs zur Verfügung, welche von FERAL für Simulationen genutzt werden können. Ist keine der vorgefertigten PCUs für das Szenario geeignet, muss der Entwickler zunächst eine eigene PCU mit der gewünschten Kombination aus Netzwerkstack, Device[21] und ggf. Routingprotokoll erstellen. Die bereitgestellten PCUs decken zur Zeit WLAN, Ethernet und Switched Ethernet

[21] Ein Device oder Net Device stellt in ns-3 MAC-Funktionalitäten für eine spezifische Kommunikationstechnologie sowie eine Schnittstelle zwischen Netzwerkschicht und physikalischer Schicht bereit.

mit den Protokollen TCP/IP und UDP/IP ab. Alternativ kann auch ohne Protokollstack, unter direkter Verwendung des Data-Link-Layers kommuniziert werden, sodass lediglich die MAC-Funktionalitäten der jeweiligen Kommunikationstechnologie zur Anwendung kommen. Auf der C++-Seite fungiert das `NSScenario` als Container für die Verwaltung der PCUs und der weiteren zu dem simulierten Medium gehörenden Instanzen. Das eigentliche Verhalten des Mediums wird durch ein Objekt der Klasse `Channel` bereitgestellt.

Zu Schritt zwei der Adaption gehört – neben der Festlegung der Input- und OutputPorts – auch die Definition von Nachrichten- und Nachrichtendatentypen, um Sendeaufträge bzw. empfangene Rahmen zwischen FSCs und CSCs austauschen zu können. Der ns-3-CSC stellt die Nachrichtentypen `NS3NetworkPacketSend` und `NS3NetworkPacketReceive` für die Repräsentation von Sende- und Empfangsereignissen. Diese implementieren die von `COM_Generic` definierten `BusEvent`s, sodass die Schnittstelle für den Nachrichtenaustausch der ns-3-CSC kompatibel mit der Schnittstelle der anderen CSCs ist (vgl. Abbildung 9.4 oder 9.9). Für die Speicherung und Übermittlung von Rahmen sowie Netzwerkpaketen steht der spezielle Nachrichtendatentyp `NS3NetworkPacketData` zur Verfügung. Das UML-Diagramm 9.14 zeigt die Nachrichten- und Nachrichtendatentypen und deren Zusammenhang mit den `BusEvent`s aus dem `COM_Generic`.

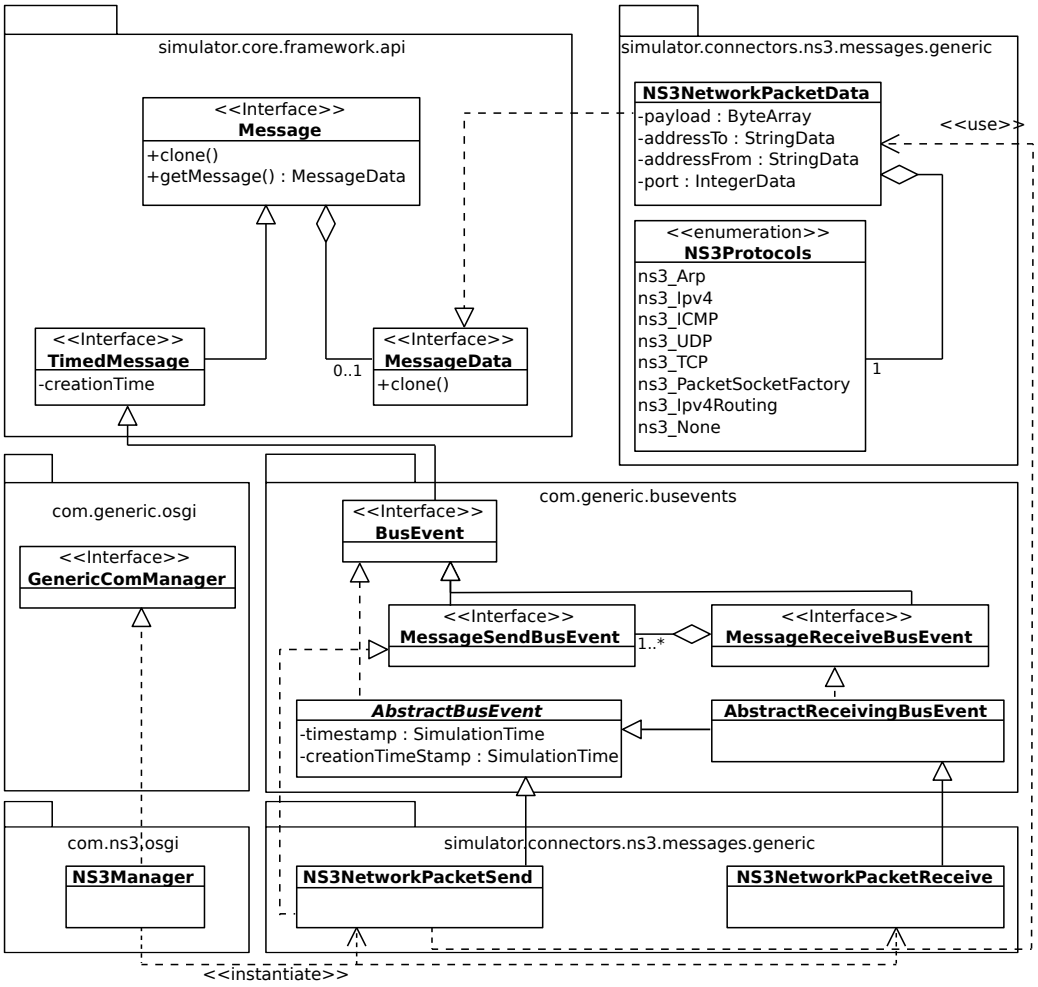


Abbildung 9.14: ns-3-CSC Nachrichtentypen und Nachrichtendatentypen.

Der Wrapper implementiert das von COM_Generic definierte Abstract Factory-Pattern, welches es erlaubt, die wesentlichen Objekte der ns-3-CSC, wie bei den anderen CSCs, über eine einheitliche Schnittstelle zu instanziieren. Ebenfalls werden die für den Einsatz der Bridges (vgl. Kapitel 10.2) und Gateways notwendigen Implementierungen durch den Wrapper bereitgestellt. Hierzu gehören die kommunikationsspezifischen Implementierungen der Nachrichtenfilter, aber auch Funktionen zur Erzeugung der von ns-3 verwendeten Nachrichten- und Nachrichtendatentypen für die Generierung von Sendeaufträgen. Alle direkt zu diesem Wrapper gehörenden Komponenten werden durch das optionale OSGi-Bundle COM_NS3 bereitgestellt, während die restliche ns-3-Integration in einem separaten Bundle gekapselt ist. Der Wrapper wird lediglich benötigt, wenn die ns-3-CSC zusammen mit Bridges und Gateways eingesetzt werden und ein leichterer Austausch zwischen verschiedenen CSCs innerhalb eines Simulationssystems ermöglicht werden soll.

Es bleibt anzumerken, dass der Wrapper keine vollständige Implementierung der in COM_Generic definierten Schnittstelle liefert. So fehlt aktuell die Unterstützung für die Konfigurationsschnittstelle sowie die Möglichkeit, die Konfiguration des ns-3 in Dateien auszulagern (vgl. Kapitel 10.5). Auch Aspekte, die eine Anpassung des ns-3 selbst erfordern, wurden nicht umgesetzt; hierzu gehört die Simulation von Übertragungsfehlern sowie die Integration der Debug-Schnittstelle von FERAL.

10 Austauschbarkeit von CSCs und FSCs

Ein weiteres Anwendungsgebiet von FERAL ist, neben Entwicklung und Test komplexer verteilter Systeme, die Evaluation von Designalternativen. Als Grundlage für den Vergleich unterschiedlicher Designalternativen dienen Simulationen. Die Ergebnisse dieser Simulationen ermöglichen einen objektiven Vergleich von Designalternativen und erlauben somit eine auf reproduzierbaren Resultaten und nachvollziehbaren Fakten basierende Entscheidung. Die Voraussetzungen hierfür schafft FERAL durch die Unterstützung von Simulationskomponenten auf unterschiedlichen Abstraktionsstufen, sodass Simulationen bereits in sehr frühen Entwicklungsphasen ermöglicht werden. Zudem lassen sich über den gesamten Entwicklungsprozess hinweg die Ergebnisse einer solchen Evaluation jederzeit erneut prüfen, da das Simulationssystem zusammen mit den zu entwickelnden Komponenten sukzessive verfeinert werden kann. So kann auch in späteren Phasen ermittelt werden, ob die getroffene Wahl noch tragfähig ist bzw. bei Veränderungen hinsichtlich wichtiger Anwendungsparameter oder Anforderungen die Alternativen noch einmal getestet werden. Designalternativen, die mit Hilfe von FERAL evaluiert werden können, betreffen beispielsweise die Verteilung einzelner Funktionen eines verteilten Echtzeitsystems auf verschiedene Knoten oder die Wahl einer geeigneten Kommunikationstechnologie für solche Systeme. Während sich FERAL prinzipiell für beide Aufgaben eignet, konzentrieren wir uns im Folgenden auf den zweiten Aspekt und betrachten, welche Anforderungen sich hieraus ergeben.

Bei der Evaluation von Designalternativen geht es darum, unterschiedliche konkrete Simulationssysteme von einem abstrakten Simulationssystem abzuleiten, welche dann ihrerseits die zur Wahl stehenden Designalternativen einsetzen. In Bezug auf die Evaluation unterschiedlicher Kommunikationstechnologien für einen bestimmten Bus unterscheiden sich die konkreten Simulationssysteme durch die Nutzung verschiedener CSCs (für den jeweiligen Bus) voneinander[1]. In diesem Kapitel stellen wir die von uns entwickelten Konzepte und Lösungen vor, um die hierbei resultierenden Probleme und Aufwände für die Erstellung der Simulationssysteme zu minimieren.

Um die Austauschbarkeit von CSCs und FSCs zu vereinfachen, nutzen die CSCs eine einheitliche Konfigurationsstruktur, welche es erlaubt, die Konfigurationen außerhalb der eigentlichen Szenarienbeschreibung[2] abzulegen. Eine einfache Austauschbarkeit der CSCs, im Sinne der Instanziierung unterschiedlicher CSCs innerhalb der Szenarienbeschreibung, wird durch den Einsatz des Abstract Factory-Pattern gewährleistet (Kapitel 9.1). Des Weiteren gestatten Bridges die deklarative Beschreibung des Kommunikationsverhaltens von FSCs. Gateways verbinden CSCs untereinander und ermöglichen die deklarative Spezifikation des Weiterleitungsverhaltens zwischen den CSCs. Die Einführung von Bridges und Gateways ermöglicht die Kapselung des kommunikationsspezifischen Wissens innerhalb dieser Komponenten. Beim Einsatz von Bridges werden die FSCs nicht mehr direkt über Links mit den CSCs verbunden, sondern nur indirekt über eine Bridge. Die Möglichkeit, das Kommunikationsverhalten in die Bridge auszulagern erlaubt es, FSCs bzw. deren Verhaltensmodelle ohne Kenntnisse und unabhängig von der verwendeten Kommunikationstechnologie zu spezifizieren. Dementsprechend kann die Kommunikationstechnologie (in FERAL durch eine instanziierte CSC repräsentiert) beliebig ausgetauscht werden, ohne dass eine Anpassung der Verhaltensmodelle der FSCs erforderlich wird. Dies gilt in gleichem Maße auch für

[1] Wie wir sehen werden, ist dies der Idealfall. Unter Umständen kann es auch erforderlich sein, dass unterschiedliche Verhaltensmodelle für die FSCs benötigt werden.

[2] Bei der Szenarienbeschreibung handelt es sich um Java-Code, welcher die Schnittstellen von FERAL nutzt, um ein konkretes Simulationssystem so zu beschreiben, dass FERAL dieses ausführen kann.

die Gateways zwischen verschiedenen CSCs innerhalb des Simulationssystems. Der Entwickler muss lediglich die Regeln, die das Weiterleitungsverhalten festlegen, anpassen.

Bevor wir uns im Detail mit dem Konzept der Bridges und Gateways beschäftigen, betrachten wir in Kapitel 10.1 zunächst, wie mittels FERAL ein Simulationssystem für ein Szenario erstellt wird und welche Zusammenhänge zwischen abstrakten und konkreten Simulationssystemen sowie der Szenarienbeschreibung bestehen. Gleichzeitig werden wir Strategien diskutieren, die dabei helfen, den Wartungs- und Verwaltungsaufwand bei der Evaluation von Designalternativen zu minimieren. Dann widmen wir uns in Kapitel 10.2 den verschiedenen Arten von Bridges sowie der Beschreibung des Kommunikationsverhaltens. Kapitel 10.3 stellt die von FERAL bereitgestellten Gateways vor, während Kapitel 10.4 weitere Nutzungsmöglichkeiten von Bridges als Lastgeneratoren beschreibt. Abschließend widmen wir uns in Kapitel 10.5 den Konventionen für die Konfiguration von CSCs, Bridges und Gateways innerhalb der Szenarienbeschreibung eines Simulationssystems.

Die Ergebnisse dieses Kapitels wurden in [5, 10] publiziert.

10.1 Erstellung eines Simulationssystems

Die Erstellung eines Simulationssystems für FERAL setzt sich aus zwei konzeptionellen Schritten sowie deren Umsetzung in Form einer von FERAL ausführbaren Szenarienbeschreibung zusammen. Der erste Schritt besteht in der Erstellung eines abstrakten Simulationssystems, welches die Struktur und Topologie des zu simulierenden Systems sowie die Rollen bzw. Namen der verwendeten Simulationskomponenten festlegt.

Der zweite Schritte beinhaltet die Erstellung des konkreten Simulationssystems. Um dieses zu erhalten, wird das abstrakte Simulationssystem *instanziiert*, indem für dessen Simulationskomponenten geeignete Simulatoren und deren Verhaltensmodell (FSC) oder bei CSCs die jeweilige Kommunikationstechnologie (bzw. deren Simulator) samt dazugehöriger Konfiguration gewählt werden. Dieser Schritt beinhaltet ggf. auch eine notwendige Verfeinerung der Topologie des abstrakten Simulationssystems (z. B. durch Einführung weiterer Direktoren).

Im letzten Schritt wird eine durch FERAL ausführbare Szenariobeschreibung (aktuell in Form von Java-Code) des konkreten Simulationssystems erstellt, auf deren Basis dann die eigentlichen Simulationen durchgeführt werden.

10.1.1 Abstraktes Simulationssystem

Ein abstraktes Simulationssystem beschreibt Aufbau und Struktur eines Systems, welches mittels FERAL simuliert werden soll. Hierzu gehören Simulationskomponenten, deren Rolle innerhalb des simulierten Systems sowie dessen Topologie (Direktoren, Links, Schachtelung der Direktoren).

Bei der von uns gewählten Darstellung für die Beschreibung eines abstrakten Simulationssystems werden die Simulationskomponenten durch Boxen repräsentiert, wobei der Typ der Simulationskomponenten (CSC, FSC, Bridge oder Gateway) durch einen Stereotypen angegeben wird. Links zwischen den Simulationskomponenten werden durch Linien dargestellt und dienen der Beschreibung der Interaktion zwischen den Simulationskomponenten in einer abstrakten Form. Das heißt, wir verzichten auf dieser Ebene auf gerichtete Links sowie die Auszeichnung der von FERAL verwendeten Input- und OutputPorts[3]. Jede Simulationskomponente wird über einen eindeutigen Namen identifiziert und sollte zur Dokumentation der Rolle der Simulationskomponente innerhalb des Simulationssystems genutzt werden. Zwar legt das abstrakte Simulationssystem den Typ der jeweiligen Simulationskomponente fest, jedoch nicht, welcher

[3] Ein Grund hierfür ist, dass die konkreten Verhaltens- und Kommunikationsmodelle noch nicht festgelegt sind und daher die Ausprägungen (Anzahl, Namen) der verwendeten Input- und OutputPorts der Simulationskomponenten häufig noch nicht feststehen.

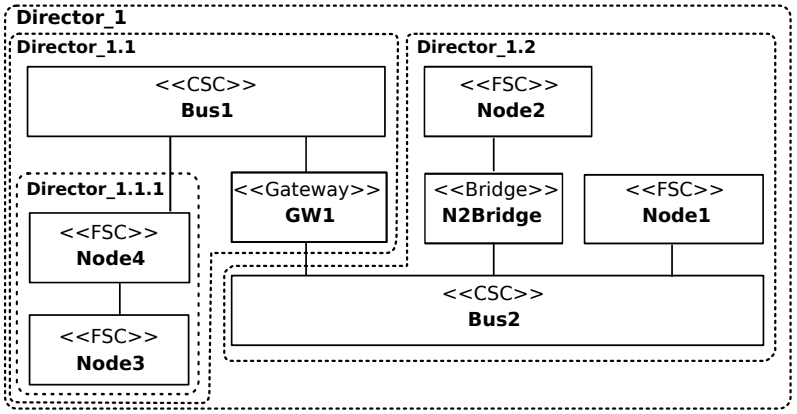


Abbildung 10.1: Beispiel für ein abstraktes Simulationssystem.

Simulator und welches Verhaltens- bzw. Kommunikationstechnologie (Simulator und Konfiguration) für die einzelnen Komponenten zum Einsatz kommt.

Die Topologie des abstrakten Simulationssystems beschreibt die logische Struktur des simulierten Systems aus Sicht von FERAL. Hierzu werden die Simulationskomponenten den jeweiligen Direktoren zugeordnet, welche für deren Ausführung verantwortlich sind. Ein Bestandteil der abstrakten Topologie ist die hierarchische Strukturierung der Direktoren. Diese orientiert sich an inhaltlichen Aspekten der Simulation sowie deren Zielen. So können die Direktoren z. B. genutzt werden, um Simulationskomponenten, die mit einer höheren Genauigkeit simuliert werden sollen, zu gruppieren und diese von Simulationskomponenten zu trennen, für die eine geringere Genauigkeit ausreicht. Auch entsprechende rollenbasierte oder funktionale Gruppierungen sind zulässig und liegen in der Verantwortlichkeit des Entwicklers. Es handelt sich hierbei um eine abstrakte Topologie, da der Typ der Direktoren noch nicht festgelegt ist. Der benötigte Typ hängt von der Wahl der Simulatoren für die Simulationskomponenten und deren MOCC ab, die erst im nächsten Schritt erfolgt[4].

Ein Beispiel für ein abstraktes Simulationssystem ist in Abbildung 10.1 dargestellt. Dieses besteht aus zwei CSCs (*Bus1* und *Bus2*) und insgesamt vier FSCs. Die beiden CSCs sind über das Gateway *GW1* miteinander verbunden. Die Ausführung der Simulationskomponenten sowie deren Interaktion wird von vier teilweise ineinander verschachtelten Direktoren kontrolliert. Die Anbindung der FSC *Node2* an *Bus2* erfolgt über die Bridge mit der Bezeichnung *N2Bridge*. Die Knoten *Node3* und *Node4* hingegen interagieren direkt mit dem simulierten Kommunikationssystem.

Bridges und Gateways sind aus der Sicht von FERAL wiederum eigenständige konkrete Simulationskomponenten. Der Entwickler kann diese innerhalb seines Simulationssystems verwenden, ohne besondere Vorkehrungen treffen zu müssen. Eine ausführliche Erläuterung der Bridges und Gateways sind in Kapitel 10.2 zu finden.

10.1.2 Konkretes Simulationssystem

Aus dem abstrakten Simulationssystem erhält man ein konkretes Simulationssystem, indem man Simulatoren sowie Verhaltensmodelle für alle FSCs und für alle CSCs die zu verwendenden Kommunikationstechnologien (samt Simulatoren) und deren Konfigurationen festlegt.

Um FSCs zu erstellen, stehen in FERAL aktuell drei integrierte Simulatoren für Verhaltensmodelle, in Form von Matlab Simulink-Modellen, SDL-Spezifikationen und Java-Quellcode zur Verfügung (vgl. Ka-

[4] Aus diesem Grund können, nach der Erzeugung des konkreten Simulationssystems, einem Direktor auch Simulationskomponenten mit unterschiedlichen MOCCs zugeordnet sein. Diese Problematik wird bei der Erzeugung der konkreten Topologie durch das Einführen weiterer Direktoren und Schachtelungsebenen gelöst.

pitel 8). Unterstützte Simulatoren für Kommunikationstechnologien sind CAN-CSC und FR-CSC für CAN bzw. FlexRay sowie ACOM-CSC für die Simulation eines abstrakten Kommunikationsmodells. Zusätzlich erlaubt die Integration des ns-3 (ns-3-CSC) die Simulation verschiedener drahtgebundener und drahtloser Kommunikationstechnologien, wie z. B. IEEE 802.3 sowie IEEE 802.15.4, in Verbindung mit unterschiedlichen Protokollen und Protokollstacks (TCP/IP, UPD/IP etc.).

Im Anschluss an die Festlegung der Simulatoren gilt es, die abstrakte Topologie in eine konkrete Topologie zu überführen. Hierzu müssen auf Basis der MOCCs, der FSCs und CSCs die Typen der Direktoren bestimmt und deren Aktivitätsperioden festgelegt werden. Ebenso werden die Links zwischen den Ports der konkreten Simulationskomponenten festgelegt, um deren Interaktion zu realisieren. Sind einem Direktor Simulationskomponenten mit unterschiedlichen MOCCs zugeordnet, wird dieser Konflikt gelöst, indem weitere geschachtelte Direktoren hinzugefügt werden, welche die benötigten MOCCs unterstützen. Die einzelnen Simulationskomponenten werden dann den entsprechenden Direktoren auf Basis der Ausführungssemantik zugeordnet.

Für die Darstellung eines konkreten Simulationssystems orientieren wir uns an der Notation von UML-Objektdiagrammen: Die Bezeichner der konkreten Simulationskomponenten sind unterstrichen dargestellt, um zu illustrieren, dass es sich um konkrete Instanzen handelt. Mittels eines Doppelpunktes von dem Bezeichner abgetrennt wird der Simulator spezifiziert, welcher für die Komponente zum Einsatz kommt. Die gleiche Syntax wird auch verwendet, um die Typen der Direktoren festzulegen. An dieser Stelle sei angemerkt, dass die graphische Darstellung in Abbildung 10.2 ein Simulationssystem nicht vollständig beschreibt, sondern lediglich einen Überblick über den Aufbau gibt. So fehlt beispielsweise die exakte Angabe der zu verwendenden Verhaltensmodelle für die FSCs ebenso wie ein Bezug auf die Konfiguration oder die explizite Darstellung der Input- und OutputPorts der konkreten Simulationskomponenten.

Ein weiterer Bestandteil bei der Erstellung des konkreten Simulationssystems ist, neben der Konfiguration der CSCs, auch die Spezifikation des Kommunikationsverhaltens der Bridges entsprechend den Anforderungen der Anwendung (des FSC). Das Kommunikationsverhalten wird deklarativ, unter Verwendung einer domänenspezifischen Sprache, durch die Definition von Regeln spezifiziert (Kapitel 10.2) und in einer Konfigurationsdatei abgelegt (Kapitel 10.5). Dies gilt ebenfalls für das Weiterleitungsverhalten der Gateways.

Die Abbildung 10.2 zeigt ein Beispiel für ein konkretes Simulationssystem. Dieses wurde aus dem abstrakten Simulationssystem (Abbildung 10.1) durch die Auswahl konkreter Simulatoren sowie Verhaltensmodelle für die FSCs und entsprechender Kommunikationsmodelle für die CSCs gebildet. Das Verhalten der beiden FSCs *Node2* und *Node3* wird durch ein Simulink-Modell beschrieben, wohingegen das Verhalten von *Node1* in Form einer nativen Simulationskomponenten umgesetzt wird. Das Verhalten von Knoten *Node4* wird mittels SDL spezifiziert. *Bus1* verwendet CAN und wird durch die CAN-CSC simuliert, während es sich bei *Bus2* um ein Ethernet basiertes Netzwerk handelt, welches unter Verwendung des ns-3-CSC simuliert wird.

In dem hier gezeigten Beispiel erfordert die Überführung der abstrakten Topologie des abstrakten Simulationssystems in eine konkrete Topologie das Hinzufügen eines zusätzlichen Direktors (`Director_1.2.1`, Abbildung 10.2). Dies ist in dem dargestellten Szenario notwendig, da *Node1* ein ereignisgetriggertes Ausführungsmodell (ETD) nutzt, während alle anderen, dem Direktor `Director_1.2` untergeordneten Simulationskomponenten, ein zeitgetriggertes Ausführungsmodell (TTD) aufweisen. Wir verwenden für `Director_1.2` ein zeitgetriggertes MOCC. Da dieser den Knoten *Node1* nicht direkt ausführen kann, erweitern wir die Topologie um den untergeordneten ereignisgetriggerten Direktors `Director_1.2.1` für die Ausführung von *Node1*. Diese Anpassung der Struktur ist in der Abbildung 10.2 in Rot dargestellt.

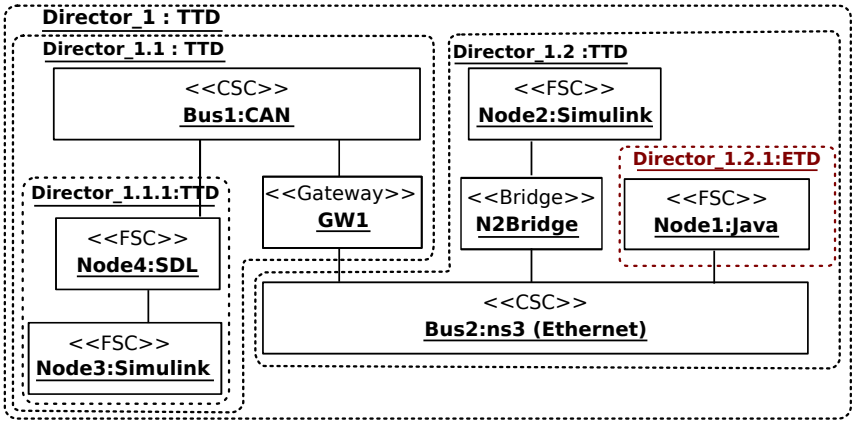


Abbildung 10.2: Beispiel für eine konkretes Simulationssystem.

10.1.3 Repräsentation von Simulationssystemen innerhalb von FERAL

Die Unterscheidung zwischen abstrakten und konkreten Simulationssystemen ermöglicht sowohl den konzeptionellen Überblick über das zu simulierende System und dessen Struktur[5] zu behalten als auch sehr flexibel Designalternativen zu evaluieren. Hierzu werden die für die Evaluation unterschiedlicher Designalternativen benötigten Systeme jeweils in Form eigenständiger konkreter Simulationssysteme abgebildet, die auf einem gemeinsamen abstrakten Simulationssystem beruhen. Bezogen auf das Beispiel aus Abbildung 10.2 könnte ein anderes konkretes Simulationssystem anstelle der CAN-CSC eine FR-CSC für *Bus1* verwenden[6].

Die Umsetzung eines konkreten Simulationssystems in FERAL erfolgt mittels einer Szenariobeschreibung. Hierbei handelt es sich um Java-Code, welcher die Schnittstellen von FERAL nutzt, um das konkrete Simulationssystem in einer von FERAL ausführbaren Form zu spezifizieren. Die Szenariobeschreibung wird als eigenständiges OSGi-Bundle realisiert, welches alle für die Ausführung benötigten Konfigurationen und Verhaltensmodelle enthält. Die Realisierung als Bundle hat sich bewährt, da diese Kapselung eine einfache Archivierung und (erneute) Ausführung erlaubt. Der `Activator` des Bundle, welcher beim Start ausgeführt wird, instanziiert die für die Ausführung benötigten Simulationskomponenten und erstellt die Links zwischen den Ports der Simulationskomponenten für deren Interaktion. Zur Instanziierung der Simulationskomponenten gehört das Laden und Initialisieren der Simulatoren mit deren Verhaltensmodellen (FSCs) bzw. Konfigurationen (CSCs).

Um die Übersichtlichkeit zu verbessern und die Verwaltung sowie Erstellung eines solchen Bundle zu vereinfachen, wird für jedes FSC des Simulationssystems ein eigenes *Bundle Fragment*[7] erstellt. Dieses beinhaltet jeweils alle für die Ausführung des FSCs benötigten Artefakte, wie z. B. das Verhaltensmodell oder zusätzliche Konfigurationsdateien. Bei einem auf Matlab Simulink basierenden FSC enthält das Bundle Fragment sowohl die aus dem Verhaltensmodell erstellte Bibliothek als auch den generierten Simulator Glue-Code zum Laden und Ansteuern des Verhaltensmodells (vgl. Kapitel 8.3). Dies gilt analog für SDL-Systeme (vgl. Kapitel 8.4) und native Simulationskomponenten. Bei diesen ist der Java-Code, welcher das Verhalten implementiert, Gegenstand des Bundle (vgl. Kapitel 8.2).

In Bezug auf die CSC genügt es, den jeweiligen Simulator entsprechend der ausgewählten Kommunikationstechnologie zu instanziieren und diesem eine Konfiguration zu übergeben (vgl. Kapitel 9), in der

[5] In diesem Kontext bezieht sich der Begriff *Struktur* sowohl auf die konkreten Simulationskomponenten und deren Interaktionen als auch auf den Aufbau des Simulationssystems in FERAL (Direktoren, deren Schachtelung, Links etc.).

[6] Zusätzlich wäre hier ebenfalls ein anderes Verhaltensmodell für *Node4* notwendig, da der Knoten direkt mit dem CSC interagiert (vgl. Kapitel 8.4.2).

[7] Ein Fragment erweitert ein OSGi-Bundle, indem es ihm den eigenen Inhalt zur Verfügung stellt.

die Eigenschaften des simulierten Mediums festgelegt sind (vgl. Kapitel 10.5). Dies gilt ebenfalls für die Beschreibung des Kommunikationsverhaltens der Bridges und Gateways.

10.2 Bridges als Schnittstelle zwischen FSCs und CSCs

Um den Aufwand für die Evaluation von Designalternativen in Bezug auf die verwendeten Kommunikationstechnologien zu minimieren werden Konzepte benötigt, welche den Austausch der CSCs innerhalb der Simulationssysteme unterstützen und erleichtern. Idealerweise wird ein Austausch ermöglicht, bei dem keine Modifikationen oder Redefinitionen des abstrakten Simulationssystems sowie der FSCs und deren Verhaltensmodellen erforderlich sind. Um dieses Ziel zu erreichen, haben wir das Konzept der Bridges[8] entwickelt. Eine Bridge fungiert als Verbindungsglied zwischen einer FSC und einer CSC. Ihre Aufgabe besteht darin, kommunikationsspezifische Verhaltensaspekte von dem Verhaltensmodell des FSC zu trennen und zu kapseln. Durch den konsequenten Einsatz von Bridges lassen sich, ausgehend von einem abstrakten Simulationssystem, beliebige konkrete Simulationssysteme ableiten, welche sich lediglich in der Wahl der CSCs (Kommunikationstechnologien) und dem Kommunikationsverhalten der Bridges voneinander unterscheiden.

Hierzu verbinden Bridges FCSs mit CSCs derart, dass kommunikationsspezifisches Wissen und Funktionalitäten (wie z. B. die Erzeugung eines CAN-Rahmens mit einer bestimmten Priorität oder die Nummer eines Nachrichtenpuffers bei FlexRay) innerhalb der Bridges gekapselt und somit vor der FSC (bzw. dessen Verhaltensmodell) verborgen werden. Für den Entwickler des Verhaltensmodells einer FSC ist dieser Mechanismus transparent. Dies erlaubt es, den Umstand, dass die zu verarbeitenden oder erzeugten Daten über ein Kommunikationsmedium übertragen werden, innerhalb des Verhaltensmodells der FSCs zu ignorieren. Hierdurch kann sich der Entwickler auf die Beschreibung der eigentlichen Funktionalität konzentrieren und beschränken. Auch die Schnittstelle der FSC wird nur nach funktionalen Gesichtspunkten gestaltet; über Input- und OutputPorts werden funktional geprägte Daten eingelesen und bereitgestellt, anstatt Rahmen oder Sendeaufträge für eine konkrete, durch eine CSC simulierte Kommunikationstechnologie zu erzeugen. Durch diesen Ansatz wird jede Abhängigkeit zwischen dem Verhaltensmodell und einer konkreten Kommunikationstechnologie vermieden. Stattdessen wird das Kommunikationsverhalten der FSC durch die Bridge realisiert. Diese extrahiert Daten aus den empfangenen Rahmen bzw. erzeugt Sendeaufträge auf Basis der Werte der OutputPorts der FSC. Das Kommunikationsverhalten der Bridges wird mit Hilfe von Regeln deklarativ spezifiziert. Der Austausch einer Kommunikationstechnologie erfordert somit lediglich die Anpassung dieser Regeln. Eine Modifikation der Verhaltensmodelle der über die Bridges angebundenen FSCs entfällt.

Um dies zu erreichen, sind die FSCs nur indirekt über ihre Bridges mit den CSCs verbunden: Die Input- und OutputPorts einer FSC sind über Links mit den Ports ihrer Bridge verbunden, und diese ist ihrerseits wiederum über Links mit einem Port der CSC verbunden. Für die Interaktion zwischen Bridge und FSC wird auf das von FERAL bereitgestellte Typsystem zurückgegriffen, während zwischen Bridge und CSC die von der jeweiligen CSC definierten Nachrichten- und Nachrichtendatentypen zum Austausch von Ereignissen verwendet werden.

Um das Kommunikationsverhalten der Bridges in adäquater Weise spezifizieren zu können, wurde mit der *Domain Brigde Language* (DBL) eine eigens auf diesen Zweck zugeschnittene domänenspezifische Sprache entwickelt. Für die Realisierung der DBL wurde das auf Eclipse aufbauende Werkzeug *Xtext*[9] verwendet. Dieses generiert auf Basis einer vorgegebenen Grammatik neben einem Parser auch einen in Eclipse integrierten Editor mit Syntaxhervorhebung, Codevervollständigung und Syntaxprüfung, welcher

[8] Der Begriff *Bridge* bezeichnet auch Komponenten zur Verbindung von Segmenten eines Netzwerks auf dem Data-Link-Layer (vgl. [Tan03]). Bei den hier beschriebenen Bridges handelt es sich jedoch im Gegensatz dazu, um spezielle Simulationskomponenten für FERAL. Bridges in FERAL übernehmen die Aufgaben einer Middleware im Kommunikationsbereich und dienen als *Glue* zwischen FSCs und CSCs (vgl. Abbildung 10.4).

[9] Verfügbar unter http://www.eclipse.org/Xtext.

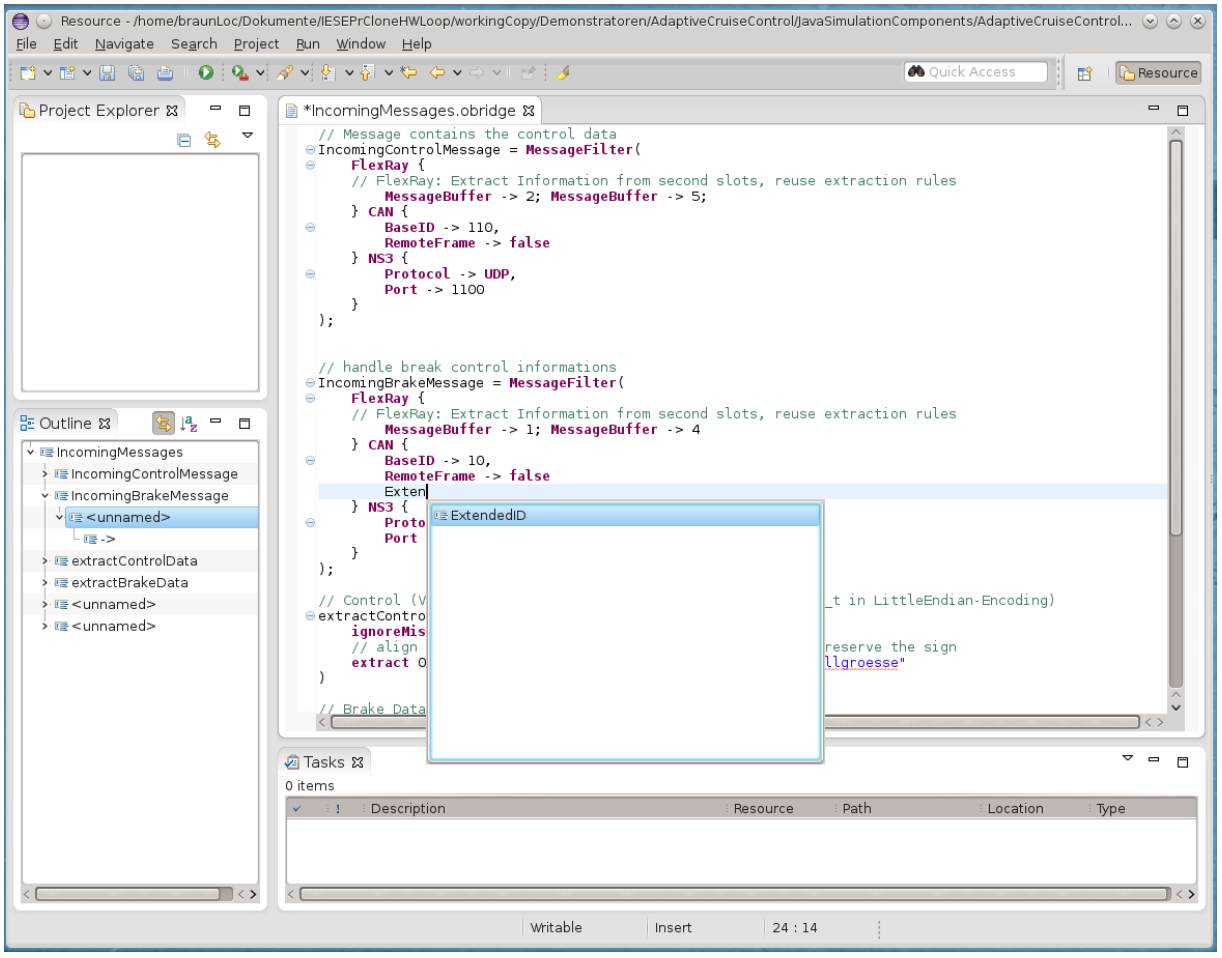


Abbildung 10.3: Beschreibung des Kommunikationsverhaltens einer Bridge innerhalb von Eclipse.

den Designer bei der Entwicklung einer Verhaltensspezifikation unterstützt. Die Verhaltensbeschreibung selbst wird in einer Konfigurationsdatei hinterlegt, die bei der Instanziierung der Bridge durch FERAL geladen wird. Ein Interpreter verwendet die hinterlegten Informationen, um das notwendige Objektgeflecht zur Abbildung des beschriebenen Verhaltens durch die Bridge anzulegen. Die Bridges selbst sind aus der Sicht von FERAL lediglich konkrete Simulationskomponenten, die unter der Kontrolle eines Direktors ausgeführt werden. Die Abbildung 10.3 zeigt den mit Hilfe von *Xtext* erzeugten Eclipse-Editor beim Modifizieren einer Verhaltensbeschreibung unter Verwendung der angebotenen Codevervollständigung.

Wir unterscheiden zwischen zwei verschiedenen Arten von Bridges: *Input2Com-Bridges* und *Com2OutputBridges*, die jeweils einen eigenen Dialekt der DBL zur Beschreibung des Kommunikationsverhaltens verwenden[10]. Eine Input2Com-Bridge verbindet die OutputPorts einer FSC mit einem InputPort der CSC. Zu ihren Aufgaben gehört die Erzeugung von Sendeereignissen, unter Verwendung der Nachrichten- und Nachrichtendatentypen der CSC, basierend auf den Werten der OutputPorts der FSC, in Übereinstimmung mit dem vorgegebenen Kommunikationsverhalten.

Demgegenüber stehen die Com2Output-Bridges. Eine Com2Output-Bridge verbindet einen OutputPort der CSC mit den InputPorts der jeweiligen FSC. Die Aufgabe einer Com2Output-Bridge besteht darin, die von der CSC erzeugten Ereignisse zu interpretieren und für die FSC aufzubereiten. Im Falle eines

[10]Ein dritter Dialekt, der die beiden ersteren miteinander kombiniert, erlaubt die Spezifikation des Weiterleitungsverhaltens von Gateways (vgl. Kapitel 10.3).

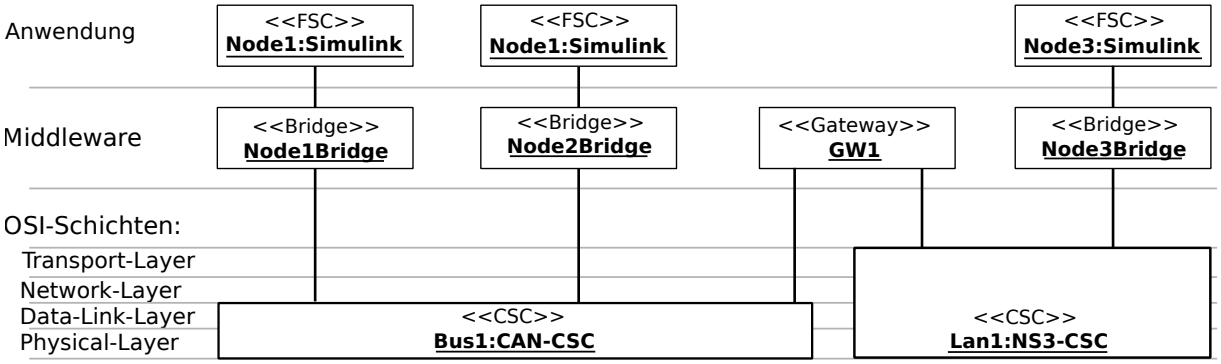


Abbildung 10.4: Logische Struktur eines Kommunikationssystems aus Anwendungssicht beim Einsatz von Bridges.

Rahmenempfangs dekodiert die Com2Output-Bridge die enthaltene Payload, extrahiert die übertragenen Werte (unter Verwendung des von FERAL bereitgestellten Typsystems) und weist diese den InputPorts der FSC zu.

Würde man auf Bridges verzichten, so müssten die beschriebenen Funktionalitäten direkter Bestandteil des Verhaltensmodells der FSC sein und hierfür entsprechendes Wissen in das Verhaltensmodell einfließen. Ein Austausch der CSC wäre dementsprechend mit umfangreichen Änderungen am Verhaltensmodell verbunden. An dieser Stelle sei darauf hingewiesen, dass in der Darstellung der abstrakten und konkreten Simulationssysteme nicht zwischen Input2Com-Bridges und Com2Output-Bridges unterschieden wird. Welche der beiden Varianten der Bridge erzeugt wird, legt der Entwickler durch die bereitgestellten Konfigurationen, die Bestandteil der Szenarienbeschreibung eines konkreten Simulationssystems sind, fest (vgl. Kapitel 10.5). Sind z. B. Konfigurationen für Input2Com- und Com2Output-Bridge einer FSC vorhanden, werden beide Arten von Bridges instanziiert.

Aus Sicht der Anwendung erfüllen die Bridges für die FSCs die Aufgaben einer klassischen Middleware im Kommunikationsbereich und entkoppelt diese von den eigentlichen Kommunikationsabläufen sowie den Details der Kommunikationstechnologie. Dies wird dadurch erreicht, dass zwischen Bridge und Middleware lediglich funktional geprägte Anwendungsdaten ausgetauscht werden. Die eigentliche Kommunikation erfolgt für die FSC transparent und wird durch ihre Bridge(s) abgewickelt[11]. Daher gibt zwischen FSC und CSC innerhalb des Simulationssystems keine Verbindung, die direkte Interaktion mit der CSC erfolgt alleine durch die Bridge (vorwiegend über den Austausch von Sende- und Empfangsereignissen).

Die Abbildung 10.4 zeigt die hieraus resultierende Schichtenarchitektur, die sich beim Einsatz von Bridges ergibt. Die Schichten sind über Links miteinander verbunden, da es sich jeweils um eigenständige Simulationskomponenten handelt. Auf die Darstellung der Ports haben wir aus Gründen der Übersichtlichkeit ebenso verzichtet, wie auf die Unterscheidung zwischen Input2Com-Bridges und Com2Output-Bridges. Dargestellt sind drei FSCs, welche jeweils an die beiden CSCs `Bus1` und `Lan1` über ihre jeweiligen Bridges angebunden sind, welche das Kommunikationsverhalten der FSCs realisieren und die Interaktion mit den CSCs übernehmen. Beide CSCs sind über einen Gateway miteinander verbunden (vgl. Kapitel 10.3). Dieser ermöglicht den Austausch von Informationen zwischen den FSCs der beiden CSCs[12]. Wie in Abbildung 10.4 ebenfalls dargestellt können die von den CSCs simulierten Protokolle

[11] Zu den Aufgaben der Bridges gehören z. B. die Abbildung der Daten in geeigneter Weise auf Rahmen bzw. die Extraktion von Daten aus empfangenen Rahmen unter Berücksichtigung der Eigenarten der verwendeten Kommunikationstechnologie sowie der CSC, die diese simuliert.

[12] Die Funktionen des Gateways gehen über das einfache Weiterleiten von Rahmen hinaus. So kann dieser aus den empfangenen Rahmen und deren Daten selektiv Werte extrahieren, zwischenspeichern und aus diesen neue Rahmen zusammenstellen (unter

unterschiedliche Ebenen des OSI-Referenzmodells umfassen. Die CAN-CSC beschränkt sich auf die Simulation des Data-Link- und Physical-Layer, während die hier verwendete Instanz der ns-3-CSC in dem dargestellten Szenario neben IEEE 802.3 [Ins12] auch einen kompletten TCP/IP-Stack simuliert.

10.2.1 Input2Com-Bridges

Die Input2Com-Bridge verbindet die OutputPorts einer FSC mit dem InputPort der jeweiligen CSC und ist für Erzeugung von Sendeereignissen verantwortlich. Um diese Ereignisse an die jeweilige CSC zu übermitteln, verwendet die Bridge die durch die CSC definierten Nachrichten- und Nachrichtendatentypen.

Hierzu wird zunächst für jede FSC, die eine Input2Com-Bridge für die Abbildung ihres Kommunikationsverhaltens verwendet, eine ebensolche instanziiert. Diese Instanz verfügt für jeden OutputPort der FSC, über einen gleichnamigen InputPort. Die OutputPorts der FSC werden mit den gleichnamigen InputPorts der Input2Com-Bridge über Links verbunden. Die Abbildung 10.5 verdeutlicht dies anhand eines konkreten Beispiels. In diesem ist die FSC `Node1` über ihre instanziierte Input2Com-Bridge `Node1Bridge` an den simulierten CAN-Bus `Bus1` angebunden. In der DBL können die InputPorts der Bridges (bzw. deren aktueller Wert) über die Namen der Ports adressiert werden. Bezüglich der Semantik der InputPorts der Bridges gilt, dass diese jeweils den aktuell am OutputPort der FSC anliegenden Wert solange zwischenspeichern, bis dieser durch einen neuen Ausgabewert der FSC überschrieben wird.

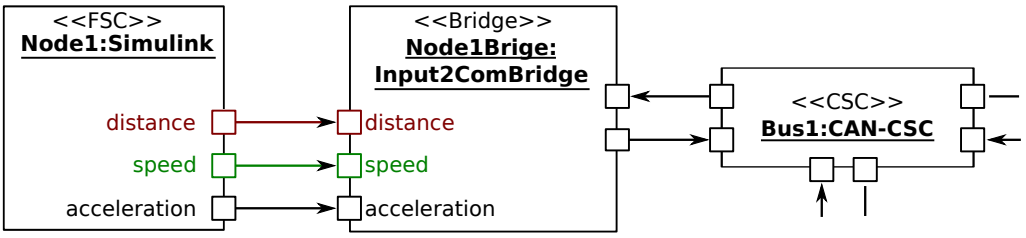


Abbildung 10.5: Schnittstelle zwischen FSCs und Input2Com-Bridges.

Die DBL wurde speziell für die Evaluation von Designalternativen hinsichtlich der Verwendung unterschiedlicher Kommunikationstechnologien entwickelt. Aus diesem Grund kann innerhalb einer einzigen DBL-Spezifikation das Kommunikationsverhalten der Bridge für mehrere unterschiedliche CSCs spezifiziert werden. Zur Laufzeit werden nur die Teile der DBL-Spezifikation interpretiert, welche zu der instanziierten CSC passen, mit der die Bridge verbunden ist. Hierdurch kann eine einzige DBL-Verhaltensspezifikation einer Bridge bei der Evaluation von verschiedenen Kommunikationstechnologien in unterschiedlichen konkreten Simulationssystemen eingesetzt werden. Korrekturen, Wartung und Pflege können somit zentral vorgenommen werden, und man muss diese nicht aufwändig in andere Simulationssysteme (Szenarienbeschreibungen) übernehmen.

Die DBL Spezifikation zur Beschreibung des Kommunikationsverhaltens einer Input2Com-Bridge erfolgt mit Hilfe sogenannter Aktionsregeln. Diese setzen sich aus zwei Teilen zusammen: Der erste Teil legt fest, unter welchen Umständen ein Sendeereignis erzeugt und an die jeweilige CSC übermittelt wird. Der zweite Teil definiert, wie ein solches Sendeereignis (bzw. der erzeugte Rahmen) aufgebaut ist und welche Informationen darin enthalten sind.

Wir widmen uns zunächst dem zweiten Teil der Aktionsregeln und betrachten die Anweisungen zur Beschreibung der Eigenschaften und des Aufbaus der zu generierenden Sendeereignisse, bevor wir uns dem ersten Teil zuwenden.

Berücksichtigung der Anforderungen der Ziel-CSC). Das Verhalten des Gateways wird ähnlich wie das der Bridges über Regeln festgelegt.

10.2.1.1 Festlegen der Eigenschaften und Parameter von Sendeereignissen

Die Anweisungen zur Festlegung der Eigenschaften der zu generierenden Sendeereignisse, bezeichnen wir als *Messagedefinitionen*. Eine solche Messagedefinition legt jeweils für einen Typ von Sendeereignissen sowohl die protokollspezifischen Parameter und Eigenschaften für die Übertragung des Rahmens als auch die Nutzdaten fest. Die Nutzdaten setzen sich aus den Werten der InputPorts der Bridge zusammen, die wiederum ihrerseits vorher durch die FSC bereitgestellt wurden.

Eine Messagedefinition wird mit dem Schlüsselwort `MessageDef` eingeleitet. Um später Bezug auf eine Messagedefinition zu nehmen und diese innerhalb einer Aktionsregel zu referenzieren, können diesen eindeutige Bezeichner zugeordnet werden. Dies verbessert die Übersichtlichkeit und erlaubt es, Messagedefinitionen mehrfach (in unterschiedlichen) Aktionsregeln zu verwenden.

Das Listing 10.1 zeigt die Messagedefinition *vDistanceSensing*. Der von dieser Messagedefinition erzeugte Rahmen enthält die Abstandsinformationen eines Radarsensors, welcher in Form einer nativen Simulationskomponente realisiert wurde. Die Messagedefinition ist in Blöcke unterteilt, die von geschweiften Klammern umschlossen sind. Jeder Block beginnt mit einem reservierten Schlüsselwort (`Generic`, `ACOM`, `CAN`, `FlexRay` oder `NS3`), welches den Typ der CSC festlegt, für den der Block und dessen Parameter gültig ist. Bis auf die Optionen des `Generic`-Blocks gelten die innerhalb eines Blockes spezifizierten Parameter nur für Sendeereignisse, die für eine CSC des entsprechenden Typs erzeugt werden. Das bedeutet, die Optionen innerhalb des `CAN`-Blockes werden nur berücksichtigt, wenn es sich bei der CSC, mit der die Bridge verbunden ist, um eine CAN-CSC handelt. Analog kommen die Informationen des `FlexRay`-Blockes zur Anwendung, wenn statt der CAN-CSC eine FR-CSC instanziiert wird. Mit welcher CSC die Input2Com-Bridge verbunden ist, wird dynamisch bei der Ausführung des Simulationssystems (während der Initialisierungsphase von FERAL) ermittelt.

Listing 10.1: Beispiel einer Messagedefinition für einen Radarabstandssensor.

```
1  vDistanceSensing = MessageDef(
     Generic{
3      ConcatPayload("RadarObstacleDistance", "RadarObstacleSpeed");
     } ACOM {
5      Identifier -> 64;
     } CAN {
7      BaseID -> 20;
       RemoteFrame -> false;
9    } FlexRay{
       ConcatPayload("RadarObstacleDistance", "RadarObstacleSpeed",
         "RadarObstacleAcceleration", "Resolution");
11     MessageBuffer -> 0;
     } NS3{
13     Protocol -> UDP;
       Port -> 200;
15     BroadCast -> true;
     });
```

Falls es sich bei der CSC um eine ACOM-CSC handelt, wird gemäß der Messagedefinition in Zeile 2-6, ein Sendeereignis für einen ACOM-Rahmen mit dem Identifier 64 erzeugt. Handelt es sich um eine CAN-CSC, wird ein Sendeereignis für einen CAN-Datenrahmen mit dem Identifier 20 generiert. Entsprechendes gilt analog für den FlexRay- und NS3-Block. Die in den entsprechenden Blöcken zulässigen Parameter können innerhalb der DBL-Sprachgrammatik eingesehen werden und umfassen im Wesentlichen die Attribute der Nachrichten- und Nachrichtendatentypen der jeweiligen CSCs.

Im Gegensatz zu den protokollspezifischen Blöcken, legt der `Generic`-Block allgemeine Eigenschaften und Parameter fest, welche in allen CSC-spezifischen Blöcken gelten. In Zeile 3 des `Generic`-Blockes wird festgelegt, wie sich die Nutzdaten eines gemäß dieser Messagedefinition erzeugten Rahmens zusammensetzen. Die Byterepräsentation der Nutzdaten wird mit einem sogenannten *Payloadbuilder* erzeugt.

Der hier verwendete `ConcatPayload`-Builder ist Bestandteil des `COM_Generic`-Bundle und nimmt als Parameter eine Liste von Strings mit Namen der InputPorts der Bridge. Die aktuellen Werte der InputPorts[13] werden zunächst in eine Byte-Repräsentation umgewandelt und dann zu einer Byte-Sequenz verkettet. Diese Byte-Sequenz bildet die Payload des Rahmens. Mit jeder Ausführung der Messagedefinition, als Bestandteil einer Aktion (siehe unten), wird ein neues Sendeereignis mit den jeweils aktuellen Werten der InputPorts als Payload erzeugt.

Die innerhalb eines CSC-spezifischen Blockes definierten Parameter besitzen eine höhere Priorität und können die innerhalb des `Generic`-Blocks spezifizierten Optionen und Parameter überschreiben. Zeile 10 (Listing 10.1) nutzt dies, um bei der Verwendung von FlexRay weitere Nutzdaten innerhalb des Rahmens zu versenden. So lassen sich protokollspezifische Besonderheiten, wie etwa die größere Nutzdatenmenge bei FlexRay, einfach berücksichtigen.

Um die Bytesequenz der Nutzdaten eines Rahmens zu erhalten, muss der Payloadbuilder zunächst die Werte der InputPorts in ihre Byte-Repräsentation überführen. Für die von FERAL bereitgestellten Typen (vgl. Abbildung 7.2) existieren bereits entsprechende Konvertierungsfunktionen. Zusätzlich definiert FERAL weitere spezielle Nachrichtendatentypen, welche die Schnittstelle `LowlevelData` implementieren. Hierbei handelt es sich um Nachrichtendatentypen, welche die zu verwendende Byterepräsentation direkt bereitstellen, sodass es keiner Konvertierung bedarf. Diese kommen z. B. beim Austausch von Daten mit FSCs auf Basis von Matlab Simulink oder SDL zum Einsatz. So können Verhaltensmodelle bzw. deren Implementierungen die exakte Byterepräsentation für die Erzeugung der Payload vorgeben, da diese von der Byterepräsentation der Java-Typen abweichen kann. Ein in C realisiertes Verhaltensmodell kann andere Datentypen und Kodierungen (Endianess, Anzahl der verwendeten Bytes) als Java verwenden. Dies kann sich auf die Nutzdaten (Länge und Aufbau) und somit auf die Übertragungsdauer[14] auswirken.

Neben der reinen Byterepräsentation reichert der Payloadbuilder den generierten Sendeauftrag zusätzlich um Metainformationen an. Diese sind Erweiterungen des Nachrichtendatentyps, welcher den Rahmen repräsentiert, und enthalten eine Zuordnung zwischen dem Namen des InputPorts und dessen Wert, der in den Nutzdaten kodiert ist. Die Metainformationen werden von den CSCs unverändert weitergeleitet[15] und stehen den Empfängern innerhalb des `MessageSendEvents` zur Verfügung. Diese Informationen können von der Com2Output-Bridge ausgewertet werden und gestatten es, sich auf die einzelnen, in den Nutzdaten kodierten, Elemente zu beziehen und diese gezielt aus der empfangenen Byte-Sequenz zu extrahieren (siehe Kapitel 10.2.2, Abbildung 10.8).

Zusätzlich zu dem bisher vorgestellten `ConcatPayload`-Builder, stellt FERAL noch weitere Payloadbuilder zur Erzeugung der Byte-Repräsentation der Nutzlast (vgl. Kapitel 10.4) bereit. Anstelle des Namens eines existierenden Payloadbuilder kann auch der vollqualifizierte Name einer Java-Klasse angegeben werden, welcher die Schnittstelle `IPayloadBuilder` implementiert. Auf diese Weise kann ein Entwickler eigene Payloadbuilder verwenden. Die entsprechende Schnittstelle `IPayloadBuilder` besteht lediglich aus der Methode `buildPayload` (vgl. Abbildung 10.6). Dieser Methode wird als Parameter ein Objekt übergeben, welches den Zugriff auf die InputPorts der Input2Com-Bridge ermöglicht. Zwei weitere Datenstrukturen ermöglichen das Speichern der erzeugten Byterepräsentation der Payload sowie der Metainformationen.

Der Interpreter der DBL-Spezifikation erzeugt zur Laufzeit für jede Messagedefinition eine Instanz des Typs `IEventGenerator`[16]. Diesem Generator wird eine CSC-spezifische Konfiguration des Typs `BusEventGeneratorConfiguration` übergeben, die alle Informationen und Parameter aus der

[13] In unserem Beispiel die Werte der InputPorts der Bridge mit den Namen `RadarObstacleDistance` und `RadarObstacleSpeed`.

[14] Bei Verwendung eines CAN-Busses kann eine andere Kodierung, z. B. die Anzahl der Stuffing-Bits, die Übertragungsdauer beeinflussen.

[15] Natürlich beeinflussen die Metainformationen nicht die simulierte Übertragungsdauer.

[16] Um exakt zu sein, wird die jeweilige zu der verwendeten CSC gehörende Subklasse von `GenericMessageGeneratorImpl` instanziiert (vgl. Abbildung 10.6).

Messagedefinition enthält. Die Erzeugung eines Sendeereignisses erfolgt über den Aufruf der Methode `generateEvents` der `IEventGenerator`-Schnittstelle. Die abstrakte Implementierung `GenericMessageGeneratorImpl` des `COM_Generic`-Bundle stellt die abstrakten Methoden `createBusMessageEvent` und `createBusMessage` bereit, welche bei der Erzeugung eines Sendeereignisses aufgerufen werden. Diese werden von jeder CSC in der entsprechenden Subklasse von `GenericMessageGeneratorImpl` implementiert. Die Methode `createBusMessageEvent` erzeugt den für die CSC spezifischen Nachrichtentyp, während die Methode `createBusMessage` eine Instanz des Nachrichtendatentyps der jeweiligen CSC zur Repräsentation des Rahmens erstellt.

Das UML-Klassendiagramm in Abbildung 10.6 stellt diese Zusammenhänge dar und zeigt die allgemeine Schnittstelle des Payloadbuilders sowie zwei konkrete Ausprägungen dieser Schnittstelle. Wir beschränken uns in der Darstellung auf die wichtigsten Konzepte (Methoden und Attribute), Auslassungen sind mit Punkten gekennzeichnet. Um die Erzeugung der unterschiedlichen Klassen für die CSCs einheitlich zu gestalten, kommt auch hier das Abstract Factory-Pattern zum Einsatz (ohne Abbildung).

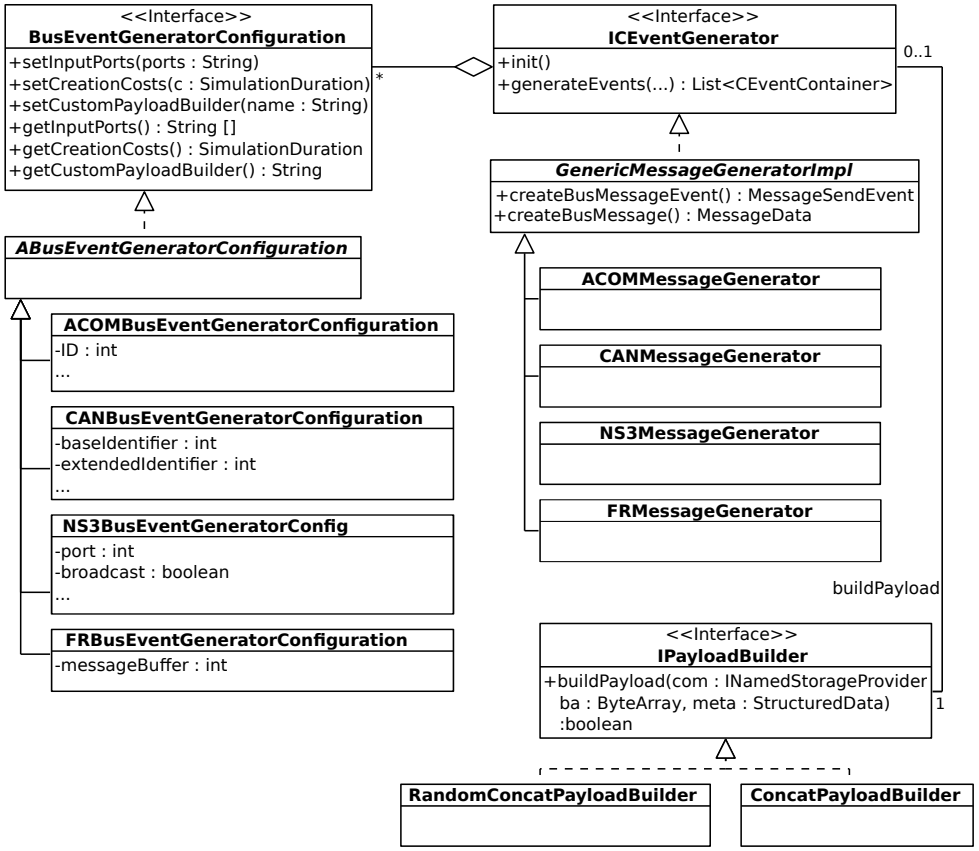


Abbildung 10.6: Beschreibung des Kommunikationsverhaltens einer Bridge innerhalb von FERAL.

10.2.1.2 Auslösen von Sendeereignissen

Messagedefinitionen beschreiben den Aufbau von Rahmen und legen die Übertragungsparameter von Sendeereignissen fest. Nun konzentrieren wir uns auf Mechanismen zur Beschreibung, wann und unter welchen Umständen eine Messagedefinition angewendet wird, um Sendeereignisse zu erzeugen.

Erstellen von Aktionsregeln

Hierzu führt die DBL das Konzept der Aktionsregeln ein. Eine Aktionsregel beginnt immer mit dem Schlüsselwort `on`, gefolgt von einem *Trigger* sowie einer *Aktion*. Der Trigger definiert eine Bedingung, die entweder erfüllt oder nicht erfüllt ist, d.h., zu true oder false evaluiert wird. Ist die Triggerbedingung nicht erfüllt, wird die Abarbeitung der Aktionsregel abgebrochen und mit der nächsten Aktionsregel fortgefahren (siehe unten). Ergibt die Auswertung, dass die Triggerbedingung erfüllt ist, so wird auch die in der Regel definierte Aktion ausgeführt. Diese kann beispielsweise dazu verwendet werden, die Erzeugung eines Sendeereignisses entsprechend einer vorgegebenen Messagedefinition auszulösen. Eine Aktionsregel heißt *ausgeführt* genau dann, wenn die durch den Trigger der Aktionsregel definierte Bedingung erfüllt ist und die Aktion der Regel ausgeführt wird.

Bevor wir uns den Feinheiten widmen, betrachten wir in Listing 10.2 zunächst ein einfaches Beispiel. Die gezeigte Aktionsregel erzeugt periodisch, in einem Intervall von 10 ms, ein Sendeereignis gemäß der Messagedefinition `vDistanceSensing` (vgl. Listing 10.1). Hierzu kommt der generische `PeriodicTrigger` zum Einsatz. Die Erzeugung des Sendeereignisses wird durch die Aktion `SendMessage` ausgelöst, welche als Parameter eine Messagedefinition entgegennimmt[17].

Listing 10.2: Definition einer einfachen periodischen Aktionsregel.

```
on PeriodicTrigger(interval->0:10000000) do SendMessage(vDistanceSensing);
```

Die Gültigkeit einer Aktionsregel lässt sich auf bestimmte CSCs beschränken. In diesem Fall wird die Regel nur angewendet, wenn der OutputPort der Input2Com-Bridge mit einer CSC des jeweiligen Typs verbunden ist. Hierzu wird der Aktionsregel eine Liste der zulässigen CSCs vorangestellt. Mit geschweiften Klammern lassen sich Aktionsregeln zu Blöcken gruppieren, deren Gültigkeit sich auf diese Weise beschränken lässt. Fehlt eine solche Einschränkung, gilt bzw. gelten die Aktionsregeln für alle CSCs.

Die Aktionsregeln, die innerhalb einer DBL-Spezifikation definiert sind und für die jeweilige CSC, mit der die Bridge verbunden ist, gültig sind, werden ihrer Reihenfolge entsprechend nacheinander abgearbeitet. In einem Ausführungsschritt der Input2Com-Bridge können beliebig viele Aktionsregeln *ausgeführt* und somit auch mehrere Ereignisse (z. B. Sendeereignisse) ausgelöst werden[18].

Das Listing 10.3 kombiniert die vorgestellten Möglichkeiten zu einer vollständigen DBL-Spezifikation (bzgl. der Messagedefinition siehe Listing 10.1), die das Kommunikationsverhalten eines Radarsensors für unterschiedliche CSCs (ACOM-, CAN-, FR- und NS3-CSC) beschreibt. Der beschriebene Radarsensor misst den Abstand zu einem vorausfahrenden Fahrzeug und verfügt über einen dedizierten Alarmausgang *Radaralarm*, dessen Zustand sich ändert, sobald ein eingestellter Minimalabstand unterschritten wird und eine Übertragung auslösen soll.

Listing 10.3: DBL-Spezifikation des Kommunikationsverhaltens eines Radarsensors.

```
1 CAN, NS3 {
    on OnRisingEdge("Radaralarm") do SendMessage(vDistanceSensing);
3   on PeriodicTrigger(interval->0:100000000) do SendMessage(vDistanceSensing);
  }
5
  ACOM, FlexRay: on PeriodicTrigger(interval->0:2500000) do SendMessage(vDistanceSensing);
```

[17] Anstelle des Bezeichners der Messagedefinition könnte diese auch direkt innerhalb der Klammern spezifiziert werden.
[18] Diese Semantik kann durch weitere Optionen beeinflusst werden, siehe Kapitel 10.2.1.3.

Kommen als Kommunikationstechnologien CAN bzw. ein durch den ns-3 simuliertes Kommunikationsmedium zum Einsatz, so wird ein Sendeereignis für einen Rahmen mit den Abstandsinformationen erzeugt, sobald eine steigende Flanke an dem OutputPort *Radaralarm* detektiert wird (Zeile 2). Hierfür kommt der vordefinierte Trigger `OnRisingEdge` zum Einsatz.Zusätzlich wird das Sendeereignis periodisch alle 100 ms erzeugt. Diese Modellierung hat den Nachteil, dass auch unmittelbar nachdem bereits ein Sendeereignis aufgrund einer Alarmmeldung generiert wurde (Zeile 2), ein weiteres (identisches) Sendeereignis erzeugt werden kann (durch die Ausführung der Aktionsregel in Zeile 3). In diesem Fall würden unmittelbar aufeinanderfolgend zwei identische Rahmen gesendet. Eine Lösung hierfür bieten *exklusive Gruppen*, deren Semantik in Kapitel 10.2.1.3 behandelt wird, an.

Komplexe Trigger und eigene Triggerdefinitionen

Neben periodischen Triggern (mit und ohne explizitem Startzeitpunkt) stellt `COM_Generic` etliche weitere Trigger bereit, auch solche, die auf eine steigende Flanke sowie auf eine Änderung des Werts eines InputPorts reagieren. Zusätzlich werden Trigger zur Modellierung stochastischer Kommunikationsmodelle bereitgestellt. Bei diesen gibt es keine Triggerbedingung, stattdessen löst der Trigger mit einer bestimmten konfigurierbaren Wahrscheinlichkeit aus. Ein anderer wichtiger Trigger ist der `SyncTrigger`. Dieser erlaubt es, Aktionen zeitversetzt in Bezug auf ein Synchronisationsereignis (Typ `ISyncBusEvent`, vgl. Abbildung 9.4 und 9.9) auszulösen. Ein solches Synchronisationsereignis wird beispielsweise bei FlexRay zu Beginn jedes Kommunikationszyklus generiert und kann dazu verwendet werden, die Erzeugung von Sendeereignissen zeitlich optimal auf die reservierten Zeitslots abzustimmen. Neben der höheren Aktualität der in der Nachricht enthaltenen Daten wird gleichzeitig auch der resultierende Jitter des Nachrichtentyps minimiert. Nachteilig ist die resultierende enge temporale Kopplung zwischen Anwendung und Kommunikationsplan sowie die sich daraus ergebenden Echtzeitanforderungen für die Umsetzung in der Praxis, um eine rechtzeitige Erstellung der Rahmen zu gewährleisten.

Das Listing 10.4 zeigt noch einmal unser Beispiel für die Definition des Kommunikationsverhaltens eines Radarsensors. Diesmal kommt bei FlexRay der `SyncTrigger` zum Einsatz. Das Sendeereignis wird unmittelbar vor dem für die Übertragung exklusiv reservierten Slot erzeugt – hierfür ist ein Offset von 2,5 ms auf den Beginn des Kommunikationszyklus erforderlich.

Listing 10.4: Verwendung des `SyncTrigger` bei zeitgetriggerter Kommunikation.

```
on SyncTrigger(delay->0:2500000) do SendMessage(vDistanceSensing);
```

Um komplexere Bedingungen auszudrücken, kann der Designer Trigger mit logischen Ausdrücken verknüpfen (vgl. Listing 10.5). Zur besseren Übersicht können Trigger auch außerhalb einer Aktionsregel über das Schlüsselwort `Trigger` definiert und mit einem Bezeichner assoziiert werden, der dann innerhalb der Aktionsregeln zum Einsatz kommt.

Genügen die vordefinierten Trigger nicht, können über die Schnittstelle `ICEventTrigger` eigene Trigger implementiert werden. Um diese zu verwenden, muss innerhalb der Aktionsregel der Klassenname der eigenen Trigger-Implementierung verwendet werden. Das UML-Klassendiagramm in Abbildung 10.7 zeigt die Schnittstelle `ICEventTrigger` sowie eine Übersicht der bereitgestellten Trigger. Die nach logischen Verknüpfungen benannten Trigger werden implizit durch den Interpreter verwendet und stehen in der DBL in Form der logischen Operatoren `&`, `|` und `!` zur Verfügung. Die Methode `init` dient der einmaligen Initialisierung eines Triggers, während die Methode `triggerEventCreation` bei jeder Auswertung der Aktionsregel aufgerufen wird und `true` zurückliefert, wenn die durch den Trigger geprüfte Bedingung erfüllt ist. Die weiteren Parameter erlauben dem Trigger den Zugriff auf die InputPorts der Bridge sowie die aktuelle Simulationszeit. Die Methode `resetTrigger` wird aufgerufen, nachdem die Aktionsregel vollständig ausgewertet wurde, hierbei gibt der Parameter an, ob die mit der Aktionsregel verknüpfte Aktion ausgeführt wurde oder nicht.

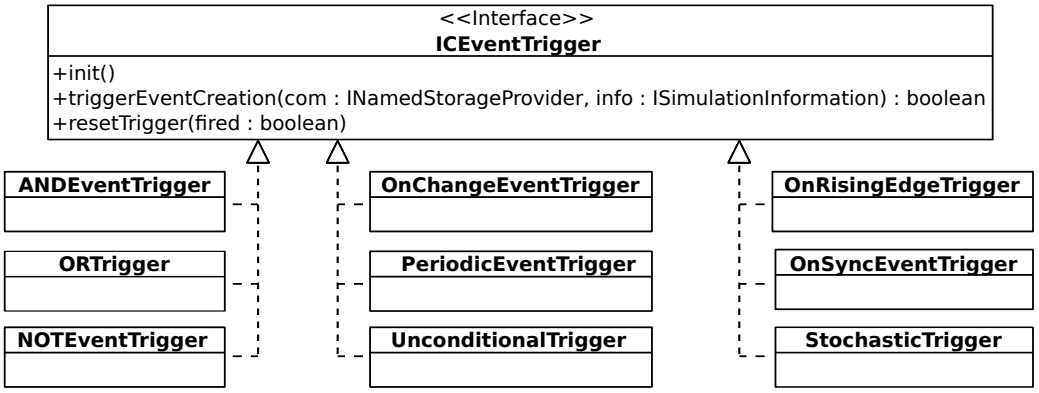


Abbildung 10.7: Bereitgestellte Trigger sowie deren Schnittstelle `ICEventTrigger`.

Das Listing 10.5 zeigt in der Zeile 1 die Definition eines Triggers mit dem Bezeichner `mySpecific-Trigger`, welcher durch den Designer in der Klasse `DemoEventTrigger` implementiert wurde. Die bei der Definition angegebenen Parameter des Triggers werden dem Konstruktor der Klasse bei deren Instanziierung durch den Interpreter übergeben. Zeile 3 illustriert die Verknüpfung von Triggern mittels logischer Ausdrücke. Die zuvor definierten Trigger kommen dann in den Aktionsregeln in Zeile 5 und 6 zum Einsatz.

Listing 10.5: Komplexe Trigger und deren Verwendung.

```
1  mySpecificTrigger = Trigger(com.generic.busBridge.dsl.tests.eventTrigger.DemoEventTrigger
     ( myParameter -> 2.3 ))

3  myTrigger = Trigger(StochasticTrigger(seed->9) & (PeriodicTrigger(startTime->0:0,
     interval-> 2:1) | OnChange("Port1") ))

5  on mySpecificTrigger do SendMessage(m1);
   on myTrigger do SendMessage(m2);
```

Definition von Aktionen

Analog zu den Triggern, stellt `COM_Generic` einige vordefinierte Aktionen bereit, die bei der Formulierung von Aktionsregeln verwendet werden können. Hierzu gehört die Aktion `SendMessage`, welche ein Sendeereignis anhand einer MessageDefinition generiert. Im Gegensatz hierzu erlaubt es die Aktion `SampleValue`, den Wert eines InputPorts der Bridge in einer Variablen zu speichern und diesen zu einem späteren Zeitpunkt zu verwenden. Die Variablen können z. B. innerhalb von Messagedefinitionen verwendet werden, um den gespeicherten Wert als Eingabe für einen Payloadbuilder – und damit für die Nutzdaten eines Rahmens – zu verwenden (Listing 10.6). Ebenso wie bei den Triggern kann der Benutzer auch eigene Aktionen entwickeln, indem er die `ICEventGenerator`-Schnittstelle implementiert. Des Weiteren erlaubt es die Aktion `Combine`, eine beliebige Anzahl von Aktionen (getrennt durch Kommata) miteinander zu verknüpfen, sodass alle aufgeführten Aktionen bei der Ausführung der Aktionsregel angewendet werden.

Listing 10.6 illustriert die Anwendung der verschiedenen Aktionen in einem Beispiel. Hierzu wird zunächst die Messagedefinition `M1` eingeführt, die einen Rahmen mit den Nutzdaten `val1` und `oldVal1` erzeugt. In diesem Beispiel ist `oldVal1` kein InputPort wie `val1`, sondern eine mittels `SampleValue` erzeugte Variable. Der Wert der Variable `oldVal1` wird durch die Aktionsregel in Zeile 13 alle 500 ms auf den aktuellen Wert des InputPorts `val1` gesetzt. In Zeile 10 wird eine durch den Benutzer implementierte

Listing 10.6: Komplexe Trigger und deren Verwendung.

```
M1 = MessageDef(
  Generic{
    ConcatPayload("val1", "OldVal1");
  }
  CAN {
    BaseID -> 20;
    RemoteFrame -> false;
  });

myAction = com.generic.tests.DemoEventGenerator(enableDebug -> true, interval -> 10.3)
on PeriodicTrigger(interval->1:0) do Combine(myAction, SendMessage(M1));

on PeriodicTrigger(interval->0:500000000) do SampleValue("val1" -> "OldVal1");
```

Aktion instanziiert (inkl. Übergabe von Parametern an den Konstruktor der Aktion), die in der Aktionsregel in Zeile 11 jede Sekunde zusammen mit der Erzeugung eines Sendeereignisses gemäß `M1` ausgeführt wird.

10.2.1.3 Weiterführende Konzepte

Wie bereits anhand des Beispiels aus Listing 10.3 beschrieben kann die gewählte Ausführungssemantik bei der Interpretation von DBL-Spezifikationen zur Folge haben, dass mehrere Aktionsregeln innerhalb des gleichen Interpretationsablaufs ausgeführt werden. Dies kann gewünscht sein – wie Listing 10.6 zeigt –, kann aber auch zu unerwünschten Effekten führen. In Listing 10.3 führt dies mitunter dazu, dass die Aktionsregeln der Zeilen 2 und 3 zwei identische Sendeereignisse (und somit Rahmen) erzeugen.

Deshalb erlaubt die DBL die Gruppierung von Aktionsregeln zu *exklusiven Gruppen*. Die Aktionsregeln einer DBL-Spezifikation werden sequentiell abgearbeitet. Wird jedoch eine Aktionsregel ausgeführt, die zu einer exklusiven Gruppe gehört, werden alle nachfolgenden Aktionsregeln, die derselben Gruppe angehören, übersprungen. Die DBL erlaubt die Definition unterschiedlicher exklusiver Gruppen mittels eindeutiger Gruppennamen. Wird auf die Angabe eines Gruppennamens verzichtet, so werden die entsprechenden Aktionsregeln alle der gleichen exklusiven Gruppe zugeordnet. Eine Aktionsregel wird einer exklusiven Gruppe mit dem Schlüsselwort `exklusive` (siehe Listing 10.7) zugeordnet, wobei jede Aktionsregel maximal einer exklusiven Gruppe angehören kann.

Mit den bisherigen Sprachmitteln ist es ebenfalls nicht möglich, die Häufigkeit der Auslösung von Aktionsregeln zu steuern, d.h., wenn die Triggerbedingung einer Aktionsregel erfüllt ist, so wird diese bei jeder Interpretation der DBL-Spezifikation ausgeführt. Gerade bei der Erzeugung von Sendeereignissen ist dies jedoch im Allgemeinen nicht erwünscht. Deshalb kann mit dem Schlüsselwert `block`, gefolgt von einer Zeitangabe, eine Totzeit für Aktionsregeln festgelegt werden. Die Totzeit spezifiziert eine Zeitspanne, die nach der Ausführung einer Regel mindestens verstreichen muss, bevor diese erneut ausgeführt wird (unabhängig von der Bedingung des Triggers). Wird das Konzept einer Totzeit mit der Zuordnung einer Aktionsregel zu einer exklusiven Gruppe kombiniert, so gilt die Totzeit für alle Aktionsregeln der Gruppe.

Mit diesen beiden Konzepten können wir nun die unerwünschten Verhaltensweisen des Beispiels aus Listing 10.3 korrigieren. Listing 10.7 zeigt die entsprechend modifizierten Aktionsregeln für CAN- und NS3-CSC. Beide Aktionsregeln sind einer exklusiven Gruppe mit dem Namen *G1* zugeordnet. Wenn aufgrund eines ausgelösten Alarms (Unterschreitung des Mindestabstandes) die Aktionsregel in Zeile 2 ausgeführt wird, werden für mindestens 20 ms keine weiteren Sendeereignisse mehr erzeugt. Dies gilt auch dann, wenn ein erneuter Radaralarm ausgelöst oder die in Aktionsregel in Zeile 3 spezifizierte Periode

Listing 10.7: DBL-Spezifikation des Kommunikationsverhaltens eines Radarsensors mit Totzeiten.

```
CAN, NS3 {
  on OnRisingEdge("Radaralarm") do SendMessage(vDistanceSensing) exclusive ("G1") block
    for(0:2000000);
  on PeriodicTrigger(interval->0:100000000) do SendMessage(vDistanceSensing) exclusive
    ("G1");
}
```

innerhalb der Totzeit abläuft[19]. Umgekehrt blockiert eine – gemäß dem Zeitplan zulässige – Ausführung der Aktionsregel in Zeile 3 niemals die Erzeugung eines Sendeereignisses aufgrund eines Radaralarms.

Ein Aspekt des Kommunikationsverhaltens wird mit den bisherigen Sprachmitteln der DBL noch nicht widergespiegelt: In der Realität ist die Erzeugung eines Rahmens und die Übermittlung eines Sendeereignisses mit Verzögerungen verbunden. Soll die Simulation möglichst realistisch gestaltet werden, ist auch die Abbildung dieser Verzögerungen innerhalb der Input2Com-Bridges wünschenswert. Aus diesem Grund können sowohl Kosten für die Abarbeitung der DBL-Spezifikation (Auswertung der Triggerbedingungen) als auch für die ausgeführten Aktionen innerhalb der Aktionsregel spezifiziert werden. Kosten sind hier gleichbedeutend mit Verzögerungen, die für die Verarbeitung bzw. Ausführung der jeweiligen Aktion anfallen, z. B. Verzögerungen für die Generierung eines Rahmens inkl. der Übermittlung des Sendeereignisses[20].

Listing 10.8 illustriert dies anhand unseres Radarsensors. So führt die Auswertung der DBL-Spezifikation zu einer simulierten Verzögerung von 5 µs (Zeile 1). Die Erzeugung von Sendeereignissen für den CAN- bzw. NS3-CSC gemäß den Aktionsregeln in Zeile 6 und 7 erfolgt mit einer Gesamtverzögerung von 20 µs (15 µs aufgrund der Aktionsausführung und 5 µs infolge der Interpretation der Regeln der DBL-Spezifikation).

Listing 10.8: Berücksichtigung von Verzögerungen bei der Erzeugung von Sendeereignissen.

```
ConstantCosts (0:5000)

[...]

CAN, NS3 {
  on OnRisingEdge("Radaralarm") do SendMessage(vDistanceSensing) costs (0:15000) exclusive
    ("G1") block for(0:2500000);
  on PeriodicTrigger(interval->0:100000000) do SendMessage(vDistanceSensing) costs
    (0:15000) exclusive ("G1") ;
}
```

10.2.2 Com2Output-Bridges

Com2Output-Bridges bilden aus Kommunikationssicht das Gegenstück zu den Input2Com-Bridges und realisieren das Empfangsverhalten von FSCs. Aufbau und Verbindungsstruktur gleichen denen der Input2Com-Bridges: Für jede FSC, deren Empfangsverhalten mit Hilfe einer Com2Output-Bridge beschrieben werden soll, wird eine eigene Com2Output-Bridge instanziiert. Die Instanz der Com2Output-Bridge verfügt über je einen gleichnamigen OutputPort für jeden InputPort der FSC. Die OutputPorts sind über Links mit den gleichnamigen InputPorts der FSC verbunden. Innerhalb der DBL können die

[19] Dieses Beispiel geht davon aus, dass in diesem Szenario eine durch den Radaralarm ausgelöste Rahmenübertragung innerhalb der Totzeit hinreichend aktuelle Informationen bereitstellt.

[20] Auch innerhalb des `Combine`-Ausdruckes können auf diese Weise den einzelnen Aktionen individuelle Kosten zugeordnet werden.

Namen der OutputPorts verwendet werden, um diesen neue Werte zuzuweisen, welche dann, über die Links, an die InputPorts der FSC weitergeleitet werden. Zusätzlich verfügt die Com2Output-Bridge über einen InputPort, welcher mit einem OutputPort der CSC verbunden ist, sodass die Empfangsereignisse der entsprechenden PCU, welche die FSC repräsentiert, direkt an die Bridge weitergeleitet werden.

Die Aufgabe der Com2Output-Bridges besteht in der Verarbeitung eingehender `BusEvents` und hier insbesondere in der Aufbereitung von Empfangsereignissen (d.h. Instanzen von `MessageReceiveEvent`) für die durch die Bridge repräsentierte FSC. Hierzu gehört die Interpretation der Payload des empfangenen Rahmens, in Abhängigkeit von dessen Identifikationsmerkmalen – wie z. B. CAN-Identifier, FlexRay Frame ID etc. – und die Extraktion der einzelnen Werte aus den Nutzdaten sowie deren Weiterleitung an die InputPorts der FSC. Die Beschreibung des Verhaltens erfolgt mit Hilfe von Nachrichtenregeln in der domänenspezifischen Sprache DBL. Auch hier kann in einer Spezifikation das Empfangsverhalten für unterschiedliche Kommunikationstechnologien spezifiziert werden, sodass die gleiche Spezifikation in verschiedenen konkreten Simulationssystemen genutzt und die Wartung mehrerer Artefakte vermieden werden kann. Hierzu können, wie bei den Input2Com-Bridges, die Regeln zur Definition des Empfangsverhaltens auf bestimmte Typen von CSCs eingeschränkt werden.

Wie bei der DBL-Spezifikation der Input2Com-Bridges, bestehen auch die Nachrichtenregeln der Com2Output-Bridges aus zwei Teilen: Dies sind zum einen *Consume*-Aktionen (kurz C-Aktionen[21]), welche den Umgang mit Empfangsereignissen definieren. Hierzu gehört z. B. die Festlegung, wie Werte aus der Payload zu extrahieren und diese den OutputPorts der Com2Output-Bridge zuzuordnen sind, zum anderen die Definition von Filtern, mit denen jede Nachrichtenregel beginnt. Mittels Filter können empfangene `BusEvents` nach bestimmten Kriterien (z. B. nur Empfangsereignisse mit CAN-Rahmen eines bestimmten CAN-Identifier) selektiert werden. Für jedes eingehende `BusEvent` werden alle Nachrichtenregeln der DBL-Spezifikation der Reihe nach abgearbeitet. Nur wenn das `BusEvent` die durch den Filter spezifizierten Kriterien erfüllt, wird die Aktion der Nachrichtenregel ausgeführt.

Wir beginnen mit der Beschreibung von C-Aktionen und betrachten dann detailliert den Aufbau der Nachrichtenregeln und der Filter.

10.2.2.1 Verarbeitung von Empfangsereignissen mittels Consume-Aktionen

Die einfachste C-Aktion besteht lediglich aus dem Schlüsselwort `Consume`, ohne Angabe weiterer Parameter (siehe Listing 10.9). Grundlage für die Interpretation der Payload des empfangenen Rahmens bilden dann die eingebetteten Metainformationen, welche die absendende Input2Com-Bridge bereitgestellt hat. Diese enthalten zu jedem in der Payload kodierten Wert den dazugehörigen Nachrichtendatentyp sowie den Namen des OutputPorts der FSC, von dem der Wert stammt. Bei einer C-Aktion ohne Parameter prüft die Com2Output-Bridge für alle Werte der Payload anhand der Metainformationen, ob es einen Wert gibt, welcher mit einem Namen assoziiert ist, der dem Namen eines OutputPorts der Bridge entspricht. Ist dies der Fall, wird er aus der Payload extrahiert und dem gleichnamigen OutputPort der Com2Output-Bridge zugewiesen und somit auch dem entsprechenden (gleichnamigen) InputPort der FSC. So lassen sich mit Hilfe der Bridges, anhand der Namen der Input- und OutputPorts, einfach virtuelle Kommunikationskanäle für den Austausch von Daten zwischen FSCs aufbauen.

Die Abbildung 10.8 illustriert dies anhand zweier FSCs, die über eine CAN-CSC mittels zweier Bridges kommunizieren. Die in dem Beispiel von `Node1Bridge` verwendete Messagedefinition nutzt den `ConcatPayload`-Builder, um die Werte der Ports `distance` und `speed` in einem CAN-Rahmen zu versenden. Dieser bettet automatisch entsprechende Metainformationen in das erzeugte `CANMessage`-Objekt ein. Wie in der Abbildung dargestellt, handelt es sich hierbei um Informationen bzgl. des Nachrichtentyps, der Struktur der Payload sowie dem Namen des Ports, aus dem der ursprüngliche Wert stammt[22].

[21] Die C-Aktionen der Com2Output-Bridges sind unabhängig von den Aktionen der Input2Com-Bridges.

[22] Bei den Metainformationen handelt es sich um reine Verwaltungsinformationen für die CSCs und Bridges, die natürlich keinen Einfluss auf die simulierte Übertragungsdauer haben. Wie wir sehen werden, erleichtern die Metainformationen lediglich den

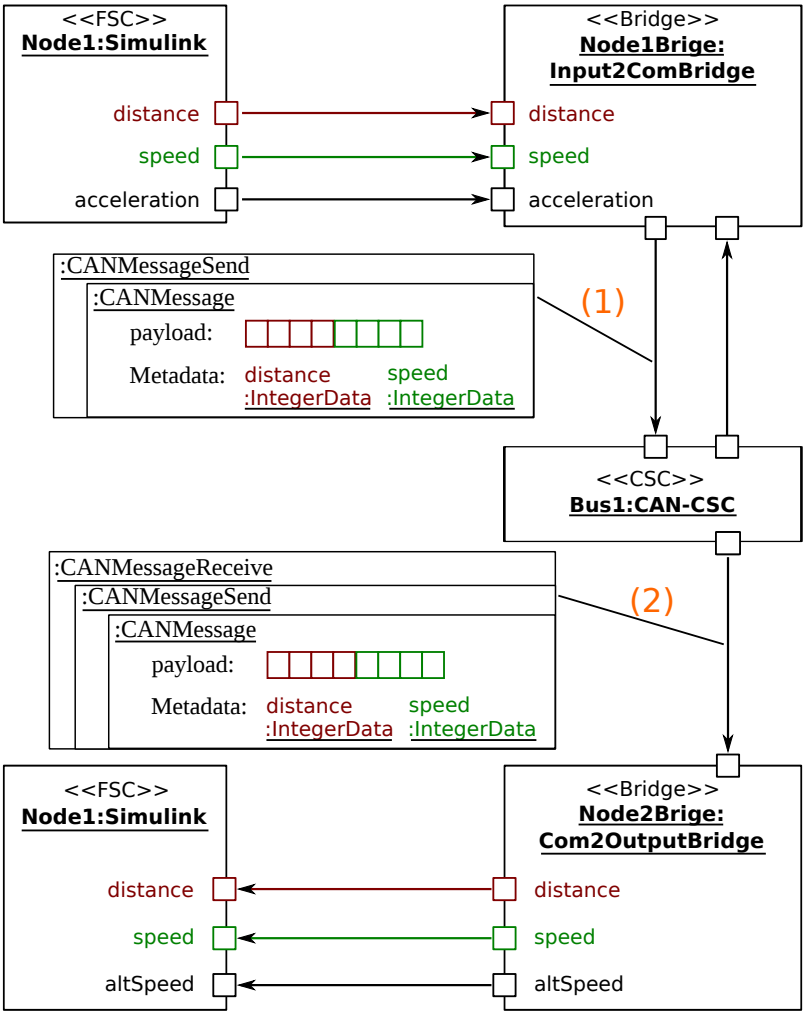


Abbildung 10.8: Beispiel für die Kommunikation über Bridges.

Die `Node2Bridge` verwendet eine C-Aktion ohne Parameter zur Verarbeitung des eingehenden CAN-Rahmens (vgl. Zeile 1 in Listing 10.9).

Das Sendeereignis (1) wird an die CSC übergeben, welche nach der erfolgreichen Übertragung des CAN-Rahmens ein entsprechendes Empfangsereignis (2) erzeugt. Dieses beinhaltet alle Informationen des Sendeereignisses und wird nun durch die `Node2Bridge` verarbeitet. Diese extrahiert anhand der Metainformationen die entsprechenden Nutzdaten aus dem CAN-Rahmen und weist die Werte ihren gleichnamigen OutputPorts zu. In der Abbildung 10.8 sind die beiden virtuellen Kommunikationskanäle für die Übertragung von Werten von `Node1` zu `Node2` in grün bzw. rot hervorgehoben.

Die Verwendung gleicher Bezeichner für Input- und OutputPorts zur Kennzeichnung von Datenströmen erleichtert nicht nur den Aufbau der Simulationssysteme, sondern verbessert auch die Nachvollziehbarkeit der Interaktionen zwischen den FSCs innerhalb des verteilten Systems. Allerdings ist dieser Idealzustand nicht immer gegeben (verschiedene Entwickler, externe zugekaufte Komponenten), deshalb unterstützen C-Aktionen das *Renaming*. Hierbei kann ein (anhand der Metainformationen), innerhalb der Payload, identifizierter Wert einem beliebigen OutputPort zugewiesen werden. Das Listing 10.9 zeigt verschiedene C-Aktionen, die sich auf das in Abbildung 10.8 dargestellte Szenario beziehen. Die C-Aktion ohne

Aufbau der Simulationssysteme. In der Realität muss das Wissen über Aufbau und Kodierung der Nutzdaten eines Rahmens jedem Knoten vorliegen, damit eine Interpretation möglich ist.

Parameter (Zeile 1) bewirkt, die in der Abbildung gezeigte Zuweisung der übertragenen Werte, basierend auf der Namensgleichheit der Input- und OutputPorts von `Node1` zu `Node2` (Ports `distance` und `speed`). Zeile 3 enthält eine C-Aktion, welche *Renaming* verwendet. Wenden wir die C-Aktion `c2` auf das Empfangsereignis (2) aus Abbildung 10.8 an, so wird der innerhalb des Rahmens übertragene Wert des OutputPorts `speed` von `Node1` dem OutputPort der `Node2Bridge` mit dem Namen `altSpeed` zugewiesen. Der Wert des OutputPorts `distance` des Knotens `Node1` wird auch weiterhin mit dem OutputPort `distance` der `Node2Bridge` assoziiert. Ist dieses Verhalten nicht erwünscht, so kann, wie in der C-Aktion in Zeile 5-8, mit dem Schlüsselwort `drop` explizit eine Zuordnung (resp. Wert) verworfen werden.

Listing 10.9: Beispiele für die Definition von C-Aktionen für Com2Output-Bridges.

```
c1 = Consume;

c2 = Consume( "speed" -> "altSpeed" );

c3 = Consume(
  "distance" -> drop,
  "speed" -> "altSpeed"
);
```

Kommen innerhalb eines Simulationssystems ausschließlich Bridges zum Einsatz, um FSCs mit ihren CSCs zu verbinden, so ermöglichen die durch die Input2Com-Bridges bereitgestellten Metainformationen eine einfache Extraktion von Werten. Wäre dies die einzige Möglichkeit, so ergäbe sich hieraus eine nicht akzeptable Einschränkung hinsichtlich der Konstruktion von Simulationssystemen: Werden, z. B. die dedizierten SEnF-Treiber für SDL zur Interaktion mit einer CSC verwendet, so stehen diesem Treiber auf der jeweiligen Abstraktionsebene keine Informationen bzgl. Kodierung und Aufbau der Payload zur Verfügung. Dementsprechend fehlen hier die von den Com2Output-Bridges benötigten Metainformationen. In der Praxis löst der Entwickler das Problem, indem er dem Empfangsknoten Wissen hinsichtlich des Aufbaus der Nutzdaten und der darin kodierten Werte zur Verfügung stellt. Genau diesen Ansatz erlaubt auch die DBL durch die Erweiterung der C-Aktionen um Extraktionsregeln, die diese Informationen explizit bereitstellen.

Eine Extraktionsregel spezifiziert zunächst den Bereich innerhalb der Byte-Sequenz der Nutzdaten, auf den sich diese bezieht. Dann wird ein Konverter festgelegt, um die extrahierten Bytes in einen Nachrichtendatentyp (Objekt der Klasse `MessageData`) zu konvertieren. Für das Typsystem von FERAL stehen hierfür vordefinierte Konverter zur Verfügung. Die Unterstützung anderer Nachrichtendatentypen (z. B. aufgrund eines eigenen Payloadbuilders) ist über die Verwendung eines eigenen Konverters möglich. Dieser muss die Schnittstelle `IByteConverter` implementieren und kann durch Angabe des Klassennamens in der DBL geladen werden.

Um die korrekte Umwandlung in die Nachrichtendatentypen von FERAL sicherzustellen, benötigt der Konverter Informationen über die verwendete Kodierung (Little oder Big-Endian[23]) sowie den Zieldatentyp. Aufgrund der unterschiedlichen Realisierung der FSCs können sich die innerhalb der Nutzdaten verwendeten Kodierungen in Bezug auf deren Länge von der durch die Konverter erwarteten Länge unterscheiden: So kann eine natürliche Zahl mit einem Wertebereich zwischen 0 und 255 in C durch ein Byte repräsentiert werden, obwohl der von FERAL verwendete `IntegerData` Typ zur Repräsentation 4 Bytes verwendet und der Konverter diese (implizit) erwartet (vgl. Listing 10.10). Hierfür stellt die DBL eine Alignment-Funktion zur Verfügung, welche die extrahierte Bytefolge auf die von dem

[23] Fehlt diese Information geht der Konverter von einer Big-Endian Kodierung aus.

Konverter erwartete Länge bringt. Neben der Art des Alignment (Links- oder Rechts-Alignment[24]) kann ebenfalls festgelegt werden, ob die hinzugefügten Bytes mit Nullen oder dem MSB aufgefüllt werden.

In Listing 10.10 extrahiert die C-Aktion `c4` die ersten beiden Bytes der Payload und interpretiert das erste als vorzeichenlose Zahl (Zeile 2) und das zweite als vorzeichenbehaftete Zahl (Zeile 3). Beide Zahlen werden in FERALs Nachrichtendatentyp `IntegerData` konvertiert. Die beiden extrahierten Werte werden den OutputPorts `port1` bzw. `port2` zugewiesen. Zeile 4 und 5 illustrieren das gleiche Vorgehen für die Nachrichtendatentypen `FloatData` sowie `BooleanData`. Zeile 6 demonstriert die Verwendung eines vom Entwickler implementierten Konverters.

Listing 10.10: Beispiele für die Definition von C-Aktionen mit Extraktionsregeln.

```
c4 = Consume(
  extract 0:0 (LittleEndian, AlignRightZero) as Integer to "port1",
  extract 1:1 (LittleEndian, AlignRightMSB) as Integer to "port2",
  extract 2:5 as Float to "port3",
  extract 6:6 as Boolean to "port4",
  extract 7:10 as generic.com.CustomDataTypConverter to "port6"
)
```

10.2.2.2 Aufbau der Nachrichtenregeln zur Spezifikation des Empfangsverhaltens

Die Nachrichtenregeln der Com2Output-Bridges verknüpfen einen Filter und eine Aktion miteinander. Analog zu den Aktionsregeln der Input2Com-Bridges kann für jede einzelne oder für einen ganzen Block von Nachrichtenregeln festgelegt werden, für welche Typen von CSCs diese gültig ist. Jede Nachrichtenregel beginnt mit dem Schlüsselwort `on`, gefolgt von der Definition oder Referenzierung eines *Filters*. Danach kommt das Schlüsselwort `do` und die auszuführende Aktion. Ein Typ von zulässigen Aktionen[25] sind die bereits beschriebenen C-Aktionen, welche die Abbildung der Nutzdaten eines Rahmens auf die OutputPorts beschreiben.

Die Aufgabe des Filters besteht darin, Kriterien für die von einer Nachrichtenregel zu verarbeitenden `BusEvent` festzulegen und diese zu prüfen. Für jedes `BusEvent` werden alle Nachrichtenregeln der DBL-Spezifikation der Reihenfolge nach abgearbeitet. Hierbei wird zuerst geprüft, ob das anliegende `BusEvent` den Filterkriterien entspricht. Ist dies der Fall, wird die in der Regel spezifizierte Aktion ausgeführt und mit der nächsten Nachrichtenregel fortgefahren. Erfüllt das `BusEvent` nicht die spezifizierten Kriterien, wird direkt zur nächsten Regel gesprungen. Wie auch schon bei den Aktionsregeln der Input2Com-Bridges bezeichnen wir eine Nachrichtenregel als ausgeführt genau dann, wenn ihre Aktion ausgeführt wird.

Bevor wir uns den Feinheiten der Filter widmen, betrachten wir zunächst wieder ein einfaches Beispiel. Die in Listing 10.11 spezifizierte Nachrichtenregel wird genutzt, um die durch den Radarsensor versendeten Abstandsinformationen aus dem empfangenen Rahmen zu extrahieren und den OutputPorts der Com2Output-Bridge zuzuweisen. Wir gehen in diesem Beispiel davon aus, dass die Rahmen des Radarsensors durch eine Input2Com-Bridge, gemäß der Messagedefinition aus Listing 10.1, erstellt werden. Als Empfänger fungiert der Knoten `Node2` aus Abbildung 10.8. Da dessen InputPorts andere Bezeichnungen aufweisen als die OutputPorts des Radarsensors (vgl. Listing 10.1), ist ein *Renaming* erforderlich. Dies erfolgt durch die C-Aktion `distanceConsume` in den Zeilen 1-4.

[24] Ein Rechts-Alignment bedeutet, dass das letzte Bit der extrahierten Bytefolge zum LSB wird.

[25] An dieser Stelle sei darauf hingewiesen, dass die Aktion der Nachrichtenregel und die Aktionen der Aktionsregeln nicht miteinander kompatibel sind.

Listing 10.11: Empfang der Nachrichten des Radarsensors.

```
1  distanceConsume = Consume(
     "RadarObstacleDistance" -> "distance",
3    "RadarObstacleSpeed" -> "speed"
   )
5
   distanceConsumeFilter = MessageFilter(
7    ACOM {
       Identifier -> 64
9    } CAN {
       BaseID -> 20,
11     RemoteFrame -> false
     } FlexRay {
13     MessageBuffer -> 3
     } NS3 {
15     Protocol -> UDP,
       Port -> 200,
17     From -> "10.0.1.24"
     })
19
   on distanceConsumeFilter do distanceConsume;
```

Die Definition des Nachrichtenfilters erfolgt in den Zeilen 6-18. Wie auch schon bei den Triggern der Aktionsregeln, können auch Filter außerhalb der Regel definiert und dann referenziert werden (Zeile 20). Der Filter `distanceConsumeFilter` legt für unterschiedliche CSCs fest, welchen Kriterien das Empfangsereignis entsprechen muss, damit die C-Aktion `distanceConsume` angewendet werden kann. Die Schlüsselwörter zur Spezifikation der Filterkriterien entsprechen denen zur Definition der Eigenschaften von Rahmen in den Messagedefinitionen. Ein empfangener Rahmen muss alle (für die jeweilige CSC) in dem Filter aufgeführten Kriterien erfüllen. Wird auf die Angabe eines Kriteriums verzichtet, darf das entsprechende Attribut des Rahmens oder Sendeereignisses einen beliebigen Wert aufweisen. In dem Beispiel in Listing 10.11 würde ein Verzicht auf Zeile 11 (`RemoteFrame -> false`) dazu führen, dass alle CAN-Rahmen mit dem CAN-Identifier 20 die Kriterien erfüllen – sowohl Datenrahmen als auch Datenanforderungsrahmen.

Das UML-Klassendiagramm 10.9 gibt einen Überblick über die Architektur der Com2Output-Bridges hinsichtlich der bereitgestellten Filter sowie deren Struktur. Das Interface `ICEventConsumeFilter` definiert deren allgemeine Schnittstelle; bei der Abarbeitung einer Nachrichtenregel wird für den jeweiligen Filter über die Funktion `canConsume` geprüft, ob das übergebene `BusEvent` die Kriterien des Filters erfüllt. Filter können mit Hilfe von logischen Ausdrücken zu komplexen Filterausdrücken kombiniert werden. Das entsprechende Verhalten wird von den Klassen `NOT-, OR-` und `ANDEventConsume-Filter` bereitgestellt. Da die Filter die Prüfung spezifischer Kriterien der übertragenen Rahmen erlauben, die abhängig von der Kommunikationstechnologie sind, stellt jede CSC hierfür eine eigene Implementierung bereit. Diese implementieren die Schnittstelle `IPSpecificFilter` und operieren auf dem spezifischen Nachrichten- und Nachrichtendatentyp, welche die jeweilige CSC verwendet. Als Container für die Filteroptionen erzeugt der Interpreter der DBL-Spezifikation für jede Filterdefinition ein Konfigurationsobjekt des Typs `BusEventConsumerFilterConfig` und übergibt dieses dem instanziierten Filter. Hierbei wählt der Interpreter die zu instanziierenden Klassen für Filter sowie Konfiguration anhand des Typs der CSC, die mit der Com2Output-Bridge verbunden ist.

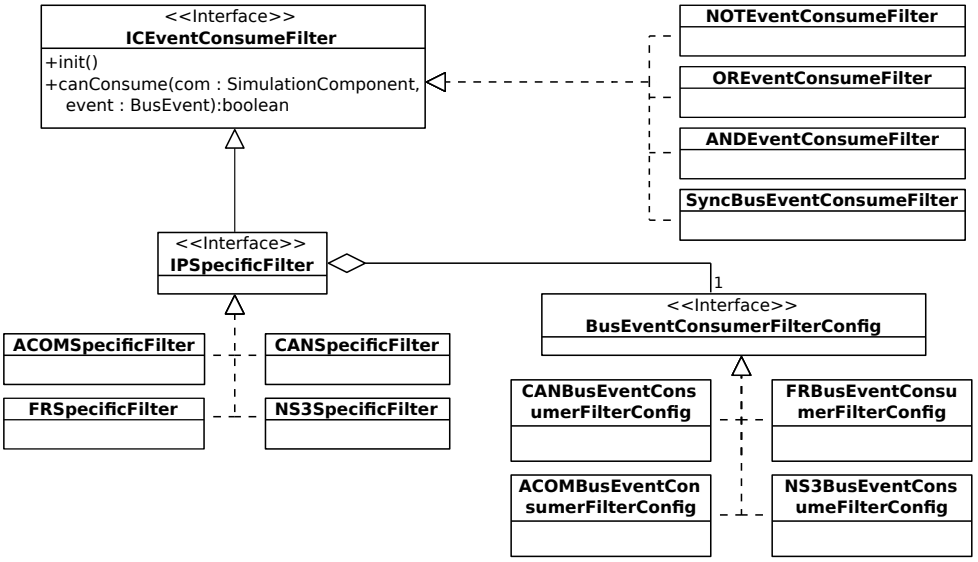


Abbildung 10.9: UML-Klassendiagramm der bereitgestellten Filter und Aktionen für Nachrichtenregeln.

Über die Implementierung der `ICEventConsumeFilter`-Schnittstelle kann der Entwickler eigene Filter realisieren. Innerhalb der DBL-Spezifikation werden diese über ihren Klassennamen referenziert. Dies gilt ebenfalls für Aktionen, auch hier genügt es in der Nachrichtenregel den Namen einer Klasse anzugeben, welche die Schnittstelle `ICEventConsumer` implementiert. Das Listing 10.12 zeigt ein entsprechendes Beispiel für die Verwendung eigener Filter und Aktionen.

Listing 10.12: Verwendung eigener Filter und Aktionen innerhalb von Nachrichtenregeln.

```
myAction = com.generic.MyAction(reportError->true);

on com.generic.MyFiler(seed->2.0) do myAction;
```

10.2.2.3 Weiterführende Konzepte

Manchmal ist eine stärkere Interaktion zwischen dem Verhaltensmodell der FSC und der Com2Output-Bridge als Reaktion auf bestimmte `BusEvents` erforderlich. Aus diesem Grund gibt es die `Toggle`-Aktion. Diese Aktion kann innerhalb einer Nachrichtenregel als Reaktion auf ein eingehendes `BusEvent` ausgeführt werden und sorgt dafür, dass der spezifizierte OutputPort einen anderen Wert annimmt. Das Verhaltensmodell kann hierauf in geeigneter Weise reagieren und z. B. eigene Verarbeitungen anstoßen.

Die bisher vorgestellten Filter beschränkten sich auf den Nachrichtenempfang. Mit dem `SyncBus-EventFilter` (vgl. Abbildung 10.9) kann auch auf Synchronisationsereignisse[26] auf dem simulierten Medium (z. B. den Start eines neuen Kommunikationszyklus bei FlexRay) Bezug genommen werden. Das Beispiel in Listing 10.13 kombiniert diese beiden Konzepte, sodass bei jedem Synchronisationsereignis jeweils eine Flanke an dem OutputPort `Sync` der Com2Output-Bridge erzeugt wird.

[26] Deshalb akzeptiert der Filter nur `BusEvents`, welche die Schnittstelle `ISyncBusEvent` implementieren.

Listing 10.13: Kombination von `SyncBusEventFilter` und `Toggle`-Aktion.

```
1 FlexRay: on SyncFilter() do Toggle("Sync");
```

Ein weiteres Konzept, welches von den Input2Com-Bridges übernommen wurde, ist die Möglichkeit, Kosten in Form von Verzögerungen für die Verarbeitung der Regel sowie deren Aktionen zu spezifizieren. Ebenso können Nachrichtenregeln zu exklusiven Gruppen zusammengefasst und für eine ganze Gruppe oder eine einzelne Nachrichtenregel eine Totzeit festgesetzt werden. Da sowohl Syntax als auch Semantik identisch mit den entsprechenden Konzepten bei der DBL-Spezifikation des Kommunikationsverhaltens von Input2Com-Bridges sind (vgl. Kapitel 10.2.1.3), verzichten wir an dieser Stelle auf eine Darstellung.

10.3 Gateways

In der Praxis bestehen komplexe verteilte Echtzeitsysteme (z. B. im Automotive-Bereich) nicht nur aus einem einzigen Kommunikationsbus, sondern aus mehreren über Gateways miteinander verbundenen Bussen. Sinn ist neben der Bereitstellung der erforderlichen Bandbreite und Kommunikationsgüte die Isolation der Kommunikation zwischen Komponenten mit gemeinsamen Aufgabenbereichen und Sicherheitsanforderungen. Zweck der Gateways ist die Weiterleitung ausgewählter Nachrichten, um z. B. die Interaktionen zwischen verschiedenen Subsystemen unterschiedlicher Busse zu ermöglichen. Bei der Gestaltung der Gateways ist zu beachten, dass sich die Busse hinsichtlich Kommunikationstechnologie, Paradigma (zeit- vs. ereignisgetriggert) sowie der Bandbreite sehr stark unterscheiden können.

10.3.1 Varianten zur Realisierung von Gateways

Die Simulation eines solchen verteilten Echtzeitsystems mit unterschiedlichen Bussystemen ist in FERAL einfach über die (mehrfache) Instanziierung der entsprechenden CSCs abbildbar. Für die Realisierung der Gateways sind unterschiedliche Ansätze denkbar: Die offensichtlichste Variante der Realisierung eines Gateways besteht darin, dass der Entwickler für jedes konkrete Simulationssystem oder Szenario und für jeden dort verwendeten Gateway eine eigene spezialisierte FSC entwickelt – z. B. in Form einer nativen Simulationskomponente. Eine solche Gateway-FSC wird über Links direkt mit den CSCs verbunden und interagiert mit der CSC, über den Austausch von Nachrichten. Der Entwickler muss dabei sicherstellen, dass Gateway-FSCs in der Lage sind, die von den CSCs verwendeten Nachrichten- und Nachrichtendatentypen auf geeignete Weise zu verarbeiten: Basierend auf der Interpretation der Empfangsereignisse erstellt der Gateway-FSC (für ausgewählte Rahmen) wiederum Sendeereignisse für die jeweils andere CSC[27], unter Verwendung deren nativen Nachrichten- und Nachrichtendatentypen. Das Verhalten des Gateways kann beliebig kompliziert ausfallen; so können die Daten mehrerer Rahmen gesammelt und hierfür ein einzelner Rahmen auf dem Zielbus übertragen werden, oder die Nutzdaten eines Rahmens müssen aufgrund der Eigenschaften des Zielbusses auf mehrere Rahmen aufgeteilt werden. Durch die direkte Verwendung der Nachrichten- und Nachrichtendatentypen der CSC ergeben sich in Bezug auf die Evaluation von Designalternativen hohe Aufwände, da für jedes konkrete Simulationssystem, bei denen die CSCs variieren, an die der Gateway angeschlossen ist, eine angepasste Implementierung der Gateway-FSC benötigt wird. Im Endeffekt ergibt sich somit eine ähnliche Problematik wie bei der direkten Anbindung von FSCs an CSCs, welcher wir durch die Entwicklung der Bridges begegnet sind.

Dementsprechend liegt es nahe, zunächst auf Basis der Bridges eine Lösung für Gateways zu suchen. Wie bei der ersten Variante basiert auch die zweite auf einer von dem Entwickler entworfenen FSC für jeden Gateway. Im Gegensatz zur ersten Varianten wird die Gateway-FSC jedoch nicht direkt an die CSCs

[27] Wir gehen hier davon aus, dass ein Gateway jeweils zwei CSCs miteinander verbindet.

angebunden, sondern über Bridges. Das heißt, bei einem bidirektionalen Gateway kommen vier Bridges zum Einsatz – pro CSC jeweils eine Com2Output-Bridge und eine Input2Com-Bridge. Jede Bridge verfügt über eine eigene DBL-Spezifikation zur Beschreibung ihres Kommunikationsverhaltens. Der Vorteil dieser Lösung besteht darin, dass das Verhalten der FSC minimalistisch gestaltet werden kann, da diese lediglich Input- und OutputPorts sowie die Logik bereitstellen muss, um die extrahierten Daten aus den empfangenen Rahmen von den Com2Output-Bridges entgegenzunehmen und die weiterzuleitenden Werte an die OutputPorts anzulegen, welche mit der Input2Com-Bridge der Ziel-CSC verbunden sind. Das Kommunikationsverhalten selbst ist in den Bridges gekapselt, d.h., die Nutzung einer anderen Kommunikationstechnologie erfordert keine Anpassungen der Logik der FSC. Modifikationen der FSC können jedoch notwendig werden, wenn eine Änderung des Szenarios die Weiterleitung anderer Typen von Rahmen mit anderen Nutzdaten erfordert. In diesem Fall muss die FSC um entsprechende zusätzliche Ports sowie das entsprechende Verhalten zur Weiterleitung ergänzt werden. Ein weiterer Nachteil ist die Notwendigkeit, vier getrennte DBL-Spezifikationen pro Gateway zu verwalten und zu warten. Zudem ist der Entwickler wieder gezwungen pro Gateway eine eigene FSC zu entwickeln, wenn dies auch mit geringerem Aufwand als bei der vorherigen Variante verbunden ist.

Die dritte Variante besteht in der Verwendung einer speziell für diesen Zweck entworfenen generalisierten Gateway-Komponente. Hierbei handelt es sich um eine eigenständige konkrete Simulationskomponente (zeitgetriggert, wie die bereitgestellten CSCs), deren einzige Aufgabe die Realisierung eines bidirektionalen Gateways zwischen beliebigen CSCs ist. Das Verhalten des Gateways – in diesem Fall das Weiterleitungsverhalten – wird mit Hilfe eines speziellen DBL-Dialekts in einer einzigen DBL-Spezifikation beschrieben. Der Vorteil bei der Verwendung einer standardisierten Komponente liegt auf der Hand; der Entwickler muss weder selbst eine FSC mit der entsprechenden Funktionalität entwickeln noch diese beim Austausch einer Kommunikationstechnologie anpassen. Wie bei den Com2Output-Bridges und Input2Com-Bridges kann innerhalb der DBL-Spezifikation das Verhalten für unterschiedliche Konstellationen von instanziierten CSCs beschrieben und bei der Evaluation von Designalternativen kann die gleiche Spezifikation von verschiedenen konkreten Simulationssystemen genutzt werden. Durch den konsequenten Einsatz der vorgestellten Gateways und Bridges wird so erreicht, dass sich die konkreten Simulationssysteme zur Evaluation von Designalternativen lediglich hinsichtlich der instanziierten CSCs unterscheiden.

10.3.2 Realisierung einer generischen Simulationskomponente für Gateways

Konzeptionell ist die von uns bereitgestellte konkrete Simulationskomponente eines Gateways funktional intern ähnlich aufgebaut wie die zweite skizzierte Realisierungsvariante eines Gateways. Prinzipiell besteht der Gateway, für jede Kommunikationsrichtung, aus einer Input2Com-Bridge und einer Com2Output-Bridge für jede CSC, mit der der Gateway verbunden ist. Anstelle einer FSC sind die Bridges mit einem (schlüsselbasierten) gemeinsamen Speicher verbunden, welcher es erlaubt, beliebige extrahierte Werte aus den Nutzdaten der empfangenen Rahmen zwischenzuspeichern und für die Erzeugung neuer Rahmen wieder auszulesen. Die Interaktion mit dem Zwischenspeicher erfolgt direkt, d.h., hierfür kommen innerhalb des Gateways keine aus FERAL bekannten Links oder Ports zum Einsatz (im Gegensatz zur beschriebenen Umsetzung von Variante 2). Die Aufgabe der Com2Output-Bridges besteht in der Extraktion der Nutzdaten und deren Metainformationen aus dem Empfangsereignis. Die Input2Com-Bridges sind für die Erzeugung neuer Sendeereignisse (Rahmen inkl. Nutzdaten) für das CSC des Zielbusses verantwortlich. Beide Datenrichtungen teilen sich den gleichen Speicher und können auf diese Weise auch auf die extrahierten Informationen der anderen Kommunikationsrichtung zugreifen. Ergänzt wird dieses Konzept um einen speziellen DBL-Dialekt, der es erlaubt, das Weiterleitungsverhalten in einer kompakten und einfachen Form zu spezifizieren und dabei die Konzepte der DBL-Spezifikation von Input2Com-Bridges und Com2Output-Bridges kombiniert.

Die Abbildung 10.10 illustriert den beschriebenen konzeptionellen Aufbau in graphischer Form. Dargestellt ist der Datenfluss in beide Richtungen, wobei in diesem Beispiel `Bus1` durch eine CAN-CSC und `Bus2` von einer FR-CSC simuliert wird. An dieser Stelle sei jedoch darauf hingewiesen, dass es sich lediglich um die Darstellung des Konzeptes handelt und die Abbildung nicht die reale Implementierung widerspiegelt. Es werden nicht tatsächlich mehrere Input2Com-Bridges und Com2Output-Bridges pro Gateway instanziiert, jedoch können viele der entwickelten Basiskonzepte und Funktionalitäten der Bridges wiederverwendet werden.

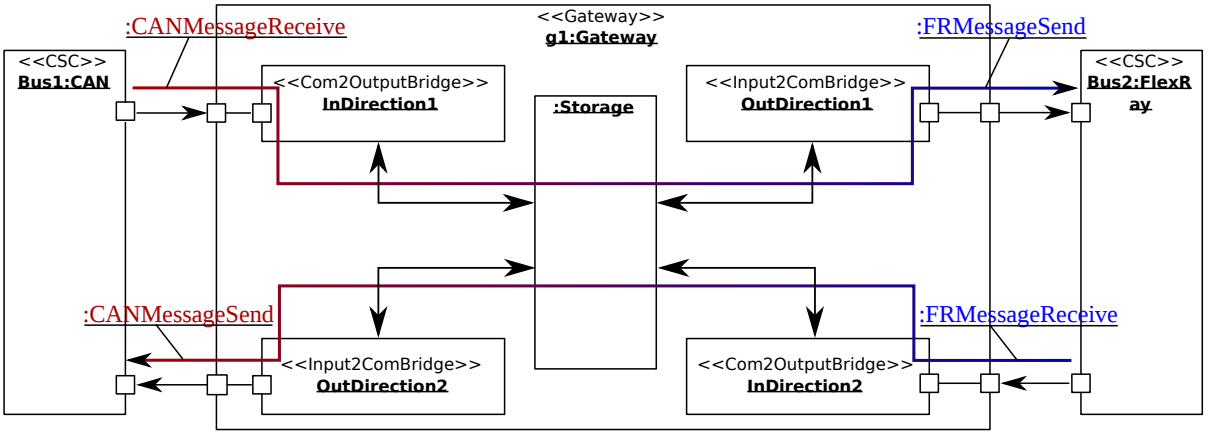


Abbildung 10.10: Konzeptionelle Struktur eines Gateways.

Das Weiterleitungsverhalten selbst wird, wie das Verhalten der Input2Com-Bridges und Com2Output-Bridges, mit Hilfe von Weiterleitungsregeln beschrieben. Sämtliche spezifizierten Weiterleitungsregeln werden für jedes anliegende `BusEvent` der Reihenfolge nach abgearbeitet. An dieser Stelle wird der konzeptionelle Aufbau der Gateways noch einmal deutlich sichtbar: Eine Weiterleitungsregel ist im Wesentlichen eine Verkettung einer Nachrichtenregel (Com2Output-Bridges) mit einer Aktionsregel (Input2Com-Bridges). Innerhalb der beiden Regelteile dürfen sämtliche Mechanismen verwendet werden, die bereits bei der Vorstellung der Konzepte der Input2Com-Bridges und Com2Output-Bridges eingeführt wurden.

Jede Weiterleitungsregel beginnt mit dem Schlüsselwort `on`, gefolgt von einem Filter, welcher sich auf ein eingehendes `BusEvent` bezieht und festlegt, ob diese Regel weiter ausgewertet werden soll. Nur wenn die innerhalb des Filters spezifizierten Kriterien durch das `BusEvent` erfüllt sind, werden die in der Regel angegebenen Aktionen ausgeführt. Hierbei können alle Filter verwendet werden, die den Vorgaben für Filter (komplexen Filterausdrücken oder eigene Implementierung von Filtern) für Com2Output-Bridge genügen (vgl. Kapitel 10.2.2).

Im Anschluss an den Filter folgt das Schlüsselwort `do` und die Spezifikation einer Nachrichtenaktion, gefolgt von dem Schlüsselwort `and`. Auch an dieser Stelle dürfen alle Nachrichtenaktionen (inkl. eigener Implementierungen) genutzt werden, die auch in Com2Output-Bridges zulässig sind. Dies gilt insbesondere für C-Aktionen, mit deren Hilfe die Nutzdaten aus dem empfangenen Rahmen extrahiert werden. Die mittels einer C-Aktion extrahierten Werte werden in dem schlüsselbasierten Zwischenspeicher abgelegt (vgl. Abbildung 10.10). Hierbei dienen die in der C-Aktion spezifizierten Namen (entweder implizit über die Metainformationen oder explizit über Renaming) der *OutputPorts* als Schlüssel für die extrahierten Werte innerhalb des Speichers. Verzichtet der Entwickler auf die explizite Vorgabe einer Nachrichtenaktion innerhalb der Weiterleitungsregel, wird automatisch eine C-Aktion (ohne Parameter) ausgeführt. Soll keine Nachrichtenaktion ausgeführt werden, muss anstelle der Nachrichtenaktion das Schlüsselwort `NoAction` verwendet werden (z. B. aufgrund der Nutzung des `OnSyncEventConsume`-Filters).

Der nächste Teil der Weiterleitungsregel ist optional und erlaubt die Angabe eines *Triggers* unter Ver-

wendung des Schlüsselwortes `iff`. Dieser muss den Vorgaben an Trigger für Aktionsregeln entsprechen (vgl. Kapitel 10.2.1). Der Bezug auf InputPorts, wie bei den Triggern der Input2Com-Bridges, ist in diesem Fall nicht möglich, stattdessen bezieht sich der Trigger auf die Werte innerhalb des Zwischenspeichers. Die bei den Triggern der Input2Com-Bridges verwendeten Bezeichner zur Adressierung der InputPorts werden als Schlüssel für den Zugriff auf die Werte des Zwischenspeichers interpretiert. Falls ein Trigger definiert ist, werden die nachfolgenden Aktionen der Weiterleitungsregel nur ausgeführt, wenn dieser erfüllt ist. Den Abschluss einer Weiterleitungsregel bildet das Schlüsselwort `do`, gefolgt von einer Aktion – gemäß den Vorgaben für Aktionen in Input2Com-Bridges (vgl. Kapitel 10.2.1). Diese können z. B. dazu verwendet werden, ein neues Sendeereignis (inkl. Rahmen sowie Nutzdaten) zu erzeugen oder auch neue Werte in den Zwischenspeicher zu schreiben (`SampleValue`-Aktion).

Bei den Input2Com-Bridges bzw. Com2Output-Bridges konnte die Gültigkeit der Regeln oder ganzer Blöcke von Regeln auf bestimmte CSCs beschränkt werden. Eine solche Einschränkung muss jedoch bei Gateways nicht eindeutig sein, da auch Gateways zwischen Bussen mit identischer Kommunikationstechnologie eingesetzt werden. Daher werden für Weiterleitungsregeln nicht nur die Typen der CSC festgelegt[28], für die diese gültig sind, sondern auch der Name der CSC, auf dem das `BusEvent` empfangen wird, sowie Name und Typ der CSC, an den dieses weitergeleitet werden soll. Das Listing 10.14 zeigt ein Beispiel für die Festlegung des Weiterleitungsverhaltens eines Gateways. In diesem Szenario empfangen wir von `Bus2` eine aggregierte Statusinformation bzgl. Geschwindigkeit und zurückgelegter Entfernung, kodiert in einem einzelnen FlexRay-Rahmen (Länge der Payload 16 Bytes). Diese Informationen sollen in zwei CAN-Rahmen aufgespalten und über `Bus1` weitergeleitet werden: Jeweils ein Rahmen für die Geschwindigkeit und einer für die zurückgelegte Entfernung (je 8 Byte Nutzdaten). Für die generierten CAN-Rahmen kommen die CAN-Identifier 10 bzw. 16 zum Einsatz. Wie bei CAN üblich, soll eine Übertragung nur erfolgen, sofern sich die Werte gegenüber der letzten Übertragung verändert haben (ereignisgetriggerter Ansatz). Zusätzlich ist ein *Renaming* erforderlich, da die FSCs (Sender und Empfänger) unterschiedliche Bezeichner für ihre Input- und OutputPorts verwenden. Für das *Renaming* ist die C-Aktion in den Zeilen 1-3 verantwortlich[29]. Die Zeilen 5-17 legen den Aufbau der beiden zu sendenden CAN-Rahmen (Nutzdaten und CAN-Identifier) fest. Die Weiterleitungsregeln sind innerhalb des in Zeile 20 beginnenden Blockes definiert. Zunächst schränken wir die Anwendbarkeit der Regeln auf das vorliegende Szenario ein (Zeile 20), dann spezifizieren wir das Weiterleitungsverhalten. Die Regel in Zeile 21 ist, wie folgt zu lesen: Sobald ein Empfangsereignis auftritt, welches einen Rahmen innerhalb des *Message Receive Buffers 1* des simulierten Kommunikationscontrollers hinterlegt, führe die C-Aktion `InComingCarStatus` aus[30]. Hat sich der mit dem Schlüssel `CSpeed` assoziierte Wert verändert, erstelle ein Sendeereignis gemäß den Vorgaben der Messagedefinition `CarSpeedMessage`. Die Weiterleitungsregel in Zeile 22 beschreibt das gleiche Verhalten für die Entfernungsangabe.

In der Gegenrichtung wollen wir die Statusinformationen zweier CAN-Rahmen mit den Identifiern 55 und 65 zu einem FlexRay-Rahmen aggregieren. In diesem Fall ist kein *Renaming* erforderlich, daher verwenden wir die implizite Variante der C-Aktion. Die Zeilen 26-31 definieren den Aufbau des entsprechenden FlexRay-Rahmens; die Weiterleitungsregeln in den Zeilen 34 und 35 erzeugen das entsprechende Sendeereignis, wobei der FlexRay-Rahmen jeweils die aktuellen Werte des Zwischenspeichers verwendet. Der Trigger `IsValid` sorgt dafür, dass nur dann ein Sendeereignis erzeugt wird, sofern sowohl für `MotorStatus` als auch `BrakeStatus` mindestens einmal ein Statuswert empfangen und im Zwischenspeicher hinterlegt wurde.

Darüber hinaus erlaubt auch dieser DBL-Dialekt die Gruppierung von Regeln zu exklusiven Gruppen über das Schlüsselwort `exclusive`, die Angabe von Totzeiten (`block for`) sowie die Festlegung von Kosten für die Abarbeitung der Weiterleitungsregeln (vgl. Kapitel 10.2.2).

[28] Hierbei können in einer Liste auch mehrere zulässige Typen von CSCs aufgelistet werden.
[29] Natürlich wären an dieser Stelle auch komplexere Extraktionsregeln, wie in Kapitel 10.2.2 beschrieben, zulässig.
[30] Diese aktualisiert die Werte mit den Schlüsseln `CSpeed` und `CDistance` innerhalb des Zwischenspeichers.

Listing 10.14: Beschreibung des Weiterleitungsverhaltens eines Gateways.

```
InComingCarStatus = Consume(
  "Speed" -> "CSpeed",
  "Distance" -> "CDistance");

CarSpeedMessage = MessageDef(
      Generic{
            ConcatPayload("CSpeed");
      } CAN {
        BaseID -> 10;
      });

CarDistanceMessage = MessageDef(
      Generic{
            ConcatPayload("CDistance");
      } CAN {
        BaseID -> 16;
      });


"Bus2"(FlexRay)->"Bus1"(CAN){
  on MessageFilter(FlexRay{MessageBuffer -> 1}) do InComingCarStatus and iff
    OnChange("CSpeed") do SendMessage(CarSpeedMessage);
  on MessageFilter(FlexRay{MessageBuffer -> 1}) do InComingCarStatus and iff
    OnChange("CDistance") do SendMessage(CarDistanceMessage);
}


CarStatus = MessageDef(
      Generic{
            ConcatPayload("MotorStatus, BrakeStatus")
      } FlexRay {
        MessageBuffer -> 2;
      });

"Bus1"(CAN)->"Bus2"(FlexRay){
  on MessageFilter(CAN{BaseID->55}) iff IsValid("MotorStatus") & IsValid("BrakeStatus") do
    SendMessage(CarStatus);
  on MessageFilter(CAN{BaseID->65}) iff IsValid("MotorStatus") & IsValid("BrakeStatus") do
    SendMessage(CarStatus);
}
```

10.4 Simulation von zusätzlichem Datenverkehr

Eine in der Praxis häufig anzutreffende Aufgabenstellung besteht in der Entwicklung einer verteilten Funktionalität, die jedoch selbst wiederum nur ein Subsystem innerhalb eines übergeordneten komplexeren verteilten Gesamtsystems ist (z. B. die Entwicklung eines Abstandsassistenten innerhalb eines PKWs). In diesem Fall steht das Kommunikationsmedium im Allgemeinen nicht exklusiv dem entwickelten Subsystem zur Verfügung, sondern wird auch von anderen Teilnehmern (Subsystemen) genutzt. Hieraus ergeben sich häufig Einschränkungen in Bezug auf die noch verfügbare Bandbreite und die bereits stattfindende Kommunikation kann zusätzlich für Verzögerungen bei der Nachrichtenübertragung sorgen. Beide Aspekte gilt es bei der Entwicklung und Evaluation zu berücksichtigen.

Im Idealfall kann das gesamte System, inklusive des neu entwickelten Subsystems, mittels FERAL simuliert werden. Auf diese Weise können das komplexe Zusammenwirken aller Subsysteme sowie die Auswirkungen eines zusätzlichen Subsystems auf die Auslastung des Mediums und Wechselwirkungen durch höhere Verzögerungen bei der Kommunikation auf das Gesamtverhalten genau untersucht werden. Ist dies nicht praktikabel, da z. B. die anderen Subsysteme des Gesamtsystems (noch) nicht verfügbar oder mit FERAL simulierbar sind, so sollten dennoch für eine realistische Evaluation der neuen Teilfunktionali-

tät die Auswirkungen der zusätzlichen Kommunikationen durch die Komponenten (Buslast, resultierende Verzögerungen) der anderen Subsysteme berücksichtigt werden. Hierzu wird eine einfache Lösung benötigt, um das Kommunikationsverhalten der im Simulationssystem nicht abgebildeten Komponenten möglichst realistisch und aufwandsarm nachzubilden.

Eine Variante wäre der Einsatz von speziell entworfenen nativen Simulationskomponenten, die das Kommunikationsverhalten jedes Subsystems oder sogar jeder Komponente der Subsysteme nachbilden. Dies ist jedoch entsprechend aufwändig. Eine Alternative stellt die Verwendung der von uns bereits vorgestellten Input2Com-Bridges und deren DBL-Spezifikationen dar. Diese können genutzt werden, um das Kommunikationsverhalten und vor allem das Kommunikationsaufkommen auf abstrakte Art und Weise mit moderatem Aufwand nachzubilden. Voraussetzung hierfür ist zunächst eine Analyse der Nachrichtenströme, die nachgebildet werden sollen, d.h., welche Nachrichten werden versendet und wie ist der Nachrichtenstrom charakterisiert (periodische Nachrichten, sporadische Nachrichten, Größe der Payload, Zwischenankunftsintervalle usw.).

Mit Hilfe dedizierter[31] Input2Com-Bridges (diese sind nur mit einer CSC verbunden, aber nicht mit einer FSC) kann das charakteristische Kommunikationsverhalten nachgebildet werden. Das Kommunikationsverhalten wird mit Hilfe von Aktionsregeln in Form einer DBL-Spezifikation beschrieben: Die Verwendung stochastischer Trigger (`StochasticTrigger`, Abbildung 10.7) ermöglicht die Erzeugung von Sendeereignissen mit einer vorgegebenen Wahrscheinlichkeit, sodass sich hierüber sporadische Nachrichten nachbilden lassen. Periodische Nachrichten lassen sich über die periodischen Trigger realisieren. Logische Ausdrücke erlauben es, diese auch mit anderen, z. B. stochastischen Triggern, zu kombinieren, um so komplexere pseudozufällige Charakteristika oder Kommunikationsmuster nachzustellen. Die Kombination mehrerer Aktionsregeln, unter Nutzung exklusiver Gruppen und Totzeiten, bietet zusätzliche Optionen, um auch komplexe Kommunikationsmuster umzusetzen. Hinzu kommt die Möglichkeit, über die Implementierung eigener Trigger beliebiges Auslöseverhalten zu realisieren, um so z. B. auch aufgezeichnete Kommunikationsströme in einem bestimmten Szenario wiederzugeben.

Da eine solche dedizierte Input2Com-Bridges mit keiner FSC verbunden ist, kann für die Erstellung der Nutzdaten der Rahmen ein spezieller `RandomPayload`-Builder (vgl. Abbildung 10.7) verwendet werden. Dieser erzeugt zufällige Byte-Sequenzen einer vorgegebenen Länge, welche als Nutzdaten der Rahmen verwendet werden, um eine realistische Auslastung des Busses zu simulieren. Alternativ könnte man natürlich auch, über einen eigenen Payload-Builder, eine zuvor aufgezeichnete Sequenz von Nutzdaten übertragen.

Das Listing 10.15 zeigt ein entsprechendes Beispiel, in dem die Bridge als Lastgenerator zum Einsatz kommt. In der DBL-Spezifikation werden drei CAN-Nachrichten *M1*, *M2* und *M3* mit 2, 6 bzw. 8 Bytes Nutzdaten (Zeile 1-20) definiert. Die zufällig generierten Nutzdaten werden eindeutigen Bezeichnern zugeordnet, sodass auch entsprechende Metainformationen mit dem Sendeereignis generiert werden können. Die Zahl in den eckigen Klammern gibt die Anzahl der zu generierenden Bytes an.

Die Aktionsregel in Zeile 22 sorgt dafür, dass alle 750 ms ein Sendeereignis gemäß der Messagedefinition *M1* generiert wird. Ein Sendeereignis entsprechend *M2* wird in jedem Zeitschritt mit einer Wahrscheinlichkeit von 0,028 – unter Verwendung einer uniformen Verteilung – erzeugt. Nach einem erzeugten Sendeereignis wird für mindestens 500 ms (Zeile 23) kein weiteres Sendeereignis mehr für *M2* generiert, d.h., das minimale Zwischenankunftsintervall beträgt 500 ms. Nachrichten der Messagedefinition *M3* liegt ein festes Intervall von 10 ms zu Grunde, allerdings sorgt die konjunktive Verknüpfung mit dem stochastischen Trigger (Zeile 24) dafür, dass im Schnitt nur alle 40 ms (d.h. jedes vierte Intervall) ein Sendeereignis erzeugt wird.

[31] Je nach Komplexität des Kommunikationsmodells genügt eine einzige Bridge oder man ersetzt jede nicht in der Simulation vorhandene Komponente durch eine eigene Bridge mit eigener DBL-Spezifikation.

Listing 10.15: Verwendung einer DBL-Spezifikation zur Erzeugung von zusätzlichem Datenverkehr.

```
M1 = MessageDef(
  Generic{
    RandomPayload("val1"[2]);
  } CAN{
    BaseID -> 70;
  });

M2 = MessageDef(
  Generic{
    RandomPayload("val1"[4], "val2"[2]);
  } CAN{
    BaseID -> 75;
  });

M3 = MessageDef(
  Generic{
    RandomPayload("tmp"[8]);
  } CAN{
    BaseID -> 90;
  });

on PeriodicTrigger(interval->0:750000000) do SendMessage(M1);
on StochasticTrigger(seed->16, prob->0.028) do SendMessage(M2) block for (0:500000000);
on (StochasticTrigger(seed->10, prob->0.25) & PeriodicTrigger(interval->0:10000000)) do
  SendMessage(M3);
```

10.5 Konventionen für die Konfiguration von CSCs, Bridges und Gateways

Wie in Kapitel 10.1.3 beschrieben, wird jedes konkrete Simulationssystem für FERAL in Form eines eigenständigen OSGi-Bundle umgesetzt. Die einzelnen FSCs des Simulationssystems werden als Fragment-Bundles realisiert. Diese enthalten alle Artefakte, welche für die Simulation des jeweiligen FSC benötigt werden. Das Bundle für das konkrete Simulationssystem enthält die Szenariobeschreibung, hierbei handelt es sich um Java-Code (Glue-Code), welcher die Schnittstellen von FERAL nutzt, um das konkrete Simulationssystem in einer von FERAL ausführbaren Form zu beschreiben. Hierzu gehört die Instanziierung und das Anlegen des Objektgeflechts, bestehend aus den verschiedenen FSCs, CSCs und Links, aus denen das Simulationssystem aufgebaut ist.

Damit das Simulationssystem ausführbar ist, werden noch die Konfiguration der CSCs sowie die DBL-Spezifikationen zur Beschreibung des Kommunikationsverhaltens der Bridges und Gateways benötigt. In Bezug auf die CSCs besteht die Möglichkeit, die in Kapitel 9 vorgestellten Klassen für die Konfiguration der CSCs direkt als Teil der Szenarienbeschreibung zu instanziieren und mit Werten zu belegen. Dieser Ansatz funktioniert, erfordert jedoch sehr viel Java-Glue-Code, sodass sich Wartung und Anpassungen aufwändig gestalten. Daher kann der Entwickler die Konfigurationen der CSCs ebenso wie die DBL-Spezifikationen in separate Dateien auslagern. Sowohl bezüglich der Ordnerstruktur innerhalb des Bundle als auch der Namensgebung der Konfigurationsdateien gibt es explizite Konventionen, sodass die Konfigurationsdateien der einzelnen Komponenten automatisiert geladen werden können. Hierdurch reduziert sich der Aufwand für den Entwickler ebenso wie die Menge des benötigten Glue-Codes für die Szenariobeschreibung. Dies vereinfacht sowohl die Entwicklung als auch die Verwaltung eines Simulationssystems erheblich.

Die Konfigurationen der CSCs erfolgt mittels XML-Dateien. Neben separaten XML-Dateien für jede CSC, die der Konfiguration der Medieneigenschaften dienen (Mediumkonfiguration), gibt es individuelle XML-Dateien für die einzelnen PCUs der CSCs. Die XML-Dateien werden zur Laufzeit unter Ver-

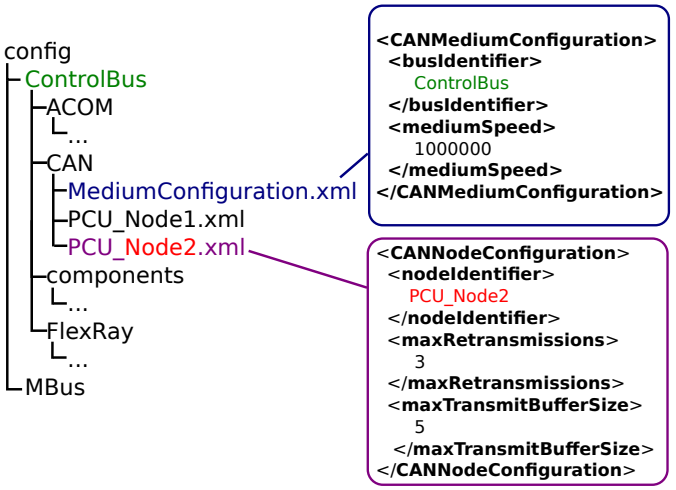


Abbildung 10.11: Übersicht über den Aufbau der Konfiguration (Mediumkonfiguration und Konfiguration einer PCU) eines konkreten Simulationssystems.

wendung von XStream[32] eingelesen und in Form von Instanzen der jeweiligen Konfigurationsklassen (vgl. Kapitel 9) den CSCs bereitgestellt. Sämtliche Konfigurationsdateien sind Bestandteil des Bundle und werden innerhalb des Ordners `config` abgelegt. Wird in der Szenarienbeschreibung eine CAN-CSC mit dem Bezeichner `ControlBus` angelegt, so werden alle Konfigurationsdateien, die sich auf diese CSC beziehen, unterhalb des Ordners `config/ControlBus` erwartet. Da die Erzeugung der CSCs mit Hilfe des Abstrakt Factory-Pattern erfolgt, kann ein Entwickler sehr einfach eine CSC durch eine CSC austauschen, welche eine andere Kommunikationstechnologie simuliert. Hierzu muss er lediglich innerhalb der Szenariobeschreibung eine andere *konkrete Factory* instanziieren – sofern das zugrunde liegende abstrakte Simulationssystem konsequent Bridges und Gateways einsetzt. In solchen Fällen wollen wir dem Entwickler nicht zumuten, für die Evaluation verschiedener Kommunikationstechnologien separate Bundles zu erstellen. Daher können für einen Bus unterschiedliche Konfigurationen für verschiedene CSC-Typen hinterlegt werden. Die Unterordner richten sich nach dem Namen der CSC; für die CAN-CSC liegen die Konfigurationsdateien für den `ControlBus` in dem Unterordner `config/ControlBus/CAN`. In dem Ordner `config/ControlBus/FlexRay` können Konfigurationsdateien für die Simulation des `ControlBus` mit der FR-CSC abgelegt werden. Hilfsfunktionen sorgen dafür, dass in Abhängigkeit von der instanziierten CSC die korrekten Konfigurationsdateien geladen werden.

In einem Konfigurationsordner für eine CSC werden die Dateien für die Mediumkonfiguration sowie die Konfigurationen der PCUs erwartet. Die Mediumkonfiguration ist in einer Datei mit der Bezeichnung `MediumConfiguration.xml` zu finden, die PCUs jeweils in einer Datei mit dem Namen `PCU_FSCName.xml`. Hierbei ist `FSCName` der eindeutige Bezeichner der FSC innerhalb des Simulationssystems, deren Busanbindung die PCU simuliert. Anhand dieser Namenskonventionen können die einzelnen Simulationskomponenten bzw. deren Ports automatisch über Links miteinander verbunden werden (vgl. Kapitel 9.3).

Die Abbildung 10.11 zeigt die Konfiguration sowie die Ordnerstruktur für das skizzierte Szenario. An den `ControlBus` sind die beiden FSCs `Node1` und `Node2` angeschlossen. Zusätzlich enthält das konkrete Simulationssystem noch eine weitere CSC, mit der Bezeichnung `MBus`, auf die wir jedoch im Folgenden nicht weiter eingehen.

Neben den Konfigurationen der CSCs werden ebenfalls die DBL-Spezifikationen benötigt. Diese werden in dem Unterordner `components` des Ordners mit der Konfiguration für den jeweiligen Bus abgelegt.

[32] http://xstream.codehaus.org/

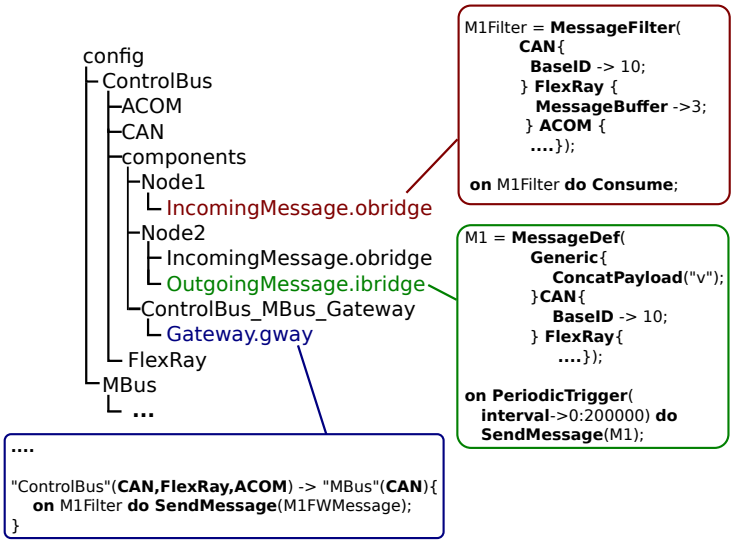


Abbildung 10.12: Ordnerstruktur für die DBL-Spezifikationen.

Um wiederum eine automatische Zuordnung zwischen einem FSC und dessen DBL-Spezifikation zu ermöglichen, existiert für jede FSC, welche über eine Bridge mit dem Bus verbunden ist, ein eigener Unterordner innerhalb des `components`-Verzeichnisses. Dieser trägt den gleichen Namen wie die FSC. Sofern die FSC eine Input2Com-Bridge verwendet, befindet sich in diesem Verzeichnis eine Datei mit der Bezeichnung `OutgoingMessages.ibridge` mit der DBL-Spezifikation des Kommunikationsverhaltens der FSC. Wird stattdessen (oder zusätzlich) eine Com2Output-Bridge genutzt, ist dort eine Datei mit dem Namen `IncomingMessages.obridge` mit der DBL-Spezifikation des Empfangsverhaltens hinterlegt. Anhand der Ordnerstruktur und den Namenskonventionen können die von einer FSC verwendeten Bridges automatisch instanziiert sowie die notwendigen Links zwischen FSCs und Bridges sowie Bridges und CSCs erstellt werden.

In dem vorherigen Szenario ist `Node2` über eine Input2Com-Bridge sowie eine Com2Output-Bridge mit dem `ControlBus` verbunden, während `Node1` lediglich über eine Com2Output-Bridge eingehende Nachrichten empfängt. Die Abbildung 10.12 zeigt die entsprechende Ordnerstruktur in Ergänzung zu Abbildung 10.11. Die beiden Busse `ControlBus` und `MBus` sind über einen Gateway verbunden, dessen Weiterleitungsverhalten ebenfalls durch eine DBL-Spezifikation beschrieben wird. Diese wird in einem der `components`-Unterordner der beiden Busse, die durch den Gateway verbunden werden, in der Datei `Gateway.gway` gespeichert. Diese liegt in einem Ordner mit dem Namen `NameBus1_NameBus2_Gateway`. Auch hier erlaubt die Namenskonvention die automatische Instanziierung und Konfiguration des Gateways. Zusätzlich wird für den Gateway ebenfalls eine entsprechende PCU-Konfiguration für jeden der beiden Busse benötigt. Die Konfigurationsdatei muss den Namen `PCU_ControlBus_MBus_Gateway.xml` tragen und für beide Busse `MBus` und `ControlBus` innerhalb der CSC-spezifischen Unterordner bereitgestellt[33] werden.

[33] In der Abbildung 10.11 haben wir auf eine entsprechende Darstellung verzichtet.

11 Anwendung von FERAL

Um sowohl FERAL als auch die von uns entwickelten und integrierten Simulatoren (CAN, FlexRay, ACOM, ns-3) und die vorgestellten Konzepte für den Austausch von Kommunikationstechnologien zu evaluieren, dienen uns zwei komplexe Anwendungsszenarien: Ein vereinfachtes Modell eines verteilt realisierten Anti-Blockier-Systems (ABS) sowie ein Adaptive Cruise Control System (ACC) mit integrierten Abstands- und Bremsassistenten, welches ebenfalls auf einer verteilten Architektur basiert.

Die Ergebnisse dieses Kapitel wurden in [4, 5, 7, 10] veröffentlicht.

11.1 Entwicklung eines vereinfachten Anti-Blockier-Systems mit FERAL

Das von uns für die Evaluation von FERAL entwickelte ABS besteht aus einem zentralen Steuergerät, jeweils einem Sensor pro Rad (für die Erfassung der Drehzahl) und einem in das zentrale Hydro-Aggregat integrierten Steuergerät. Die Sensoren übermitteln die aktuelle Drehzahl der Räder über einen gemeinsamen Kommunikationsbus an das zentrale Steuergerät. Dieses ermittelt anhand eines physikalischen Modells des Autos die erforderlichen Bremskräfte und sendet diese über den Kommunikationsbus an das Steuergerät des Hydro-Aggregats, welches den Bremsdruck der einzelnen Räder einstellt. Die Bremswirkung führt dazu, dass sich die Rotationsgeschwindigkeit der einzelnen Räder verändert, sodass über die Sensoren für die Drehzahlmessung ein geschlossener verteilter Regelkreis, bestehend aus ABS-Steuergerät, Hydro-Aggregat und Drehzahlsensoren, gebildet wird.

Sämtliche Komponenten des verteilten ABS wurden unter Verwendung von Matlab Simulink spezifiziert und basieren funktional im Wesentlichen auf dem von MathWorks für Simulink bereitgestellten Beispiel eines *Anti-Lock Braking Systems*. Aufgabe des ABS ist es, den Schlupf des Rades auf ein Minimum zu regeln, bei gleichzeitig maximal erreichbarer Verzögerung. In unserem Simulationssystem existiert für jedes Rad ein entsprechendes physikalisches Modell, welches den Schlupf auf Grundlage der aktuellen Drehzahl, des Bremsdruckes und der Reibung sowie dessen Auswirkungen auf die Drehzahl ermittelt. Innerhalb des Simulationssystems ist dieses Modell, zur Vereinfachung des Aufbaus, Bestandteil des Sensors, da diese die aktuelle Drehzahl zurückliefern müssen.

Das hier vorgestellte ABS wurde stark vereinfacht, sowohl in Bezug auf das physikalische Modell der Regelstrecke als auch auf Wirkungsweise und Struktur. Da jedoch der Schwerpunkt hier nicht auf der Entwicklung eines ABS abzielt, sondern auf den Test und die Evaluation von FERAL sowie dessen integrierten Simulatoren, stellt dies keine Einschränkung dar. Zentrales ABS-Steuergerät, Hydro-Aggregat sowie die Drehzahlsensoren werden in FERAL durch FSCs auf Basis von Matlab Simulink realisiert. Diese kommunizieren über einen zentralen Bus. Als Designalternativen für den Bus haben wir CAN und FlexRay als Kommunikationstechnologie evaluiert. Die FSCs sind über Bridges an die CSC angebunden, um diese Evaluation zu erleichtern. In dem Szenario erfolgt die Kommunikation, unabhängig von der Realisierung des Busses, streng periodisch mit einem Zwischenankunftsintervall von 5 ms.

Um die Auswirkungen unterschiedlicher Übertragungsverzögerungen zu evaluieren, wurden weitere Simulationen mit unterschiedlich hoher Busauslastung durchgeführt. Für die Simulation von zusätzlichem Traffic auf dem Bus kamen die in Kapitel 10.4 beschriebenen Ansätze zum Einsatz. Das ABS dient als Grundlage, um Effekte der Kommunikation (Ausfälle von Rahmen, Verzögerungen bei der Übertragung bei überlastetem Bus) und deren Auswirkungen auf ein komplexes verteiltes System zu studieren, wobei hier

nicht primär das ABS-Szenario, sondern die Fähigkeiten und Eignung von FERAL für die Entwicklung verteilter Anwendungen im Mittelpunkt stand. So wurden in diesem Rahmen Werkzeuge für die interaktive Evaluation und Diagnose, die automatische Prüfung von zuvor definierten Constraints sowie Mechanismen zur Protokollierung und anschließenden Auswertung von Simulationsläufen realisiert. Die abschließende interaktive Evaluation des ABS-Szenarios, mit den von FERAL bereitgestellten Werkzeugen, haben wir im Rahmen des Audits des Innovationszentrums 2012 präsentiert (vgl. Kapitel 9.6). Ein entsprechender Screencast kann unter http://agvs.cs.uni-kl.de/FERAL/ abgerufen werden. In [BGFK13] wurden anhand eines anderen Szenarios, die Auswirkungen von Verzögerungen auf die Funktionalität des ABS studiert. Hierbei konnten wir unter anderem zeigen, dass sich zu hohe Verzögerungen oder auch Verluste von Nachrichten negativ auf die Regelgüte auswirken und im Extremfall zu einem Versagen des Systems führen können.

11.2 Entwicklung eines Adaptive Cruise Control Systems

Um die Integration und Nutzung unterschiedlicher Simulatoren (für FSCs und CSCs) innerhalb eines Simulationssystems zu evaluieren, entwickelten wir für unsere Publikationen [5] und [10] ein komplexeres Anwendungsszenario, welches sämtliche in FERAL integrierten Simulatoren verwendet. Hierbei handelt es sich um ein verteiltes Adaptive Cruise Control (ACC) System mit integriertem Abstands- und Bremsassistenten, dessen Entwicklung vollständig mit FERAL erfolgte. Unsere Evaluationen konzentrieren sich bei diesem Szenario primär auf die Untersuchung von Designalternativen hinsichtlich der verwendeten Kommunikationstechnologien. Wir wollten vor allem untersuchen, inwiefern die Verwendung von Bridges und Gateways in einem realen Szenario den Austausch einer Kommunikationstechnologie unterstützt und inwiefern unterschiedliche Simulatoren mittels FERAL innerhalb eines Szenarios sinnvoll zusammen genutzt werden können. Zusätzlich ging es darum zu untersuchen, wie gut sich FERAL für das Virtual Prototyping komplexer Systeme und Umgebungen eignet.

Das von uns entwickelte ACC System erfüllt primär die Aufgabenstellung eines einfachen Tempomats: Das Fahrzeug wird auf einer bestimmten vorgegebenen Geschwindigkeit gehalten, wobei der Tempomat den Motor so regelt, dass auftretende externe Störgrößen wie Wind- und Rollwiderstand sowie Einflüsse durch Steigung und Gefälle der Fahrstrecke kompensiert werden. Zusätzlich zu den Funktionen eines normalen Tempomats ist unser ACC System mit einem Abstands- und Bremsassistenten ausgestattet. Dessen Aufgabe besteht in der Wahrung eines minimalen Sicherheitsabstandes zu vorausfahrenden Autos oder Hindernissen. Bei Unterschreiten dieses minimalen Abstandes wird das Auto automatisch verlangsamt oder eine Gefahrenbremsung initiiert, um eine Kollision zu vermeiden. Um diese Funktionalität umzusetzen, verfügt das simulierte Auto über einen Radarsensor, um die Distanz zu einem vorausfahrenden Fahrzeug oder einem Hindernis vor dem Auto zu ermitteln. Gleichzeitig ermittelt der Radarsensor die Geschwindigkeit des Hindernisses. Der minimale Abstand zu einem vorausfahrenden Fahrzeug wird auf Basis der aktuellen Geschwindigkeit des eigenen Fahrzeugs bestimmt und entspricht den Vorgaben der StVO[1].

Bei dem von uns simulierten Anwendungsszenario fährt ein Auto auf einer hügeligen Strecke mit eingeschaltetem ACC mit einer vorgegebenen Geschwindigkeit von 140 km/h, nachdem es zunächst von 50 km/h auf diese Zielgeschwindigkeit beschleunigte. Unser Modell der Regelstrecke berücksichtigt als Störgrößen den Luftwiderstand, welcher vereinfacht als proportional zur Geschwindigkeit angenommen wird, sowie die aktuelle Steigung. Beide Störgrößen beeinflussen die Geschwindigkeit des Autos und werden durch das ACC System kompensiert. Das Höhenprofil der insgesamt $1,7$ km langen Fahrstrecke ist in Abbildung 11.1 dargestellt. Auf der Teststrecke befindet sich ein weiteres Fahrzeug, welches langsamer als unser Testfahrzeug unterwegs ist (100 km/h). Im Laufe der Fahrt ist das ACC System dazu gezwungen, die vorgegebene Soll-Geschwindigkeit von 140 km/h zu reduzieren und sich an das Tempo

[1] Ist v die Geschwindigkeit des Fahrzeuges in $\frac{km}{h}$ so beträgt der minimale Sicherheitsabstand $d = 0,5\,\frac{h \cdot m}{km} \cdot v$.

des vorausfahrenden Fahrzeuges anzupassen, um ein Unterschreiten des Minimalabstandes zu vermeiden. Nach annähernd 1,7 km bremst das vorausfahrende Fahrzeug abrupt ab, sodass das ACC System eine Notbremsung einleiten muss, um die Kollision mit dem nun stationären Hindernis zu verhindern.

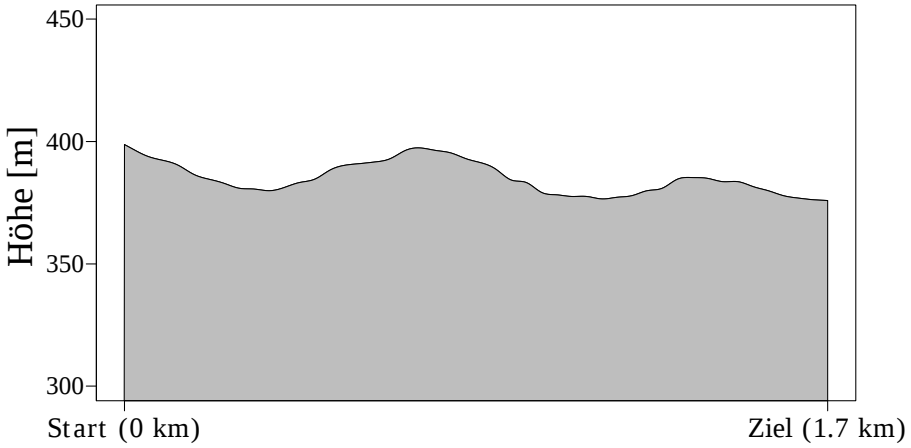


Abbildung 11.1: Höhenprofil der ausgewählten Fahrstrecke.

Im ersten Schritt gehen wir im Detail auf den Aufbau des Simulationssystems unseres verteilten ACC Systems ein, indem wir das entwickelte abstrakte Simulationssystem vorstellen (Kapitel 11.2.1). Anhand dieses abstrakten Simulationssystems werden mehrere konkrete Simulationssysteme abgeleitet, welche unterschiedliche Kommunikationstechnologien für den zentralen Bus verwenden, über den die funktionalen Komponenten des ACC Systems interagieren. Innerhalb der konkreten Simulationssysteme werden für die konkreten Simulationskomponenten die Simulatoren und Verhaltensmodelle sowie die Konfigurationen der CSCs festgelegt (Kapitel 11.2.2). Wir konzentrieren uns, wie bereits beschrieben, auf den Kommunikationsaspekt und evaluieren das Szenario für unterschiedliche Designalternativen bzgl. der einsetzbaren Kommunikationstechnologien. Die Ergebnisse dieser Simulationen werden dann in Kapitel 11.2.3 analysiert und erläutert. Zu den betrachteten Kriterien gehören hierbei sowohl die Regelgüte als auch die auftretenden Übertragungsverzögerungen und deren Charakteristika bei den unterschiedlichen Kommunikationstechnologien.

11.2.1 Aufbau des abstrakten Simulationssystems

Die Abbildung 11.2 zeigt den Aufbau des abstrakten Simulationssystems für unser ACC System. Wir widmen uns nun kurz der Beschreibung der einzelnen Komponenten sowie deren Aufgabenstellung. Zur besseren Unterscheidung der verschiedenen Arten von Simulationskomponenten sind die CSCs in der Abbildung 11.2 grau eingefärbt, während die FSCs, Bridges und Gateways in weiß gehalten sind.

Der Kontrollalgorithmus unseres ACC Systems wird durch die Komponente *ACC-Controller* implementiert. Dieser ist dafür verantwortlich, die Geschwindigkeit des Fahrzeuges entsprechend den Vorgaben des Fahrers sowie der aktuell gemessenen Geschwindigkeit zu regeln. Hierbei muss gleichzeitig der Abstand zu vorausfahrenden Fahrzeugen oder Hindernissen überwacht und die Einhaltung eines Mindestabstandes sichergestellt werden. Befindet sich kein Hindernis oder vorausfahrendes Fahrzeug vor dem kontrollierten Fahrzeug, wird ein einfacher Proportional-Integral-Derivative (PID) Regler verwendet, um den Motor so anzusteuern, dass die Abweichung zwischen Ist- und Soll-Geschwindigkeit minimiert und Einflüsse durch Störgrößen, wie den Luftwiderstand sowie Steigungen und Gefälle, ausgeglichen werden. Der PID-Algorithmus ist hierbei als untergeordneter Regler zu sehen, der als Eingabe die aktuelle Geschwindigkeit des Fahrzeugs sowie die aktuelle Soll-Geschwindigkeit erhält. Ein übergeordneter

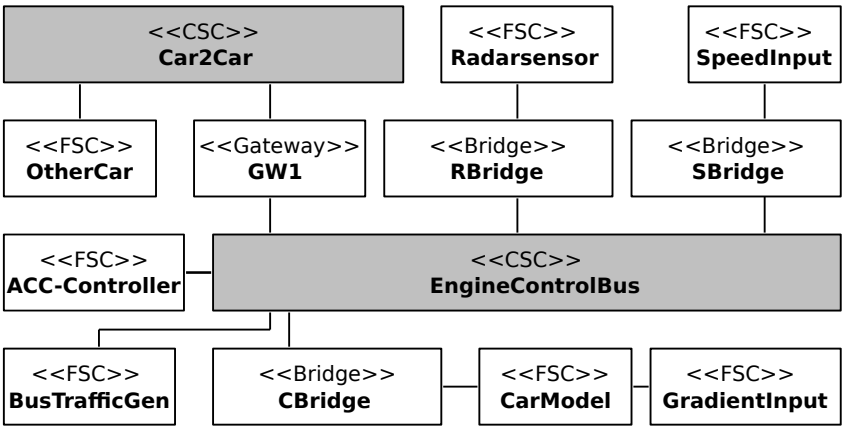


Abbildung 11.2: Abstraktes Simulationssystem des ACC Systems.

Zustandsautomat kontrolliert und überwacht den Abstand zu vorausfahrenden Fahrzeugen oder Hindernissen und passt, falls erforderlich, die dem PID-Algorithmus vorgegebene Soll-Geschwindigkeit an, sodass der minimale Sicherheitsabstand eingehalten wird. Hierbei wird die Vorgabe des Fahrers aktiv überschrieben, um Unfälle sowie die Unterschreitung des Sicherheitsabstandes zu vermeiden. Um abrupte Bremsmanöver zu vermeiden, wird beim Annähern an ein langsameres Fahrzeug die Geschwindigkeit des eigenen Fahrzeugs allmählich reduziert. Der *ACC-Controller* regelt den Motor hierbei so, dass beim Erreichen des Mindestabstands beide Fahrzeuge die gleiche Geschwindigkeit innehaben. Nimmt der Abstand zu dem vorausfahrenden Fahrzeug wieder zu (z. B. weil dessen Fahrer beschleunigt), wird auch die vom *ACC-Controller* vorgegebene Soll-Geschwindigkeit solange wieder erhöht, bis die durch den Fahrer vorgegebene Geschwindigkeit erreicht ist, wobei gleichzeitig der Mindestabstand weiterhin aktiv überwacht und beibehalten wird. Eine Auslösung der Notbremsung erfolgt nur, wenn der Mindestabstand soweit unterschritten wird, dass ohne eine sofortige Bremsung eine Kollision unvermeidbar ist. Da es sich um ein sicherheitskritisches System handelt, unterzieht der *ACC-Controller* die empfangenen Sensorwerte Plausibilitätsprüfungen. Um den Ausfall der unterschiedlichen Komponenten (z. B. Sensoren) detektieren zu können, erfolgt die Übertragung der Messwerte in periodischen Abständen (siehe unten), sodass hierüber ein Heartbeat realisiert wird. Bleibt dieser über eine definierte Dauer hinweg aus, wird in einen Notfallmodus gewechselt und der ACC-Controller deaktiviert.

Der *Radarsensor* misst den Abstands zu dem vorausfahrenden Fahrzeug oder Hindernis. Gleichzeitig bestimmt er auf Basis der aktuellen Geschwindigkeit des eigenen Fahrzeuges und der Veränderung des Abstands zu dem jeweiligen Hindernis dessen Geschwindigkeit. Zusätzlich verfügt der Radarsensor über einen Alarmausgang, welcher eine Unterschreitung des minimalen Sicherheitsabstandes sofort signalisiert. Der Alarmausgang kann (ein geeignetes Übertragungsprotokoll vorausgesetzt) dazu verwendet werden, in einem solchen Fall unmittelbar einen Rahmen an den ACC-Controller zu senden.

Das physikalische Modell der Regelstrecke (hier des Autos) wird durch die Simulationskomponente *CarModel* beschrieben und besteht aus einem vereinfachten Modell des Motors und der Kraftübertragung sowie der Bremsen (Aktuatoren). Neben den Steueranweisungen für die Aktuatoren findet die Steigung der Strecke (in Prozent) ebenfalls Eingang in das Modell. Zusätzlich berücksichtigt das Modell Roll- und Luftwiderstand[2] als Störgrößen. Mit Hilfe einer Differentialgleichung, basierend auf dem Kräftegleichgewicht, wird anhand der Ansteuerung der Aktuatoren, den Störgrößen sowie der Masse des Autos die resultierende Beschleunigung und Geschwindigkeit ermittelt. Die aktuelle Geschwindigkeit wird wiede-

[2]Roll- und Luftwiderstand werden vereinfacht durch einen zur Geschwindigkeit proportionalen Faktor modelliert.

rum den anderen Komponenten des ACCs über das gemeinsame Kommunikationsmedium bereitgestellt, d.h., der *Geschwindigkeitssensor* ist implizit Bestandteil des *CarModel*.

Die *GradientInput*-Komponente ermittelt auf Basis der aktuellen Geschwindigkeit die Position des Autos auf einem vorgegebenen Kurs und bestimmt das dort vorherrschende Gefälle bzw. die dortige Steigung. Diese Informationen werden über einen direkten Link dem *CarModel* zur Verfügung gestellt. *SpeedInput* simuliert das Verhalten des Fahrers, indem es dem ACC-Controller die gewünschte Soll-Geschwindigkeit über den zentralen *EngineControlBus (ECB)* zur Verfügung stellt.

Die Repräsentation der Funktionalitäten von *SpeedInput*, *Radarsensor* und *GradientInput* als eigenständige Simulationskomponenten ermöglicht die Eingabe beliebiger Stimuli. So könnten z. B. in Feldversuchen aufgezeichnete Messwerte als Basis für die Simulation spezifischer Szenarien bzw. Abläufe dienen. Der Vorteil einer Simulation liegt hierbei auf der Hand, die Szenarien lassen sich beliebig oft und unter reproduzierbaren Bedingungen simulieren, analysieren und umfassend auswerten.

Der ECB dient als zentraler Kommunikationsbus und verbindet *SpeedInput*, *CarModel*, *Radarsensor* sowie *ACC-Controller* miteinander. Zusätzlich ist an den ECB ein Traffic-Generator *BusTrafficGen* angeschlossen, welcher es erlaubt, das Kommunikationsverhalten anderer, nicht explizit innerhalb dieses Szenarios abgebildeter Knoten/Komponenten zu simulieren, um auf diese Weise die in der Praxis höhere Busauslastung berücksichtigen zu können[3]. Für die Realisierung des *BusTrafficGen* kommen die in Kapitel 10 beschriebenen Mechanismen zum Einsatz. Um die Möglichkeit der Kopplung unterschiedlicher Kommunikationstechnologien über Gateways zu demonstrieren, haben wir das Szenario um ein zusätzliches Kommunikationsmedium für den Informationsaustausch zwischen Autos (*Car2Car*) erweitert. Die *Car2Car*-Kommunikation basiert auf einer drahtlosen Technologie – in unserem Fall IEEE 802.11 – und ermöglicht den Autos in unmittelbarer Nähe die Übermittlung und Propagierung von Verkehrswarnungen. Der Gateway *GW1* leitet Verkehrswarnungen über den ECB weiter, sodass allen angeschlossenen Knoten diese Informationen zur Verfügung stehen. Ebenso können die Knoten über diesen Mechanismus Verkehrswarnungen senden (z. B. könnte der *ACC-Controller* andere Autos über eine eingeleitete Notbremsung informieren). Die Verkehrswarnungen dienen in diesem konkreten Szenario lediglich dazu, den Aufbau einer möglichen Car-to-X Simulationsumgebung mittels FERAL zu illustrieren und die Funktionalität der Gateways zu testen, haben jedoch keinen funktionalen Einfluss auf den Ablauf.

Um den Austausch der Kommunikationstechnologie des ECB so problemlos wie möglich zu gestalten, sind – mit Ausnahme des ACC-Controllers[4] – alle anderen FSCs mittels Bridges an den ECB angebunden oder verfügen, wie der Traffic-Generator, direkt über entsprechende Eigenschaften. Der Gateway GW1, der für die Weiterleitung der Verkehrswarnungen zwischen dem Car2Car-Medium und dem ECB verantwortlich ist, basiert auf der in Kapitel 10.3 beschriebenen universellen Gateway-Simulationskomponente, deren Weiterleitungsverhalten in Form einer DBL-Spezifikation festgelegt wird.

Die Verkehrswarnungen werden in diesem Szenario als sporadische Nachrichten modelliert und ausschließlich durch die Komponente *OtherCar* versendet. Die Simulationskomponente *OtherCar* wurde als native Simulationskomponente realisiert und verzichtet, wie der *ACC-Controller*, auf den Einsatz einer Bridge zur Anbindung an die CSCs (vgl. Kapitel 11.2.2). Bei den Nachrichten, die die aktuelle Geschwindigkeit, Steuerwerte für den Motor (Motor-Steuersignal) sowie die Soll-Geschwindigkeit übermitteln, handelt es sich um periodische Nachrichten mit einem Zwischenankunftsintervall von 20 ms (aktuelle Geschwindigkeit, Steuerwerte für Motor) bzw. einem Zwischenankunftsintervall von 100 ms (Soll-Geschwindigkeit). Abstand und Geschwindigkeit zu einem vorausfahrenden Fahrzeug oder Hindernis (Radarsignal) werden durch den Radarsensor periodisch alle 100 ms übertragen. Sobald jedoch der

[3] In der Realität wird der ECB nicht exklusiv für das ACC zur Verfügung stehen. Vielmehr wird das ACC System nur ein weiteres Assistenzsystem sein, dessen Komponenten über den ECB interagieren.

[4] Der ACC-Controller wird in SDL spezifiziert und verwendet die nativen SEnF-Treiber (vgl. Kapitel 8.4.2), um direkt mit den CSCs zu interagieren. Diese bilden die Schnittstelle und das Verhalten realer Hardwaretreiber nach. So kann aus dem SDL-System (ohne Modifikationen) Code für eine Zielplattform mit realer Hardwareunterstützung für die jeweiligen Kommunikationstechnologien erzeugt werden (vgl. Kapitel 11.2.2).

minimale Sicherheitsbereich unterschritten wird, erfolgt eine Reduktion des Zwischenankunftsintervalls[5] auf 20 ms. Die Anweisung, das Fahrzeug über den Einsatz der Bremsen zu verlangsamen oder zu stoppen (Bremsbefehl), wird über eine sporadische Nachricht realisiert, die nur beim Unterschreiten des minimalen Sicherheitsabstandes vom *ACC-Controller* gesendet wird.

11.2.2 Erstellung der konkreten Simulationssysteme

Durch die Auswahl spezifischer Simulatoren sowie Verhaltensmodelle, der Festlegung der Kommunikationstechnologien durch konkrete CSCs, samt deren Konfiguration, sowie der Spezifikation des Kommunikationsverhaltens der Bridges und Gateways lässt sich aus dem abstrakten Simulationssystem (Abbildung 11.2) des ACC Systems ein konkretes Simulationssystem ableiten. Wir wählen verschiedene CSCs für den ECB und erhalten so unterschiedliche konkrete Simulationssysteme, mit deren Hilfe sich die möglichen Designalternativen in Bezug auf die Wahl einer geeigneten Kommunikationstechnologie für den ECB evaluieren lassen.

Insgesamt evaluieren wir fünf unterschiedliche konkrete Simulationssysteme unseres ACC Systems, die sich hinsichtlich der Kommunikationstechnologie bzw. dem Kommunikationsparadigma unterscheiden. Die Tabelle 11.1 gibt eine Übersicht über die konkreten Simulationssysteme sowie die Auswahl der konkreten Simulatoren und deren Verhaltensmodelle zur Realisierung der Simulationskomponenten (angegeben in Klammern). Die letzte Zeile der Tabelle ordnet jedem der konkreten Simulationssysteme einen Namen zu, welcher die Zuordnung der später dargestellten Messergebnisse erleichtert.

Mit Ausnahme des ECB kommt für jede andere FSC und CSC jeweils der gleiche Simulator zum Einsatz. Lediglich der *ACC-Controller* verwendet ein anderes, auf den ECB zugeschnittenes Verhaltensmodell (siehe unten). Bei allen anderen FSCs kann aufgrund der Bridges das Verhaltensmodell unverändert weiterverwendet werden. Das Verhalten der Regelstrecke (repräsentiert durch die Simulationskomponente *CarModel*) wurde mit Hilfe eines Simulink Modells beschrieben. *Radarsensor*, *GradientInput*, *SpeedInput* und *OtherCar* wurden als native Simulationskomponenten realisiert, deren einfaches Verhaltensmodell direkt in Form von Java-Code vorliegt.

Der *ACC-Controller* wurde in diesem Szenario mittels SDL spezifiziert und verwendet die jeweiligen SEnF-Treiber, um direkt mit der CSC des ECB zu interagieren (vgl. Kapitel 8.4). Die verwendeten SEnF-Treiber imitieren sowohl die Schnittstelle als auch das Verhalten eines Hardwaretreibers zur Ansteuerung eines realen Bausteins, sodass das SDL-System des *ACC-Controller* somit näher an einer realen Umsetzung ist, als dies bei der Verwendung einer Bridge der Fall wäre. Deshalb ließe sich das SDL-System auch ohne Anpassungen für eine konkrete Hardwareplattform mit echtem Kommunikationscontrollern übersetzen und ausführen. Da bei den fünf konkreten Simulationssystemen verschiedene CSCs für den ECB verwendet werden, erfordert dies im Gegenzug aber auch unterschiedliche SDL-Systeme, die jeweils den passenden SEnF-Treiber für ACOM, CAN, FlexRay oder ns-3 verwenden. Durch die Mechanismen zur Kapselung und Wiederverwendung, die SDL über SDL-Pakete, Block- und Prozesstypen bietet, weichen die SDL-Spezifikationen nur in der Instanziierung der Kommunikationsschichten voneinander ab. Diese kapselt die Verwendung des konkreten SEnF-Treibers und stellt eine einheitliche Kommunikationsschnittstelle für das restliche SDL-System bereit.

Um die Auswirkungen unterschiedlicher Designentscheidungen in Bezug auf die Kommunikationstechnologie (z. B. Übertragungsverzögerungen) des ECB zu untersuchen, nutzen die fünf konkreten Simulationssysteme drei unterschiedliche Technologien sowie unser abstraktes Kommunikationsmodell für die Simulation des ECB. Somit kommen alle von FERAL unterstützten CSCs – CAN-CSC, FR-CSC, ns-3-CSC (Switched Ethernet) und ACOM-CSC – zum Einsatz. Der verwendete CAN-Bus wird mit einer Übertragungsrate von 1 MBit/s betrieben. Der FlexRay-Bus nutzt exklusive statische Reservierungen für sämtliche Nachrichtentypen des Anwendungsszenarios sowie eine Übertragungsrate von 10 MBit/s.

[5] Über den *Radaralarm*-Ausgang wird in diesem Fall, bei ereignisbasierten Kommunikationsprotokollen, eine sofortige Übertragung angestoßen, mit einer anschließenden Totzeit von 20 ms, vgl. hierzu Listing 10.7 aus Kapitel 10.2.1.3.

ECB	CSC	CAN-CSC	FR-CSC	FR-CSC	ns-3 (Ethernet)	ACOM-CSC
Car2Car	CSC	ns-3 (IEEE 802.11, WLAN)				
CarModel	FSC	Matlab Simulink (Simulink Modell)				
Radarsensor	FSC	Native Simulationskomponente (Java)				
GradientInput	FSC	Native Simulationskomponente (Java)				
SpeedInput	FSC	Native Simulationskomponente (Java)				
OtherCar	FSC	Native Simulationskomponente (Java)				
BusTrafficGen	FSC	Isolierte Bridge nach Kapitel 10.4				
ACC-Controller	FSC	SDL (CAN)	SDL (FlexRay)	SDL (FlexRay)	SDL (ns-3)	SDL (ACOM)
Name des konkreten Simulationssystems:		CAN	FlexRay ASync	FlexRay Sync	Ethernet	ACOM

Tabelle 11.1: Übersicht der konkreten Simulationssysteme.

Unser Ethernet-Netzwerk betreiben wir mit einer Übertragungsrate von 10 MBit/s, um eine bessere Vergleichbarkeit der Ergebnisse mit denen der anderen Bussysteme zu ermöglichen. Als Protokoll kommen UDP-Broadcasts zum Einsatz, wobei den verschiedenen Nachrichtentypen unterschiedliche Ports zugewiesen wurden (vgl. Tabelle 11.2). In Bezug auf das abstrakte Kommunikationsmodell (ACOM) nehmen wir an, dass die nutzbare Bandbreite 1 MBit/s beträgt. Diese wird bei gleichzeitigen Übertragungen fair auf alle Knoten verteilt, sodass alle Nachrichten in gleichem Ausmaß von Verzögerungen betroffen sind. Als Grundlage für unseren Vergleich der Designalternativen dienen die minimalen, maximalen und mittleren Ende-zu-Ende-Verzögerungen der Nachrichten des ACC Systems, welche bei der Simulation des Szenarioablaufs (vgl. Kapitel 11.2.3.1) beobachtet werden.

Zusätzlich zu den unterschiedlichen Kommunikationstechnologien für den ECB betrachten wir die Auswirkungen des Kommunikationsverhaltens auf die Ende-zu-Ende-Verzögerungen bei FlexRay genauer. Hierzu haben wir zwei konkrete Simulationssysteme *FlexRay ASync* und *FlexRay Sync* definiert, welche beide einen identisch konfigurierten FlexRay-Bus für den ECB verwenden. Die beigen Systeme unterscheiden sich nur im Kommunikationsverhalten, welches bei beiden Systemen durch unterschiedliche DBL-Spezifikationen der *RBridge* (Radarsensor), *SBridge* (SpeedInput) und *CBridge* (CarModel) umgesetzt wird.

Bei dem Simulationssystem *FlexRay ASync* ist das Kommunikationsverhalten der Simulationskomponenten (und damit auch das Kommunikationsverhalten der durch diese Komponenten repräsentierten Applikationen) zeitlich von dem Kommunikationszyklus des FlexRay-Protokolls entkoppelt. Dies bedeutet, dass die Nachrichten zu jedem beliebigen Zeitpunkt innerhalb des Kommunikationszyklus erzeugt und dem CC (PCU) übergeben werden können. Die entsprechenden Rahmen werden dann vom CC in dem nächsten für diese Nachricht reservierten Slot versendet. Wird eine Nachricht unmittelbar nach dem Beginn des reservierten Slots an den CC übergeben, so vergeht – sofern nur ein Slot pro Zyklus reserviert wurde – ein ganzer Kommunikationszyklus, bis die Übertragung erfolgt (Worst Case). Im Mittel beträgt die Verzögerung (eine uniforme Verteilung vorausgesetzt) aufgrund der fehlenden Synchronisation bei diesem Beispiel, einen halben Kommunikationszyklus.

Beim Simulationssystem *FlexRay Sync* wird die Erzeugung der Nachrichten für Ist-Geschwindigkeit, Soll-Geschwindigkeit und das Radarsignal mit dem FlexRay Kommunikationszyklus sowie den vorhandenen Slotreservierungen synchronisiert. Erreicht wird dies durch die Verwendung des *SyncTrigger* (vgl. Kapitel 10.2.1.2) innerhalb der DBL-Spezifikationen. Da der *SyncTrigger* zu Beginn jedes Kommunikationszyklus ausgelöst wird und die Slotreservierungen bekannt sind, können die Nachrichten *"Just-in-Time"*, also unmittelbar vor dem Beginn des reservierten Slots generiert werden. Dies hat mehrere Vorteile: Zum einen sind die enthaltenen Daten so aktuell wie möglich, zum anderen werden Jitter und Ende-zu-Ende-Verzögerungen minimiert. Ein Nachteil in der Praxis besteht darin, dass hieraus eine enge zeitliche Kopplung zwischen der Anwendung und dem Kommunikationszyklus resultiert, mit entsprechend hohen Anforderungen an das Scheduling und die Echtzeitfähigkeit.

Die Tabelle 11.2 liefert eine Übersicht der innerhalb des ACC Systems existierenden Nachrichtentypen und der eingesetzten Übertragungsoptionen bei den verwendeten Kommunikationstechnologien. Um den Besonderheiten eines TDMA-basierten Zugriffsverfahrens im Vergleich zu dem ereignisbasierten Zugriff bei CAN und Ethernet Rechnung zu tragen, nutzt FlexRay einen kurzen Kommunikationszyklus von ca. 5 ms[6]. Für jeden Nachrichtentyp existiert pro Kommunikationszyklus mindestens eine exklusive Slotreservierung (statisches Segment), für sicherheitskritische Nachrichtentypen (Ist-Geschwindigkeit, Motor-Steuersignal, Radarsignal und Bremsbefehl) werden zwei statische Slots pro Kommunikationszyklus reserviert. Auf diese Weise wird sowohl der Jitter als auch die maximale und mittlere Übertragungsverzögerung reduziert. Die Tabelle 11.2 listet sowohl die Slotreservierungen als auch die zugeordneten

[6]Die Länge des statischen Segments beträgt etwa 2,5 ms und setzt sich aus 91 statischen Slots zusammen. Als Vorbild für diesen Kommunikationszyklus dient die von BMW veröffentliche Konfiguration des Aufbaus, des von Ihnen verwendeten Kommunikationszyklus [BPS08].

CAN-Identifier für die verschiedenen Nachrichtentypen auf. Die Zuordnung der CAN-Identifier erfolgte unter Berücksichtigung der Kritikalität der Nachrichtentypen.

Message Type	**CAN-Identifier**	**reservierte FlexRay Slots**	**Ethernet-Ports**	**ACOM-Identifier**
Radarsignal	20	Slot 3 und 88	200	20
Ist-Geschwindigkeit	70	Slot 1 und 86	700	70
Soll-Geschwindigkeit	45	Slot 2	450	45
Motor-Steuersignal	110	Slot 5 und 90	1100	110
Bremsbefehl	10	Slot 6 und 91	100	10
Verkehrswarnung	188	Slot 67	1880	188

Tabelle 11.2: Übersicht der Nachrichtentypen des ACC Systems.

11.2.3 Ergebnisse der Simulationen

Wir simulieren alle fünf konkreten Simulationssysteme unter Verwendung des vorgegebenen Anwendungsfalls mit identischen Eingabestimuli und vergleichen die Designalternativen bzgl. Kommunikationstechnologie und Kommunikationsverhalten anhand der ermittelten minimalen, maximalen und mittleren Ende-zu-Ende-Verzögerungen der Nachrichten des ACC sowie der Regelgüte.

11.2.3.1 Das Anwendungsszenario

Bevor wir uns den Simulationsergebnissen hinsichtlich der Übertragungsverzögerungen widmen, betrachten wir zunächst das funktionale Verhalten des ACC Systems anhand der Simulationsergebnisse für das untersuchte Anwendungsszenario. Das Regelverhalten sowie der Ablauf sind als Diagramm in Abbildung 11.3 dargestellt.

In dem gewählten Anwendungsfall startet unser Fahrzeug mit einer Ausgangsgeschwindigkeit von 50 km/h. Nach 3 Sekunden wird die Soll-Geschwindigkeit von 50 km/h auf 140 km/h erhöht. Während der gesamten Simulation fährt ein anderes Auto mit konstant 100 km/h vor dem Versuchswagen her, wobei der initiale Abstand beider Fahrzeuge 220 Meter beträgt. Nach 54 Sekunden startet der vorausfahrende Wagen ein Bremsmanöver und hält an.

Der Abstand zwischen den beiden Fahrzeugen erhöht sich solange, bis unser Versuchsfahrzeug auf über 100 km/h beschleunigt hat (nach ca. $6,5$ Sekunden). Ist diese Marke überschritten, schrumpft der Abstand zum vorausfahrenden Fahrzeug beständig. Nach 20 Sekunden Simulationszeit sinkt er unter 140 Meter. Dies entspricht dem doppelten minimalen Sicherheitsabstand und bewirkt, dass der ACC-Controller mit der Reduktion und Anpassung der Geschwindigkeit beginnt, wobei er sich über die Vorgabe des Fahrers hinwegsetzt (die weiterhin 140 km/h beträgt). Wie anhand des Geschwindigkeitsverlaufes ersichtlich, führen die Anpassungen dazu, dass die Geschwindigkeit unseres Versuchsfahrzeugs beim Erreichen des minimalen Sicherheitsabstandes (bei 100 km/h etwa 50 Meter) an die Geschwindigkeit des vorausfahrenden Fahrzeugs adaptiert wurde (ca. 50 Sekunden nach Start der Simulation). Sobald das vorausfahrende Fahrzeug seine Bremsung einleitet, reagiert auch der ACC-Controller aufgrund des rapide abnehmenden Abstands mit einer Notbremsung und bringt so das Versuchsfahrzeug noch rechtzeitig zum Stehen und verhindert einen Auffahrunfall. Der Fahrer ist bei diesem Bremsvorgang nicht aktiv involviert (die vorgegebene Soll-Geschwindigkeit beträgt weiterhin 140 km/h).

Abbildung 11.3 stellt die Ergebnisse der Simulation graphisch anhand des Verlaufs der verschiedenen Kenngrößen des Autos sowie der Regelung dar. Hierbei handelt es sich um das Ergebnis der Simulation des

konkreten Simulationssystems *CAN*, bei dem der ECB durch einen CAN-Bus (CAN-CSC) realisiert wird. Die rot gepunktete Linie in der Abbildung entspricht der vom Fahrer vorgegebenen Zielgeschwindigkeit, während die blaue Linie die aktuelle Geschwindigkeit des Autos repräsentiert, welche durch den ACC-Controller gesteuert wird. Die grüne gestrichelte Linie zeigt das Ergebnis der Abstandsmessung des Radarsensors und gibt in unserem Szenario den aktuellen Abstand zu dem vorausfahrenden Fahrzeug an.

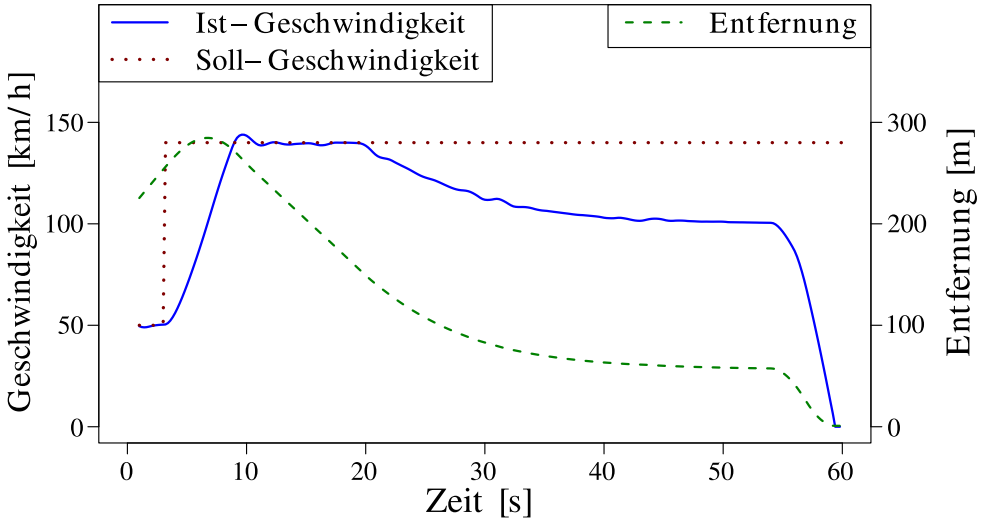


Abbildung 11.3: Automatische Anpassung der Geschwindigkeit des Fahrzeuges durch den ACC-Controller, um den minimalen Sicherheitsabstand zu einem vorausfahrenden Fahrzeug aufrechtzuerhalten (Simulationssystem *CAN*).

Wie aus der Grafik ersichtlich, funktioniert das ACC System wie gewünscht. Allerdings ist auch zu erkennen, dass der ACC-Controller bzw. der von ihm implementierte PID-Regler noch nicht optimal eingestellt ist. So kommt es nach 10 Sekunden zu einem Überschwinger bei der Geschwindigkeitsregelung, welcher dazu führt, dass die aktuelle Geschwindigkeit die vorgegebene Geschwindigkeit von 140 km/h geringfügig überschreitet. Ansonsten ist die Regelgüte jedoch zufriedenstellend. Es ergeben sich im Verlauf der Simulation nur kleine Schwankungen der Ist-Geschwindigkeit. Diese resultieren unter anderem aus den stark unterschiedlichen Steigungen der verwendeten Teststrecke (vgl. Abbildung 11.1). Die Anpassung der Geschwindigkeit an ein vorausfahrendes Fahrzeug funktioniert korrekt und nicht zu abrupt, sodass der Komfort nicht leidet. Ebenso gelingt die Einleitung der Notbremsung und die Vermeidung einer Kollision (verbleibender Abstand zwischen beiden Fahrzeugen ca. $0,5$ Meter).

Wir haben das Szenario mit allen fünf vorgestellten konkreten Simulationssystemen simuliert. In allen Fällen arbeitete das ACC System funktional korrekt und auch bei einer genaueren Auswertung zeigt sich, dass sich die unterschiedlichen Kommunikationstechnologien kaum auf die Regelgüte auswirken. Die minimalen Schwankungen sind für die Praxis nicht relevant und fallen so klein aus, dass keine vernünftige Visualisierung möglich ist. Daher verzichten wir auf eine graphische Darstellung der Simulationsergebnisse für die anderen konkreten Simulationssysteme. Stattdessen widmen wir uns den Auswirkungen unterschiedlicher Kommunikationstechnologien auf die durchschnittliche, minimale und maximale Ende-zu-Ende-Verzögerung der Nachrichten des ACC Systems.

11.2.3.2 Resultierende Verzögerungen bei unterschiedlichen Kommunikationstechnologien

Die Abbildung 11.4 stellt die in unserem Anwendungsfall gemessenen Ende-zu-Ende-Verzögerungen, in den fünf untersuchten konkreten Simulationssystemen *CAN*, *FlexRay Async*, *FlexRay Sync*, *Ethernet* und *ACOM* mit ihren jeweiligen Kommunikationstechnologien dar. Die Höhe der Balken gibt die

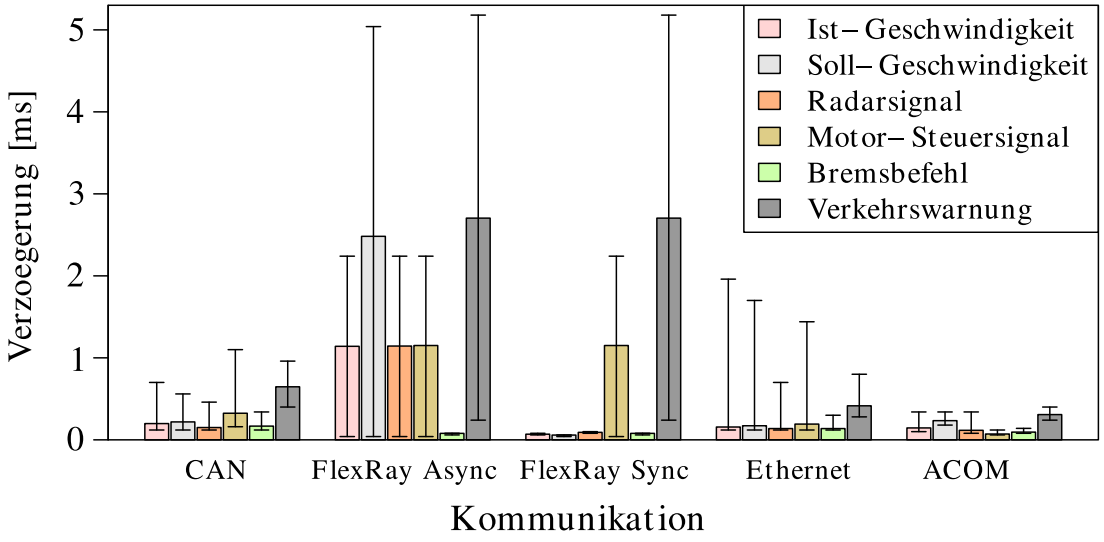


Abbildung 11.4: Durchschnittliche, minimale und maximale Verzögerung für die Rahmen der Nachrichten des ACC Systems für unterschiedliche Kommunikationslösungen.

durchschnittliche Ende-zu-Ende-Verzögerung an, während die Antennen die minimalen und maximalen Ende-zu-Ende-Verzögerungen zeigen, die während der Simulation auftraten.

Der eingesetzte Traffic-Generator (BusTrafficGen, vgl. Abbildung 11.2) simuliert eine Grundlast auf dem EBC. So wird nachgestellt, dass in einem realen Auto weitere Systeme über den ECB kommunizieren würden, die in dieser Simulation nicht detailliert nachgebildet wurden. Insgesamt erhöht der Traffic-Generator die Buslast auf dem Kommunikationsmedium auf ein realistisches Maß, sodass der verwendete CAN-Bus zu 39,7 %, der FlexRay-Bus zu 9,8 %, das verwendete Ethernet-Netzwerk zu 23,9 % und die Übertragungskapazität des abstrakten Kommunikationsmodells ACOM zu 34,1 % ausgelastet wird.

Wir vergleichen zunächst die Ergebnisse bzgl. der Verzögerungen zwischen den Simulationssystemen *CAN*, *FlexRay Async* und *Ethernet* und widmen uns anschließend der Variante *FlexRay Sync* und deren Besonderheiten. Da sowohl CAN als auch Ethernet einen sofortigen Medienzugriff erlauben (ohne dass, wie bei FlexRay, zunächst auf einen reservierten Slot gewartet werden muss), führt dies dazu, dass bei diesen beiden Ansätzen die durchschnittliche Ende-zu-Ende-Verzögerung deutlich kleiner ist als bei FlexRay Async. Trotz der hier besseren Reaktionszeiten haben diese beiden Protokolle den Nachteil, dass sie keine maximale Übertragungsverzögerung garantieren können. So konnten z. B. bei anderen Simulationen mit einer Buslast von 79 % bei CAN Ende-zu-Ende-Verzögerungen von 5,5 ms für den Bremsbefehl beobachtet werden. Bei höheren Buslasten – oder anderen Charakteristiken der Buslast – kann es aufgrund der immer größeren Verzögerungen zur Verschlechterung der Regelgüte oder gar zu funktionalen Problemen kommen, da z. B. Sensorwerte zu spät empfangen oder Stellwerte nicht schnell genug umgesetzt werden können. Ein entsprechendes Beispiel haben wir für CAN in Kapitel 11.2.3.3 untersucht.

Vergleicht man die gezeigten Ergebnisse für Ethernet und CAN genauer, so fällt auf, dass der Faktor 10 in Bezug auf die Übertragungsgeschwindigkeit bei Ethernet, nicht zu einer diesem Faktor entsprechenden Verbesserung der durchschnittlichen Übertragungsverzögerung führt. Eine Ursache hierfür ist der Overhead, den das Ethernet-Rahmenformat erzeugt: Jeder Ethernet-Rahmen besitzt eine Länge von mindestens 64 Byte, die verwendete Nutzlast in unserem ACC System liegt pro Rahmen jedoch bei nur 2 bis 8 Byte. Auch die aufgetretenen maximalen Übertragungsverzögerungen bei Ethernet sind höher als bei CAN. Dies ist im Wesentlichen auf die fehlende Unterstützung von Prioritäten zurückzuführen, da alle Rahmen (also auch alle Rahmen des Füllverkehrs) gleichberechtigt um das Medium konkurrieren.

Das Simulationssystem *ACOM* nutzt unser abstraktes Kommunikationsmodell und verwendet hierbei den Ansatz, gleichzeitige Übertragungen zuzulassen, die verfügbare Bandbreite jedoch fair zwischen allen laufenden Übertragungen aufzuteilen. Als Resultat ist die Schwankung der Verzögerungen zwischen den verschiedenen Nachrichtentypen im Vergleich zu anderen Kommunikationstechnologien sehr gering. Die geringe maximale Verzögerung von ACOM gegenüber der auf Ethernet basierten Lösung resultiert aus dem Overhead, den Ethernet in dieser speziellen Anwendung aufweist. Die andere Zeitverteilung im Vergleich zu *CAN* ist darauf zurückzuführen, dass zum einen jeder Knoten sofort seine Übertragung starten kann (Effekte wie Priority-Inversion treten nicht auf, stattdessen erfolgt eine faire Verteilung der Bandbreite) und keine Priorisierung unterstützt wird, zum anderen wirkt sich hierdurch natürlich auch die erzeugte Buslast des *BusTrafficGen* anders auf die einzelnen Übertragungen aus. Zusätzlich fällt der Overhead der ACOM-Rahmen gegenüber den CAN-Rahmen etwas geringer aus (fehlendes Bit-Stuffing, kürzerer Header).

Betrachten wir nun noch einmal etwas genauer die Ergebnisse des zeitgetriggerten FlexRay-Protokolls und vergleichen die Varianten *FlexRay Async* und *FlexRay Sync*. Wie in Abschnitt 11.2.2 erläutert, beträgt die Länge des FlexRay Kommunikationszyklus ca. 5 ms, wobei das statische Segment eine Länge von 91 Slots mit insgesamt ca. $2,5$ ms aufweist. Für alle Rahmentypen des ACC Systems existieren innerhalb des statischen Segments exklusive Slotreservierungen (vgl. Tabelle 11.2). Sicherheitskritische Nachrichten wie Radarsignal, Ist-Geschwindigkeit sowie Motor-Steuersignal und Bremsbefehl verfügen pro Kommunikationszyklus über zwei Slotreservierungen. Hiervon befindet sich jeweils eine zu Beginn und eine am Ende des statischen Segments (d.h., der Abstand der Slotreservierung beträgt ungefähr $2,5$ ms). Durch die exklusiven Reservierungen wird die maximale Verzögerung bei der Übertragung von Rahmen eines Typs durch den maximalen Abstand zweier aufeinanderfolgender – für diesen Typ reservierter – Slots festgelegt. Die Ergebnisse der Simulation (Abbildung 11.4) zeigen, dass bei dem gewählten Szenario, bei den Nachrichten des ACC Systems sowohl die gemessenen durchschnittlichen Verzögerungen als auch die maximalen Verzögerungen, entsprechend den vorherigen Überlegungen, beobachtet werden können (bei zwei Slotreservierungen eine maximale Verzögerung von $2,5$ ms und eine durchschnittliche Verzögerung von $1,25$ ms resp. 5 ms und $2,5$ ms bei nur einer Slotreservierung). Auffallend ist in der Abbildung die minimale Verzögerung in Bezug auf den Bremsbefehl, hierfür ist die geschickte Wahl der Slotreservierung verantwortlich. Ein Bremsbefehl wird nur erzeugt, sofern der Abstand (übermittelt durch das Radarsignal) einen bestimmten Schwellwert unterschreitet. Da die reservierten Slots für den Bremsbefehl jeweils zwei Slots hinter den für das Radarsignal reservierten Slots liegen, kann der ACC-Controller in dieser Zeit den empfangenen Wert verarbeiten und falls erforderlich, den Rahmen mit dem Bremsbefehl rechtzeitig für den folgenden Slot erzeugen. Auf diese Weise wird die resultierende Verzögerung sowie der auftretende Jitter minimiert. Der Vorteil von FlexRay besteht darin, dass im Gegensatz zu CAN, Ethernet oder ACOM die maximale Verzögerung aufgrund des TDMA-Schemas garantiert wird und diese darüber hinaus, bei Verwendung exklusiver Slotreservierungen, unabhängig von der Busauslastung ist.

Nun betrachten wir die Auswirkungen, wenn wir eine Synchronisation zwischen dem Kommunikationsverhalten und dem Kommunikationszyklus einführen und ausnutzen, um Rahmen mit einer möglichst geringen Verzögerung vor Beginn ihrer reservierten Slots zu erzeugen. Dieses Verhalten haben wir für ausgewählte konkrete Simulationskomponenten im Rahmen des konkreten Simulationssystems *FlexRay Sync* umgesetzt. Wie in Abbildung 11.4 klar zu erkennen, führt die gewählte Umsetzung der Synchronisation durch den `SyncTrigger` bei den Rahmentypen Radarsignal, Ist-Geschwindigkeit und Soll-Geschwindigkeit – wie erwartet – zu einer Reduktion der mittleren Übertragungsverzögerung sowie des auftretenden Jitters im Vergleich zu der nicht synchronisierten Version *FlexRay Async* oder ereignisgetriggerten Kommunikationsvarianten (*CAN*, *Ethernet*, *ACOM*). Der Grund hierfür liegt auf der Hand, die Nachrichten werden Just-in-Time, d.h. erst kurz vor Beginn des reservierten Slots erzeugt

und dann garantiert in diesem versendet. Bei den Rahmen für Bremsbefehl, Motorsteuersignal[7] und Verkehrswarnung wurde auf die Synchronisation verzichtet. Dementsprechend sind die Ergebnisse mit denen der *FlexRay Async* Variante nahezu identisch.

In Bezug auf die Verkehrswarnung bezieht sich die dargestellte Verzögerung (in allen Simulationssystemen) jeweils auf die Ende-zu-Ende-Verzögerung; diese beinhaltet sowohl die Übertragung über das Car2Car-Medium als auch die Bearbeitungsverzögerung durch den Gateway GW1 sowie die anschließende Übertragung des Rahmens über den ECB. Aus diesem Grund sind die Übertragungsverzögerungen – insbesondere auch die beobachteten minimalen Übertragungsverzögerungen – bei den Verkehrswarnungen generell höher als bei den anderen Rahmentypen.

11.2.3.3 Auswirkungen von Buslast auf die Regelgüte

Wie unsere Evaluation zeigte, ist die Regelgüte bei allen Kommunikationstechnologien in allen fünf konkreten Simulationssystemen adäquat. Nun untersuchen wir mit Hilfe von FERAL, was passiert, wenn es zu einer Situation kommt, in der andere Knoten (z. B. eines anderen Assistenzsystems) das Kommunikationsmedium nahezu vollständig auslasten und die zum ACC System gehörenden Nachrichtentypen daher einer sehr großen Verzögerung unterliegen.

Als Ausgangsbasis unserer Simulation dient das konkrete Simulationssystem *CAN*. Basierend auf diesem erstellen wir ein weiteres konkretes Simulationssystem *CAN Overload*. Wir modifizieren die DBL-Spezifikation des Traffic-Generators dahingehend, dass dieser für eine Zeitspanne von 10 Sekunden nahezu die vollständige Bandbreite des CAN-Busses belegt. Hierdurch kommt es in diesem Zeitraum zu einer starken Verzögerung der Rahmen des ACC Systems, da die durch den Traffic-Generator gesendeten Rahmen teilweise eine höhere Priorität als die des ACC Systems aufweisen. In der Praxis könnte eine solche Überlast beispielsweise eintreten, wenn aufgrund einer Gefahrensituation viele Assistenzsysteme gleichzeitig aktiv werden.

Die funktionalen Auswirkungen sind in Abbildung 11.5 dargestellt und zeigen, dass infolge der erhöhten Verzögerungen – diese treten zwischen der zwanzigsten und dreißigsten Sekunde der Simulation auf – die Regelgüte deutlich reduziert wird und der Fahrkomfort leidet (vgl. Abbildung 11.3). Als Konsequenz aus dieser Erkenntnis kann der Entwickler entweder die Nachrichtenplanung überarbeiten (z. B. hinsichtlich der Zuordnung der Nachrichtenprioritäten) oder evtl. die Assistenzsysteme durch zusätzliche CAN-Busse voneinander isolieren.

Die Abbildung 11.6 zeigt die gemessenen minimalen, maximalen und mittleren Übertragungsverzögerungen der ACC-bezogenen Rahmentypen, im direkten Vergleich zwischen den konkreten Simulationssystemen *CAN* und *CAN Overload*. An dieser Stelle sei darauf hingewiesen, dass anders als in Abbildung 11.4 eine logarithmische Zeitskala zur Anwendung kommt, um die Verzögerungen in beiden Simulationssystemen sinnvoll in einer Grafik darstellen zu können. Aufgrund des vergleichsweise kurzen Zeitraums, in dem die Überlast auftritt, sind die durchschnittlichen Übertragungsverzögerungen in dem Simulationssystem *CAN Overload* im Vergleich zum Simulationssystem *CAN* nur geringfügig erhöht. Um dies noch einmal deutlich zu illustrieren, zeigt die dritte Gruppe von Balken mit der Unterschrift *CAN Overload (20-30 Sek.)* nur die gemessenen minimalen, maximalen und durchschnittlichen Verzögerungen während der Überlastphase. Wie erwartet, liegen in diesem Zeitraum die gemessenen durchschnittlichen Verzögerungen über den für den gesamten Zeitraum ermittelten Werten[8]. Dass die durchschnittlichen Verzögerungen innerhalb der zehnsekündigen Überlastphase dennoch moderat ausfallen, bezogen auf die gesamte Zeitraumspanne, zeigt, dass dennoch viele Rahmen des ACC Systems mit geringer Verzöge-

[7] Bezüglich der Motor-Steuerung wird auf eine Synchronisation verzichtet, da eine adäquate Umsetzung auch eine Anpassung des Regelungsalgorithmus erfordern würde.

[8] Dies ist wegen der logarithmischen Skala und des kleinen Wertebereichs in der Grafik leider nur schlecht erkennbar. In der Darstellung *CAN Overload (20-30 Sek.)* fehlt der Eintrag für den Bremsbefehl, da innerhalb dieses Beobachtungszeitraums kein entsprechender Befehl abgesetzt wurde.

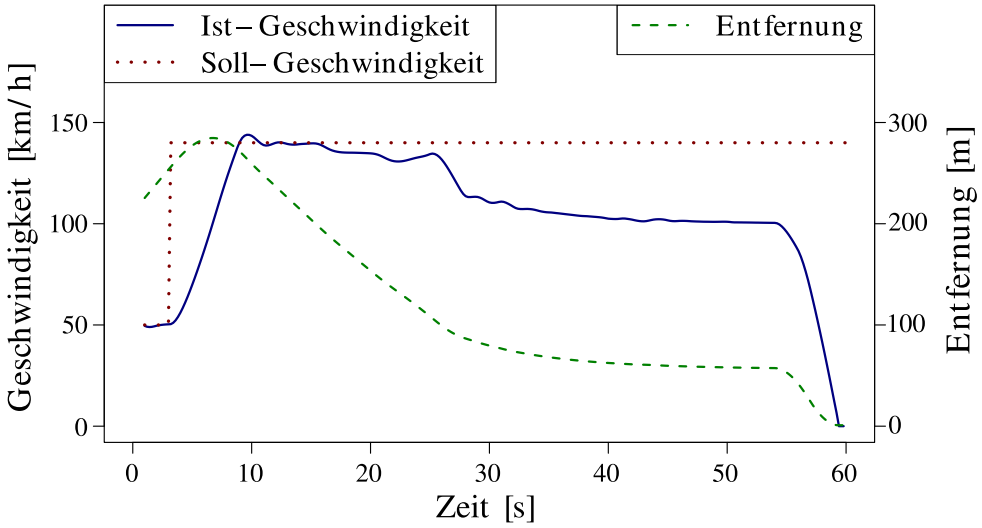


Abbildung 11.5: Auswirkungen auf die Regelgüte bei stark verzögerten Nachrichten.

rung übermittelt werden und vor allem die extremen Ausreißer für die Verschlechterung der Regelgüte verantwortlich sind – ein weiterer Indikator dafür, wie wichtig deterministische Garantien sind.

Kommen wir noch einmal auf die anderen konkreten Simulationssysteme mit den verschiedenen realen Kommunikationstechnologien zurück: Entsprechende Probleme wie bei CAN sind auch bei der Verwendung von Ethernet zu erwarten, sofern man hier eine ähnlich hohe prozentuale Auslastung anstrebt. Erschwerend kommt hinzu, dass bei Ethernet die Möglichkeit der Priorisierung fehlt, sodass eine alternative Zuordnung der Prioritäten hier keine Lösung bietet. Lediglich die zur Verfügung stehende höhere Bandbreite erlaubt – absolut betrachtet – eine höhere Auslastung, abhängig von den auftretenden Lastspitzen. Wir mussten an dieser Stelle auf vergleichende Simulationen verzichten, da sich das von ns-3 bereitgestellte Kanalmodell für Ethernet nicht für die Simulation von Überlastaspekten eignet[9]. Dementsprechend waren die Ergebnisse in diesen Fällen nicht repräsentativ und belastbar.

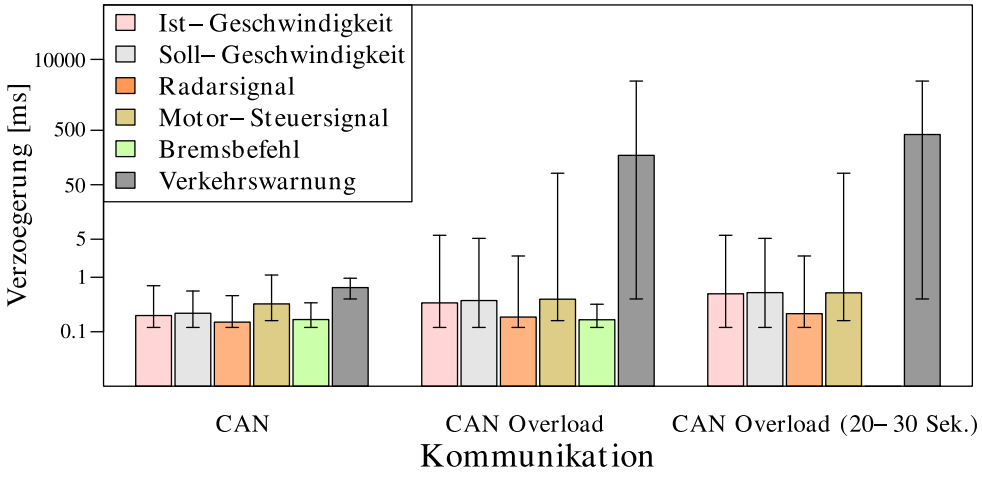


Abbildung 11.6: Durchschnittliche, minimale und maximale Verzögerung für jeden Nachrichtentyp des ACC Systems bei einer Überlastung des CAN-Busses.

[9]Gründe hierfür sind die unbeschränkten Nachrichtenpuffer innerhalb des simulierten Switches (kein Verlust von Rahmen

Bei ACOM kommt es zu einer gleichmäßigen Aufteilung der Bandbreite, sodass die Verzögerung der einzelnen Rahmentypen sehr homogen ausfällt. Deshalb sind die Beeinträchtigungen der Regelgüte deutlich weniger ausgeprägt, was unsere Vermutung unterstreicht, dass die extremen Ausreißer bei *CAN Overload* für deren Verschlechterung verantwortlich sind.

Dieses Simulationen illustrieren sehr deutlich die Probleme ereignisgetriggerter Kommunikationsprotokolle im Vergleich zu zeitgetriggerten Protokollen. Bei diesen kommt es aufgrund der exklusiven Slotreservierungen zu keinen Überlastsituationen zur Laufzeit. Falls die Bandbreite nicht ausreicht, wird dies bereits bei der Planung und der Zuteilung der Slots erkannt. Insofern ermöglicht die Verwendung von FlexRay in diesem Szenario die gleichzeitige Koexistenz zwischen den verschiedenen Assistenzsystemen, ohne dass es zu Einbußen bei der Regelgüte kommt. Während des Betriebs ermöglichen die exklusiven Reservierungen deterministische Garantien in Bezug auf die auftretenden Verzögerungen. Da die Ableitung eines konkreten Ablaufplans hier nicht Gegenstand der Untersuchungen ist und zudem die Kommunikationsanforderungen der anderen Assistenzsysteme nicht bekannt sind, beenden wir unsere Betrachtungen an dieser Stelle.

11.3 Zusammenfassung

Zusammenfassend zeigen die beiden Anwendungsszenarien, dass FERAL mit seinen Konzepten (Bridges, Gateways und die Beschreibung des Verhaltens mit der DBL) eine geeignete Plattform für das Virtual Prototyping komplexer verteilter eingebetteter Echtzeitsysteme bereitstellt. Hierbei unterstützt FERAL sowohl die Interaktion konkreter Simulationskomponenten unterschiedlicher Abstraktionsstufen miteinander als auch verschiedener Simulatoren, die deren funktionales Verhalten realisieren. Besonders hervorzuheben sind hierbei die Stärken im Bereich der Evaluation von Designalternativen, insbesondere in Bezug auf die Wahl von Kommunikationstechnologien sowie deren Austausch. Das ACOM-CSC erlaubt darüber hinaus in der Simulation die Berücksichtigung von Ressourcenengpässen bei der Kommunikation, noch bevor eine konkrete Technologie gewählt wird. Bridges und Gateways gestatten in späteren Entwicklungsschritten einen einfachen Austausch der ACOM-CSC gegen eine konkrete Kommunikationstechnologie innerhalb des Simulationssystems, ohne aufwändige Anpassungen.

Durch die Verwendung von Bridges und Gateways (für die Verbindung von FSCs und CSCs sowie CSCs mit anderen CSCs) erfordert die Evaluation der Designalternativen (am Beispiel des ACC) lediglich die Definition der konkreten Simulationssysteme, welche sich nur in Bezug auf die verwendeten CSCs sowie einem Verhaltensmodell voneinander unterscheiden. Da nahezu alle FSCs per Bridges an die CSCs angeschlossen sind, erfolgt die Anpassung des Kommunikationsverhaltens an die unterschiedlichen CSCs über die Bereitstellung entsprechender DBL-Spezifikationen. Die Mittel der DBL ermöglichen es, das Kommunikationsverhalten der einzelnen konkreten Simulationskomponenten jeweils für alle konkreten Simulationssysteme innerhalb einer DBL Spezifikation zu hinterlegen[10]. Lediglich das Verhaltensmodell des *ACC-Controller* musste für die unterschiedlichen CSCs modifiziert werden, da dort auf den Einsatz einer Bridge, zu Gunsten eines technologiespezifischen, hardwarenahen Treibers, verzichtet wurde. Daher unterscheiden sich die Szenarienbeschreibungen der konkreten Simulationssysteme kaum und lassen sich sehr einfach umsetzen.

In Bezug auf das ACC System liefert die Evaluation das Ergebnis, dass alle untersuchten Kommunikationstechnologien die funktionalen Anforderungen in Bezug auf die Regelgüte erfüllen, solange die Buslast nicht zu groß ist – auch wenn nur die Varianten mit FlexRay harte Echtzeitgarantien gewähren können. Gleichzeitig enthüllt das konkrete Simulationssystem *CAN Overload*, stellvertretend für andere

durch Überläufe) sowie die perfekte globale Sequentialisierung aller zu übertragenden Rahmen in der Reihenfolge der erteilten Sendeaufträge.

[10] An dieser Stelle muss einschränkend gesagt werden, dass *FlexRay Sync* eine eigene DBL-Spezifikation erfordert. Dies ist allerdings nicht verwunderlich, da hier zum Teil ein semantisch anderes Kommunikationsverhalten abgebildet wurde.

ereignisbasierte Kommunikationsverfahren, dass bei einer Überlast – und einer damit einhergehenden stark verzögerten Kommunikation zwischen den funktionalen Komponenten des ACC Systems – die Regelgüte sinken kann und daher eine genaue Analyse der Auslastung des Busses oder die Verwendung einer zeitgetriggerten Kommunikationstechnologie erforderlich wird.

12 Unterstützung für Hardware-in-the-Loop

Das von FERAL angestrebte Entwicklungsmodell sieht vor, konsequent die Vorteile von Virtual Prototyping und Simulationen in allen Entwicklungsphasen einzusetzen. Dies beginnt idealerweise in den frühesten Entwicklungsphasen, sodass einer der ersten Entwicklungsschritte bei einem neuen Projekt die Definition eines abstrakten Simulationssystems ist (Festlegen der FSCs, CSCs sowie deren abstrakten Interaktionen über Links). Anschließend erfolgt die Ableitung eines konkreten Simulationssystems durch Festlegung der Simulatoren und Zuweisung/Entwicklung erster, in der Regel, vereinfachter Verhaltensmodelle. Dieses Vorgehen gestattet es zudem, auch die Wahl von zukünftigen Designentscheidungen durch Simulationen und deren Ergebnisse zu stützen. Bei der weiteren Entwicklung werden die Verhaltensmodelle der FSCs schrittweise verfeinert und um neue Funktionalitäten erweitert. Die einheitlichen Schnittstellen der Simulationskomponenten ermöglichen die Interaktion von Simulationskomponenten unterschiedlicher Abstraktionsstufen innerhalb eines Simulationssystems. So entsteht durch jede Verfeinerung ein neues konkretes Simulationssystem, welches aufgrund dieses Umstands jederzeit simuliert werden kann. Dieses Vorgehen ermöglicht entwicklungsbegleitende Integrationstests mit FERAL und gestattet es so, Fehler früh zu identifizieren und zu beheben. Insbesondere schützt dieses Vorgehen auch davor, dass erst bei dem (abschließenden) Integrationstest Inkompatibilitäten in Bezug auf Schnittstelle, Interaktion und/oder Zielsetzungen aufgedeckt werden.

Aber nicht nur die FSCs können schrittweise verfeinert werden, auch eine schrittweise Annäherung der von FERAL simulierten Umgebung an die Realität ist umsetzbar, indem man beispielsweise mit dem abstrakten Kommunikationsmodell ACOM für eine CSC beginnt und dieses im Laufe der Entwicklung durch eine CSC ersetzt, welche die später verwendete Kommunikationstechnologie repräsentiert. Aber auch das simulierte Modell der physikalischen Umgebung kann um Details erweitert werden. Auch hierbei kann jeder Schritt mit Simulationen unterstützt werden, um Fehler früh zu finden und diese dadurch möglichst kostengünstig zu beheben. Die Reproduzierbarkeit der Simulationsergebnisse erlaubt des Weiteren, über die in Kapitel 9.6 angesprochenen Constraint-Checks automatische Regressionstests zu realisieren und auszuführen. Dieses Vorgehen führt insgesamt dazu, dass der Abstraktionsgrad schrittweise hin zur Integration reduziert wird.

Der letzte Entwicklungsschritt beim Entwurf eingebetteter Systeme stellt die Erzeugung von ausführbaren Artefakten dar, die im Anschluss auf die jeweiligen Steuergeräte (Hardwareplattform) aufgespielt werden. Dann wird das System in Betrieb genommen und abschließenden Tests unterzogen. In der Praxis können jedoch auch bei diesem Schritt weitere Fehler oder nicht vorhergesehene Verhaltensweisen auftreten, welche die Funktionalität oder das Verhalten des Gesamtsystems beeinflussen. Ursachen hierfür können, z. B. Verzögerungen bei der Bearbeitung eines Tasks sein oder hardwarespezifische Verzögerungen, welche bei der Simulation der funktionalen Modelle nicht oder nicht ausreichend detailliert berücksichtigt wurden.

Gerade bei reaktiven Systemen mit physikalischen Komponenten (z. B. Regelungssysteme) stellt sich die Analyse solcher Fehler sowie alleine die Durchführung der Tests aufgrund der erforderlichen Interaktion des Systems mit seiner Umwelt sehr schwierig und aufwändig dar. Zum einen sind die physikalische Umgebung und deren Einflüsse (Störgrößen) nur schwer deterministisch zu kontrollieren und daher Abläufe schlecht reproduzierbar, zum anderen können Fehler oder bestimmte Tests Schäden an den physikalischen Komponenten verursachen oder sich Gefahrensituationen ergeben. Auch das Debugging gestaltet sich schwierig, da physikalische Prozesse nicht beliebig angehalten und wieder aufgenommen werden können. Diese Probleme treten bei der reinen Simulation nicht auf, da die Umwelt Bestandteil des

Simulationssystems (z. B. in Form eines Matlab-Modells) ist und damit determinitisch und vollständig kontrollier- sowie reproduzierbar reagiert. Zudem kann die Simulation jederzeit angehalten und wieder aufgenommen werden (schrittweise Ausführung).

Aber auch bei dem letzten Schritt, der Überführung der funktionalen Komponenten auf reale Hardware(-plattformen), kann FERAL eingesetzt werden, um die Analyse von Fehlern sowie Tests zu vereinfachen. Um dies zu ermöglichen, muss FERAL in der Lage sein, nicht nur das funktionale Verhalten, sondern auch die hardwarespezifischen Eigenschaften der verwendeten Zielplattform bei der Simulation zu berücksichtigen. Um dies zu ermöglichen, bietet FERAL zwei unterschiedliche prototypische Ansätze an, die einander ergänzen und sowohl in Kombination als auch einzeln angewendet und sinnvoll genutzt werden können: eine taktgenaue Simulation der eingesetzten Plattform und ihrer Hardware sowie die direkte Ausführung der funktionalen Komponente auf der Zielplattform, in Form einer Hardware-in-the-Loop (HiL) Lösung [FFHS06]. In beiden Fällen übernimmt FERAL die Simulation der Umgebung[1] (dies können sowohl andere FSCs und CSCs als auch ein physikalisches Modell der Umwelt, z. B. einer Regelstrecke, sein).

Für die taktgenaue Simulation der jeweiligen eingesetzten Hardwareplattform wurde Synopsis [Rey13] durch die Arbeitsgruppe Wehn der TU Kaiserslautern in FERAL integriert. Basierend auf physikalischen Modellen der verwendeten Hardwareplattform gestattet dies die taktgenaue Simulation der Ausführung beliebiger Anwendungen (mit oder ohne Betriebssystem) und erlaubt so eine realistische Simulation der Funktionalität auf der Hardwareplattform innerhalb der simulierten Systemumgebung. Ein Nachteil ist die hohe Laufzeit für die Simulation derart detaillierter Modelle sowie der Aufwand, um diese Modelle für die jeweilige Plattform zu entwickeln. Wir konzentrieren uns daher auf den zweiten Ansatz, HiL-Simulationen.

Die Zielsetzung dieses Kapitels ist zweigeteilt. Zunächst stellen wir unsere prototypische Realisierung von HiL-Simulationen mit FERAL für die *Imote 2*-Plattform vor (Kapitel 12.2). Anschließend demonstrieren wir in Kapitel 12.3 die Entwicklung eines verteilten vernetzten Regelungssystems eines inversen Pendels auf Basis von SDL mit Hilfe von Virtual Prototyping durch die schrittweise Verfeinerung der definierten konkreten Simulationssysteme, wobei unter anderem, auch die zuvor vorgestellte HiL-Simulation zum Einsatz kommt. Zunächst betrachten wir jedoch die allgemeinen Ideen und Konzepte für HiL-Lösungen (Kapitel 12.1).

Die Ergebnisse dieses Kapitels wurden in [12] publiziert.

12.1 Konzepte und Ausgangssituation

Die Wurzeln der HiL-Simulation liegen in der zivilen und militärischen Luftfahrt. HiL-Simulationen werden sowohl für die Entwicklung von Lenkwaffen [BJL+01] als auch in der zivilen Flugzeugentwicklung [OPA11] eingesetzt. In beiden Fällen werden HiL-Simulationen durch die im Vergleich zu realen Tests geringen Kosten und die Vermeidung von Gefahrensituationen motiviert. Daher wurde in diesen Bereichen die Funktionalität von Steuergeräten schon früh mit Hilfe von simulierten Umgebungsmodellen getestet [OPA11]. Durch die zunehmende Komplexität aktueller eingebetteter Systeme und deren Sicherheitsrelevanz erweisen sich HiL-Simulationen auch in anderen Anwendungsbereichen als sinnvoll und gehören mittlerweile zu den etablierten Techniken, um Entwicklungszeiten und Kosten zu reduzieren [PB15]. So setzt die Automobilindustrie [BKPS07, FFHS06, HRP+06, BRSR99] HiL-Simulationen mittlerweile als Basistechnologie für die Entwicklung und Ausführung automatisierter Tests von komplexen Funktionen, Steuerungen und Regelungen ein. Aber auch in anderen Anwendungsgebieten, wie z. B. der Entwicklung von Regelungen für Windenergieanlagen [PDB08], kommen HiL-Simulationen zum Einsatz. Insgesamt haben sich HiL-Simulationen aufgrund der Komplexität aktueller Steuergeräte in nahezu allen Bereichen etabliert. Daher gibt es viele kommerzielle Anbieter und Produkte, unter anderem

[1] Hier kommt wiederum FERALs Fähigkeit zum Zuge, in einem Simulationssystem Simulationskomponenten mit unterschiedlichem Abstraktionsgrad verwenden zu können.

von MathWorks, dSpace, National Instruments oder Vector Informatik GmbH, welche HiL-Simulationen unterstützen und sowohl Hardware- als auch entsprechende Software-Lösungen anbieten.

Bei der HiL-Simulation wird die zu testende funktionale Komponente oder Teilfunktionalität direkt auf der späteren Zielplattform[2] ausgeführt. Die zu testende funktionale Komponente (inkl. der Zielhardware) wird in der Literatur häufig als DUT (Device-under-Test) bezeichnet [VDE12]. Bei der HiL-Simulation werden die Ein- und Ausgaben des DUT von einem möglichst realistischen Modell der Umgebung geliefert bzw. verarbeitet. Dieses Modell kann sowohl ein physikalisches Modell der Umwelt sein – wie z. B. bei einer Regelung ein mathematisches Modell der Regelstrecke – oder aber auch die Simulation des Verhaltens anderer funktionaler Komponenten umfassen, mit denen das DUT interagiert.

Die Vorteile dieses Ansatzes gegenüber einem vollständigen physikalischen Aufbau liegen unter anderem in einer Reduktion der Kosten und Aufwände sowie einer verbesserten Reproduzierbarkeit der Ergebnisse. Zudem ist es nicht notwendig zu warten, bis alle Komponenten soweit fertiggestellt sind, dass diese auf einer realen Hardware ausgeführt werden können bzw. deren Hardware verfügbar ist. Dies gilt natürlich auch für das physikalische Zielsystem. In [BRSR99] wird beschrieben, wie eine Motorsteuerung auf Basis eines physikalischen Modells eines neuen Motors entwickelt wird, noch bevor dieser fertiggestellt ist. Ein weiterer Vorteil eines physikalischen Modells der Strecke bzw. Umgebung besteht in der Vermeidung realer Schäden bei Fehlfunktionen bzw. deren Beschränkung auf das DUT. Insbesondere wenn die Funktionalitäten des DUT sicherheitsrelevant sind, bieten HiL-Simulationen eine sichere Möglichkeit, das System unter realistischen Bedingungen zu testen, ohne hierbei eine reale Gefährdung zu verursachen.

Die Abbildung 12.1 zeigt den konzeptionellen Aufbau eines Simulationssystems anhand eines einfachen Regelkreises, dessen Controller mittels HiL-Simulation evaluiert werden soll. Controller, Aktuator und Sensor sind jeweils Softwarekomponenten, welche die entsprechenden Funktionalitäten implementieren. Die Regelstrecke (Controlled System) besteht aus einem mathematischen Modell des zu regelnden physikalischen Systems. In dem gezeigten Beispiel wird das Verhalten von Aktuator, Sensor und Regelstrecke simuliert, während der Controller bereits auf der finalen Hardware ausgeführt wird. ECU[3] (Hardwareplattform) und Controller (Softwarekomponente) bilden zusammen das DUT. Die Eingabestimuli des Controllers werden direkt von den simulierten Komponenten (hier von dem Sensor) über eine entsprechende Schnittstelle zwischen Simulator und DUT vorgegeben. Ebenso werden die Ausgaben des DUT (z. B. neue, durch den Controller berechnete Steuergrößen) wieder in den Simulator eingespeist und an die jeweiligen simulierten Zielkomponenten weitergeleitet (hier an den Aktuator).

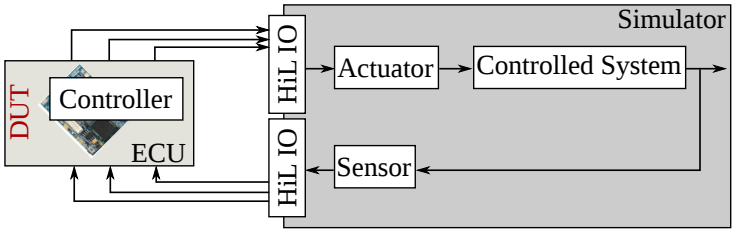


Abbildung 12.1: Beispiel für die Struktur eines einfachen Simulationssystems mit HiL.

Dadurch, dass bei einer HiL-Simulation die Ein- und Ausgaben des DUT durch einen Simulator vorgegeben bzw. verarbeitet werden, können Testläufe sehr einfach aufgezeichnet, ausgewertet und nach erfolgten Korrekturen und Modifikationen wiederholt werden. Zudem können die Ergebnisse zwischen den Ausführungen miteinander verglichen werden, da sich die simulierte Umgebung, im Gegensatz zur

[2] Alternativ bieten verschiedene Hersteller auch spezielle Hardwarelösungen für HiL-Simulationen an. In diesem Fall können jedoch unter Umständen nicht alle Einflussgrößen der späteren Zielplattform korrekt berücksichtigt werden.

[3] Electronic Control Unit

realen Umwelt, deterministisch verhält und keine unbekannten Störeinflüsse auftreten. Da sämtliche Ein- und Ausgaben des DUT über den Simulator erfolgen, lassen sich automatisierbare Testsuites definieren. Der Ablauf der Tests kann durch den Simulator überwacht und die Auswertung mit den Constraint-Checks (vgl. Kapitel 9.6) automatisiert werden. Wie bei klassischen Unittests können diese, nach der Behebung von Fehlern oder bei einer neuen Revision, erneut ausgeführt und hierdurch eine nachhaltige Form der Qualitätssicherung betrieben werden.

Die Abbildung 12.1 zeigt nur ein mögliches Beispiel für den Einsatz von HiL-Simulationen. So ließe sich in dem gezeigten Szenario anstelle des Controllers auch jede andere Softwarekomponente als DUT umsetzen und auf die gleiche Weise evaluieren. Auch die schrittweise Ersetzung der verbleibenden simulierten Softwarekomponenten durch ECUs wäre denkbar. Ein anderer Ansatz in Bezug auf HiL-Simulationen stellt die Variante dar, lediglich die funktionale Komponente innerhalb des Simulators auszuführen und anstelle eines Modells der Regelstrecke direkt mit der realen physikalischen Umwelt zu interagieren[4]. Diese Variante eignet sich z. B. für die Dimensionierung und Evaluation von Reglern und spart die Entwicklung auf einem eingebetteten System, welches in Bezug auf Debugging weniger komfortabel ist als eine PC-Umgebung bzw. der Simulator. Dieser Ansatz wird von Popovici et al. [PM12] als Rapid Control Prototyping (RCP) bezeichnet und erfordert eine Simulation des Reglers in Echtzeit, da das physikalische System das Voranschreiten der Zeit vorgibt.

Ein Beispiel für eine solche Anwendung wird in [Ala13] beschrieben, in dieser Arbeit wird der Regler für ein inverses Pendel mittels Matlab Simulink entworfen. Als Regelstrecke werden ein reales physikalisches Pendel sowie dessen Sensoren und Aktuatoren verwendet, während der Regler selbst mit Matlab Simulink simuliert wird. Die Anbindung des in Matlab simulierten Reglers an die Sensoren bzw. den Aktuator erfolgt mittels UART und einer speziell für diese Anwendung entwickelten Schaltung, welche die erforderlichen Signalwandlungen (Analog-Digital-Converter und Digital-Analog-Converter) vornimmt. Neben dieser speziellen Lösung existieren von Mathworks mit Simulink Real-Time und SimDriveline aber auch kommerzielle Lösungen, die diese Art von HiL-Simulationen direkt unterstützen.

Unabhängig von der gewählten Variante werden geeignete Schnittstellen zwischen Simulator und DUT bzw. der realen Systemumgebung benötigt. Die Realisierung der HiL-IO Schnittstelle ist hierbei abhängig von dem Simulator, dem Szenario sowie der verwendeten Zielplattform. Die mögliche Bandbreite erstreckt sich über die beschriebene Anbindung mittels UART, über USB und Ethernet, bis hin zu Feldbussen wie CAN oder FlexRay. Der Einsatz von Feldbussen bietet sich etwa dann an, wenn hierfür die Kommunikationstechnologie eingesetzt werden kann, die später auch für die Kommunikation innerhalb des verteilten Systems genutzt wird. In diesem Fall muss der Simulator in der Lage sein, den simulierten Softwarekomponenten die Kommunikation über den physikalischen Bus[5] zu ermöglichen. Weitere Kriterien bei der Auswahl der HiL-IO Schnittstelle sind die auftretenden Latenzen sowie die benötigte Bandbreite.

Die größte Schwierigkeit bei HiL-Simulationen (aber auch bei RCP) besteht in der Synchronisation von Simulator und DUT bzw. realer Systemumgebung, die während des gesamten Simulationslaufes aufrechtzuerhalten ist [PI00]. Aus dieser Notwendigkeit ergeben sich im Allgemeinen harte Echtzeitanforderungen an den Simulator, aber auch an die verwendete HiL-IO Schnittstelle. Wird beispielsweise der Controller simuliert und eine reale Regelstrecke verwendet, gibt diese den zeitlichen Ablauf der Simulation vor. Die Simulation muss in Realzeit laufen, und auch die HiL-IO Schnittstellen müssen hinreichend schnell arbeiten, um den Regler mit neuen Sensorwerten zu versorgen bzw. die Steuergröße dem Aktuator rechtzeitig zur Verfügung zu stellen. Wenn man, wie in Abbildung 12.1 dargestellt, den Controller als DUT auslegt und alle anderen Komponenten simuliert (inkl. der Regelstrecke), so gelten zunächst einmal die gleichen Grundsätze. Da jedoch kein physikalisches System involviert ist, bestehen

[4] Häufig handelt es sich hierbei um ein klassisches Regelungssystem, bei dem der Regler simuliert und die Regelstrecke als reales physikalisches Konstrukt vorliegt und in Echtzeit kontrolliert wird.

[5] Dieser Fall wird häufig als Restbussimulation bezeichnet, auch hier bieten Hersteller, wie Vector (CANalzyer, CANoe) oder IXXAT, entsprechende Lösungen an.

mehr Möglichkeiten, Simulation und DUT zu synchronisieren. Beispielsweise sorgt eine Reduktion des Taktes der eingesetzten Hardwareplattform dafür, dass der Simulation mehr Zeit für eigene Berechnungen und Abläufe zur Verfügung steht. Je nach Eingriffsmöglichkeiten in die Zielplattform kann auch der Simulator vollständig die Zeit auf der Zielplattform kontrollieren und die Ausführung der Anwendung pausieren, um eine regelmäßige Synchronisation von Simulator und DUT umzusetzen.

12.2 Unterstützung für HiL-Simulationen in FERAL

Um den Entwickler bei der Überführung der funktionalen Komponenten auf die jeweilige Zielplattform zu unterstützen, haben wir untersucht, wie FERAL sich um eine Schnittstelle für HiL-Simulationen erweitern lässt, und eine prototypische Implementierung für die *Imote 2*-Plattform umgesetzt. Hierbei haben wir uns zunächst auf konzeptionelle Probleme konzentriert und diese für FSCs, welche auf SDL als Verhaltensspezifikation basieren, gelöst. Das prinzipielle Vorgehen ist jedoch auch auf andere Plattformen und FSC-Typen – mit gewissen Einschränkungen – übertragbar.

Wie erläutert, besteht die grundlegende Schwierigkeit darin, DUT und FERAL – beziehungsweise die von FERAL simulierten FSCs und CSCs – synchron zu halten. Dies wird durch den Umstand erschwert, dass die von FERAL benötigte Ausführungszeit für einen simulierten Zeitschritt unter anderem von der Komplexität und Größe des simulierten Systems sowie den verwendeten Simulatoren abhängt. Dementsprechend kann FERAL bei der Ausführung des Simulationssystems keine Garantien bzgl. der Echtzeitfähigkeit gewähren. Daher haben wir uns für einen Ansatz entschieden, bei dem FERAL die Zeit bzw. deren Voranschreiten auf dem DUT kontrolliert.

Die für die Verwendung der HiL-Simulationen entstehenden Aufwände sind bei einem konsequenten Einsatz von FERAL als gering einzustufen, weil das Simulationssystem bzw. die Szenarienbeschreibung bereits aus den früheren Entwicklungsphasen zur Verfügung steht. Die notwendigen Anpassungen sind minimal, da die entwickelte HiL-Schnittstelle es dem DUT ermöglicht, mit den in FERAL simulierten FSCs und CSCs, wie gewohnt, über Ports und Links zu interagieren. Das heißt, bei der Umwandlung einer FSC in eine DUT bleibt die Schnittstelle für den Austausch von Daten mit den anderen Simulationskomponenten unverändert. Um dies zu realisieren, haben wir spezielle HiL-Stubs entwickelt, welche das DUT innerhalb des Simulationssystems repräsentieren. Über diese kann FERAL die Ausführung des DUT kontrollieren sowie Nachrichten mit diesem austauschen. Für FERAL sowie alle anderen Simulationskomponenten verhält sich der HiL-Stub transparent, d.h. wie jede andere FSC. Das heißt, er implementiert die `ConcreteSimulationComponent`-Schnittstelle, wird durch Direktoren ausgeführt und interagiert mit anderen Simulationskomponenten über den Austausch von Nachrichten. Wir bieten zwei Varianten der HiL-Stubs mit unterschiedlichem MOCCs an (zeitgetriggert und ereignisgetriggert).

Der HiL-Stub bildet zusammen mit seinem DUT die HiL-Simulationskomponente (HSC). Das DUT umfasst die Zielplattform samt der darauf ausgeführten funktionalen Softwarekomponente (FC), welche das Verhalten implementiert, sowie der für die Ausführung der FC benötigten Laufzeitumgebung (HiL-Runtime). Die logische Schnittstelle zwischen FERAL und einer HSC besteht, wie bei jeder anderen Simulationskomponente, aus einer Menge von Input- und OutputPorts. Diese wird innerhalb von FERAL durch die für das DUT erzeugte Instanz des HiL-Stub bereitgestellt, sodass bei einer Umwandlung einer FSC in eine HSC eine Anpassung der Szenariobeschreibung in Bezug auf die Links zwischen den Simulationskomponenten nicht erforderlich ist. Ein Nachteil dieses Ansatzes ist, dass auf der Zielplattform eine minimale Laufzeitumgebung (HiL-Runtime) für die Ausführung der eigentlichen FC – welche bei Anwendung unseres Entwicklungsmodells vorher als Simulationskomponente in FERAL simuliert wurde – benötigt wird. Die HiL-Runtime ermöglicht den Transfer von Ein- und Ausgabedaten an die Ports des instanziierten HiL-Stubs und erlaubt FERAL indirekt über dessen Methoden die Steuerung der Ausführung der FC auf dem DUT. Gleichzeitig ist die HiL-Runtime zentraler Bestandteil des Synchronisationsmechanismus zwischen FERAL und dem DUT (vgl. Kapitel 12.2.2 und 12.2.3).

Die von uns entwickelte HiL-Runtime unterstützt Verhaltensmodelle auf Basis von SDL-Spezifikationen auf der Zielplattform *Imote 2*. Der Grund für unsere Auswahl von SDL liegt darin, dass sowohl die Laufzeitumgebung SDLRe für SDL-Systeme als auch das entwickelte Environment Framework SEnF [FGW06, FGJ+05] bereits für den *Imote 2* angepasst vorliegen und auch unser Transpiler ConTraST die *Imote 2*-Plattform unterstützt. Auch die bereits existierende Anbindung von SDL an FERAL sowie die vorhandenen Anpassungen von SEnF erleichterten die Integration wesentlich (vgl. Kapitel 8.4). In diesem konkreten Fall übernimmt die HiL-Runtime die Kommunikation mit dem HiL-Stub, steuert die Ausführung des SDL-Systems gemäß den Vorgaben von FERAL und bildet SDL-Signale auf Nachrichten- und Nachrichtendatentypen von FERAL ab und umgekehrt. Hierbei können die in Kapitel 8.4 beschriebenen Konzepte wiederverwendet werden, da die HiL-Runtime die notwendigen Abstraktionen zur Verfügung stellt. Kapitel 12.2.3 beschäftigt sich mit dem Aufbau der HiL-Runtime, zunächst konzentrieren wir uns jedoch auf die Anbindung zwischen HiL-Runtime und HiL-Stub.

12.2.1 Anbindung von HiL-Simulationskomponenten an FERAL

Um FERAL die Kontrolle über die Ausführung der DUT zu ermöglichen, implementiert der HiL-Stub wie jede andere konkrete Simulationskomponente die `ConcreteSimulationComponent`-Schnittstelle. Um die Ausführung einer HSC zu steuern, nutzt FERAL die in der Schnittstelle definierten Methoden, deren Aufrufe zunächst in Steueranweisungen übersetzt werden. Der instanziierte HiL-Stub gibt die Steueranweisungen nach einer Vorverarbeitung an die HiL-Runtime auf dem DUT weiter. Diese steuert dann die Ausführung der FC, gemäß den Vorgaben von FERAL.

Die Interaktion erfolgt – ebenso wie bei den anderen Simulationskomponenten – über den Austausch von Nachrichten zwischen Ports. Die erforderlichen Ports stellt der instanziierte HiL-Stub als Stellvertreter der auf dem DUT ausgeführten FC zur Verfügung. Eingehende Nachrichten werden durch den HiL-Stub mittels ASN.1 kodiert und dann an die HiL-Runtime übermittelt. Diese ist dafür verantwortlich, die Nachrichten zu dekodieren und diese der FC in geeigneter Form zur Verfügung zu stellen. Entsprechend ist es umgekehrt die Aufgabe der HiL-Runtime, Daten von der FC entgegenzunehmen und diese ASN.1-kodiert an den instanziierten HiL-Stub zu übertragen. Eine wesentliche Aufgabe der HiL-Runtime besteht darin, das Kommunikationsmodell von FERAL – welches auf dem Austausch von Nachrichten zwischen Ports basiert – auf das von der FC unterstützte Kommunikationsmodell abzubilden.

Um den Austausch von Nachrichten und Steueranweisungen zwischen HiL-Stub und HiL-Runtime möglichst flexibel zu realisieren, wurde ein minimalistisches Übertragungsprotokoll mit einem eigenen HiL-Rahmenformat entworfen. Die HiL-Rahmen werden über TCP/IP zwischen dem HiL-Stub und einem Gateway übertragen, welcher die HiL-Rahmen über die serielle Schnittstelle an die UART-Schnittstelle des *Imote 2*-Knotens weiterreicht. Das Übertragungsprotokoll stellt sicher, dass Verluste oder Verfälschungen erkannt und durch eine erneute Übertragung korrigiert werden. Dieses Vorgehen erlaubt es, den *Imote 2*-Knoten beliebig zu positionieren. So kann z. B. die Simulation durch FERAL auf einem leistungsstarken Server ablaufen, während die Hardware in Form des *Imote 2*-Knotens an einem beliebigen lokalen Arbeitsplatz angeschlossen ist.

Abbildung 12.2 illustriert dies anhand des bereits in Kapitel 12.1 präsentierten Regelungssystems. Bei dieser HiL-Simulation wird der Controller (Regler) als DUT realisiert und evaluiert. Actuator (Aktuator), Controlled System (Regelstrecke) und Sensor sind als FSCs mit unterschiedlichen Modellierungstechniken realisiert und werden von FERAL simuliert. Während die Regelstrecke mittels Simulink beschrieben und der Sensor direkt in Java modelliert wurde, kam für Aktuator und Regler SDL zum Einsatz. Der Controller wird bei der HiL-Simulation auf einem *Imote 2*-Knoten als FC ausgeführt. FERAL steuert die Ausführung des DUT über den instanziierten HiL-Stub, über den gleichzeitig die Interaktion mit den anderen konkreten Simulationskomponenten realisiert wird. Der instanziierte HiL-Stub des Reglers leitet Steueranweisungen des zuständigen Direktors wie auch Nachrichten der anderen FSCs an das DUT bzw. dessen HiL-Runtime weiter. Der verwendete Kommunikationspfad ist in der Abbildung rot dargestellt und verläuft zunächst von

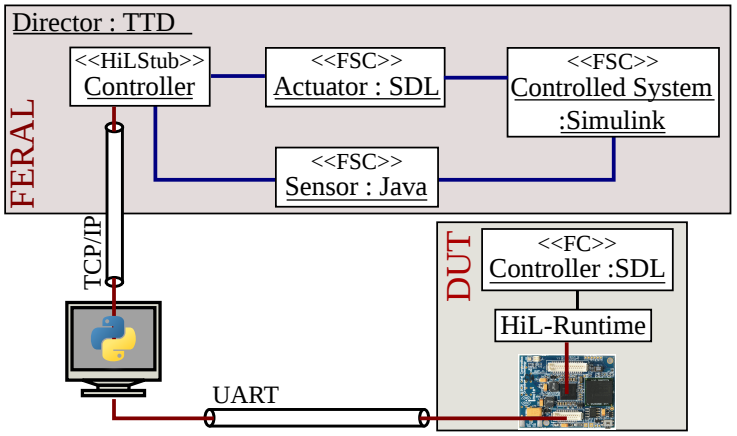


Abbildung 12.2: HiL-Simulation eines Regelkreises mit dem Regler als DUT.

dem HiL-Stub des Reglers über TCP/IP zu einem Gateway, an den das DUT über UART angeschlossen wurde. Die blauen Verbindungen zwischen den einzelnen Simulationskomponenten stellen Links[6] für die Interaktion und den Austausch von Nachrichten verwendet werden innerhalb von FERAL dar.

12.2.2 HiL-Stub – Aufbau und Architektur

Der HiL-Stub ermöglicht es FERAL, eine HSC wie jede andere konkrete Simulationskomponente auszuführen und über die Input- und OutputPorts die Interaktion der ausgeführten FC auf DUT mit den anderen Simulationskomponenten des Systems. Aktuell existiert sowohl ein zeit- als auch ein ereignisgetriggerter HiL-Stub, sodass derjenige verwendet werden kann, dessen MOCC mit dem der FC übereinstimmt. Wir konzentrieren uns auf die Darstellung der zeitgetriggerten Variante und skizzieren am Ende des Kapitels kurz die Unterschiede zur ereignisgetriggerten Variante.

Die Klassen `TimeTriggeredHiLStub` und `EventTriggeredHiLStub` realisieren den zeit- bzw. ereignisgetriggerten HiL-Stub und implementieren, wie alle anderen konkreten Simulationskomponenten, die `ConcreteSimulationComponent`-Schnittstelle. Über diese Schnittstelle steuert FERAL die Ausführung der FC auf dem DUT.

Das UML-Diagramm 12.3 zeigt die interne Struktur und den Aufbau des HiL-Stubs und gilt sowohl für die zeit- als auch die ereignisgetriggerte Variante. Jeder HiL-Stub verfügt über ein `RemoteControlInterface`, welches die Methoden für die Interaktion zwischen HiL-Stub und HiL-Runtime des HSC definiert. Die angebotenen Methoden erlauben es FERAL, die Ausführung der FC mittels der HiL-Runtime auf dem DUT zu steuern (`rebootNode` und `Run`). Weitere Methoden ermöglichen den Austausch von Nachrichten zur Interaktion zwischen der FC und anderen Simulationskomponenten. Hierfür werden sämtliche Methodenaufrufe auf dem `RemoteControlInterface` zunächst in Steueranweisungen übersetzt und in Form von HiL-Rahmen (vgl. Anhang E) an die HiL-Runtime übertragen. Wir widmen uns zuerst der Interaktion zwischen HSC und den Simulationskomponenten in FERAL. Da der HiL-Stub eine generische Komponente ist und beliebige DUTs unterstützt, wird erst bei der Instanziierung des HiL-Stubs ermittelt, welche Input- und OutputPorts dieser für die Interaktion mit anderen Simulationskomponenten bereitstellen soll. Hierfür bietet das `RemoteControlInterface` die Methoden `getInput-` bzw. `getOutputPorts` an, um die benötigten Ports von der HiL-Runtime zu erfragen. Die Ermittlung der benötigten Interaktionsports ist Aufgabe der HiL-Runtime und Teil des Aspekts der Vermittlung zwischen dem Kommunikationsmodell von FERAL und der ausgeführten FC der HSC. Mit der Methode

[6] Auf eine Darstellung der einzelnen Input- und OutputPorts haben wir aus Gründen der Übersichtlichkeit verzichtet.

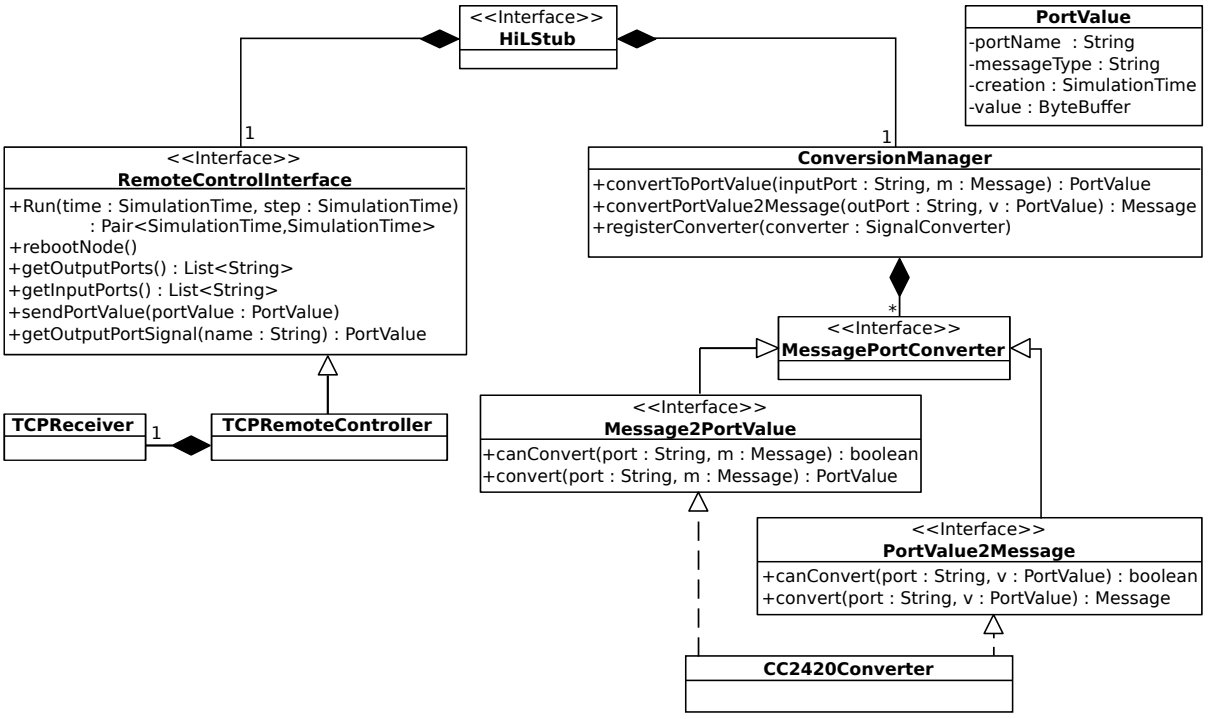


Abbildung 12.3: UML-Diagramm des internen Aufbaus des HiL-Stubs.

`sendPortValue` können die Nachrichten eines InputPorts des HiL-Stubs an die HiL-Runtime übertragen werden. Hierzu werden die Nachrichten zunächst in Objekte der Klasse `PortValue` konvertiert. Dieses Objekt wird beim Aufruf der Methode `sendPortValue` mittels ASN.1 serialisiert und dann als HiL-Rahmen an die HiL-Runtime übermittelt. Die Methode `getOutputPortSignal` realisiert die Kommunikation in umgekehrter Richtung; die HiL-Runtime kodiert die von der FC bereitgestellten Daten mittels ASN.1 und überträgt diese an den HiL-Stub. Dieser deserialisiert die empfangenen Daten wiederum zu einem `PortValue`-Objekt, und erzeugt aus diesen Nachrichten für die OutputPorts des instanziierten HiL-Stubs.

Die Objekte der Klasse `PortValue` dienen als Container und speichern alle relevanten Informationen einer Nachricht (`Message`) aus FERAL. Dazu gehört der Name des Ports, an den die Nachricht adressiert ist, der Typ der Nachricht (Klassenname), der Zeitpunkt, zu dem die jeweilige Nachricht erzeugt wurde, sowie ein ASN.1-codierter Datenstrom mit den in der Nachricht enthaltenen Nachrichtendaten (`MessageData`, vgl. Abbildung 8.2). Die Konvertierung zwischen den Nachrichten der Klasse `Message` von FERAL und den Objekten der Klasse `PortValue` erfolgt durch den `ConversionManager`. Dieser enthält eine beliebige Anzahl `MessagePortConverter`, die für die Umwandlung verantwortlich sind. Dieses modulare Vorgehen bietet sich an, da Simulatoren eigene Nachrichten- und Nachrichtendatentypen einführen können. Über die Bereitstellung eigener `Message-PortConverter` können diese Typen ebenfalls unterstützt werden. Hierfür genügt es, einen neuen `MessagePortConverter` beim `ConversionManager` zu registrieren. Ein Beispiel hierfür ist `CC2420Converter`, dieser unterstützt die CC2420-Nachrichtentypen der ns-3-CSC.

Für die Serialisierung der Nachrichtendaten verwenden wir ASN.1 anstatt der Standardmethoden der JVM, da diese auf dem *Imote 2* nicht zur Verfügung stehen. ASN.1 dagegen stellt eine etablierte und weit verbreitete Kodierung für den Datenaustausch dar. Insgesamt kommt ASN.1 zweimal zum Einsatz: Zunächst, um die innerhalb einer Nachricht enthaltenen Nachrichtendaten zu serialisieren,

anschließend wird das gesamte `PortValue`-Objekt für die Übertragung zur HiL-Runtime noch einmal unter Verwendung von ASN.1 serialisiert.

Bei der Instanziierung des HiL-Stubs wird diesem die IP-Adresse und der Port übergeben, unter dem die entsprechende HiL-Runtime erreichbar ist. Als erster Schritt wird das DUT über einen Reset zurückgesetzt, um einen definierten Ausgangszustand zu erreichen. Dann ermittelt der HiL-Stub die benötigten Input- und OutputPorts über die Methoden des `RemoteControlInterface` und legt diese an.

Beim zeitgetriggerten MOCC wird vor jedem Aufruf der `fire`-Methode einer Simulationskomponente zunächst die `prefire`-Methode ausgeführt. Der HiL-Stub konvertiert in dieser Methode die Nachrichten aller InputPorts in `PortValue`-Objekte und übermittelt diese an die HiL-Runtime. Diese speichert die Nachrichten zunächst nur. Beim Aufruf der `fire`-Methode prüft der HiL-Stub zuerst, ob die lokale Zeit des DUT gegenüber der aktuellen Zeit des Simulators plus der Schrittweite in der Zukunft liegt. Ist dies der Fall, erfolgt in diesem Zeitschritt keine Ausführung der FC. Auf diese Weise erfolgt eine schrittweise Resynchronisation der Zeit zwischen FERAL und dem FC, falls notwendig. Ist dies nicht der Fall, wird die Methode `Run` des `RemoteControlInterface` aufgerufen. Dieser wird als Parameter die aktuelle Simulationszeit sowie die Länge des Zeitschrittes übergeben. Die HiL-Runtime berechnet auf Grundlage der lokalen Zeit des DUT die noch für diesen Ausführungsschritt verbleibende Zeitspanne und startet die Ausführung der FC für diese Spanne. Im Idealfall endet die Ausführung zu dem vorgegebenen Zeitpunkt, allerdings kann je nach Implementierung der FC und deren Laufzeitumgebung die Ausführung auch später enden. Bei SDL sieht die Laufzeitumgebung z. B. nur eine Unterbrechung nach einer vollständigen Transition vor.

Die `Run`-Methode liefert neben der aktuellen Zeit des DUT auch den nächsten Zeitpunkt zurück, zu dem das FC wieder ausgeführt werden muss – sofern der HiL-Stub vorher keine Nachrichten anderer Simulationskomponenten empfängt. Die Rückgabe der aktuellen Zeit des DUT ermöglicht eine Resynchronisation mit FERAL auf die beschriebene Weise, falls das DUT gegenüber FERAL vorausgeeilt ist und dessen Zeit daher in der Zukunft liegt.

Danach erfolgt der Aufruf der Methode `postFire`, in der der HiL-Stub prüft, ob Nachrichten von der FC für FERAL in der HiL-Runtime bereitstehen. Ist dies der Fall, werden diese zunächst serialisiert und dann übertragen. Jedes `PortValue`-Objekt enthält einen Zeitstempel, welcher den lokalen Zeitpunkt des DUT enthält, zu dem die Nachricht erzeugt wurde. Der instanziierte HiL-Stub leitet nur die Nachrichten an seine OutputPorts weiter, deren Generierungszeitstempel δ kleiner oder gleich der aktuellen – durch den Direktor vorgegebenen – Simulationszeit τ am Ende des Ausführungsschrittes ist. Nachrichten, deren Generierungszeitstempel in der Zukunft liegen, speichert der HiL-Stub zwischen, um eine Verletzung der Kausalität zu vermeiden. Die gespeicherten Nachrichten werden so lange verzögert, bis die Simulationszeit den Generierungszeitpunkt der jeweiligen Nachricht erreicht hat, d.h. $\delta \leq \tau$.

Bei dem ereignisgetriggerten HiL-Stub wird die `fire`-Methode durch den Direktor nicht für die Dauer eines festen Zeitschritts und periodisch ausgeführt, sondern nur, wenn ein Ereignis vorliegt. Die Ausführungslogik bei ereignisgetriggerten HiL-Stubs unterscheidet sich ansonsten nur in Details von der Logik zeitgetriggerter HiL-Stubs. Ist ein Ereignis (empfangene Nachricht oder ein abgelaufener Timer) aufgetreten, ruft der Direktor die entsprechende für diesen Ereignistyp zuständige Methode des HiL-Stub auf. Handelt es sich um eine Nachricht, wird diese wie bei der zeitgetriggerten Variante an die HiL-Runtime übermittelt und dort gespeichert. Bevor der HiL-Stub ausgeführt wird, erfolgt, wie bei dem zeitgetriggerten HiL-Stub, die Prüfung, ob die aktuelle Simulationszeit des DUT in der Zukunft liegt. In diesem Fall wird die Ausführung solange verzögert, bis die Simulationszeit von FERAL die des DUT *eingeholt* hat. Hierfür kann beim Direktor ein entsprechender Timer aufgezogen werden, welcher den HiL-Stub zu dem entsprechenden Zeitpunkt erneut ausführt. Andernfalls führt die HiL-Runtime die FC aus und meldet, wie bei der zeitgetriggerten Variante, die lokale Zeit des DUT nach der Ausführung zurück. Ebenso wird auch der späteste Zeitpunkt übergeben, zu dem die FC erneut ausgeführt werden muss, sofern nicht vorher ein anderes Ereignis auftritt.

Wie der zeitgetriggerte HiL-Stub stellt auch der ereignisgetriggerte HiL-Stub sicher, dass nur Nach-

richten der HiL-Runtime an die OutputPorts weitergeleitet werden, deren Erzeugungszeitpunkt nicht in der Zukunft liegt. Ansonsten werden die Nachrichten entsprechend verzögert, um eine Verletzung der Kausalität innerhalb der Simulation zu verhindern.

12.2.3 HiL-Runtime

Die HiL-Runtime ist Bestandteil des DUT und dient als Laufzeitumgebung für die FC, welche mittels HiL-Simulation evaluiert werden soll (vgl. Abbildung 12.2). Um eine nahtlose Integration in FERAL und die dort definierten Simulationsbeschreibungen und -szenarien zu ermöglichen, basiert das Kommunikationsmodell der HiL-Runtime für die Interaktion zwischen Simulationskomponenten auf den von FERAL definierten Prinzipien: Die HiL-Runtime stellt eine Reihe von Input- und OutputPorts mit eindeutigen Bezeichnern zur Verfügung und ermöglicht die Interaktion mit anderen Simulationskomponenten über den Austausch von Nachrichten. Es ist Aufgabe der HiL-Runtime, zwischen dem Kommunikationsmodell der ausgeführten FC und dem von FERAL zu vermitteln. Zusätzlich muss die HiL-Runtime die Ausführung der FC gemäß den Vorgaben des Direktors, welcher den HiL-Stub ansteuert, kontrollieren. Hierfür erfolgt ein Austausch von Steueranweisungen zwischen HiL-Stub und HiL-Runtime.

Wir konzentrieren uns im Folgenden auf die konkrete Umsetzung der HiL-Runtime für unsere *Imote 2*-Plattform sowie die Unterstützung von SDL-Spezifikationen als FC. Um FCs auf Basis von SDL-Spezifikationen zu ermöglichen, haben wir die SVM, bestehend aus SEnF (SDL Environment Framework) und SDLRe, dahingehend erweitert, dass diese durch die HiL-Runtime ausgeführt und deren Abarbeitung gesteuert werden kann. Die hierfür notwendigen Anpassung der SVM erfolgten durch Dennis Christmann [Chr15]. Hilfreich hierbei war die bereits bestehende Portierung von SEnF und SDLRe auf die *Imote 2*-Plattform ebenso wie die bereits vorhandene Integration als Simulator in FERAL. Aufgrund der Integration in FERAL existierten bereits spezielle SEnF-Treiber, welche eine Interaktion mit CSCs sowie anderen Simulationskomponenten innerhalb von FERAL aus einem SDL-System heraus ermöglichten (vgl. Kapitel 8.4). Diese Treiber konnten aufgrund der hohen konzeptionellen Analogien weitestgehend übernommen werden. Der Aufwand, um eine auf einer SDL-Spezifikation basierenden FSC in eine HSC zu überführen, ist sehr gering. Es genügt, mit ConTraST den Code für die SDL-Spezifikation neu zu generieren, hierbei *HiL-Runtime* als Zielumgebung auszuwählen und den generierten Code zusammen mit der HiL-Runtime für die *Imote 2*-Plattform zu übersetzen.

Die HiL-Runtime ist als Schichtenarchitektur realisiert; die unterste Schicht bildet die HiL-Base Schicht. Diese besteht im Wesentlichen aus einem Zustandsautomaten, welcher die Steueranweisungen des HiL-Stubs über UART empfängt, auswertet und über Callbacks an die übergeordnete Schicht HiL-Glue weiterreicht. Ebenso stellt HiL-Base eigene Callback-Funktionen zur Verfügung, über die die HiL-Glue-Schicht z. B. Log-Meldungen verschickt oder beim Start die eigenen Callback-Funktionen registriert, welche dann von HiL-Base aufgerufen werden. Der `FrameReceiver` operiert direkt auf dem HiL-Rahmenformat (vgl. Anhang E) und ist für die Extraktion der Anweisungen aus dem UART Datenstrom, deren Dekodierung, Vollständigkeit sowie Korrektheit verantwortlich. Die Funktionalitäten der HiL-Base Schicht beschränken sich auf die Realisierung eines Protokolls für den sicheren und verlustfreien Austausch von Steueranweisungen mit dem HiL-Stub und sind daher nicht spezifisch auf die Ausführung von FCs auf Basis von SDL zugeschnitten. Stattdessen können diese auch als Basisfunktionalität für andere Arten von FCs, z. B. auf Basis von Matlab Simulink, dienen. Gekapselt ist die Schicht HiL-Base in einer separaten Bibliothek `libHil`.

HiL-Glue baut auf der HiL-Base Schicht auf und beinhaltet den erforderlichen Glue-Code, um die durch den HiL-Stub empfangenen und durch die HiL-Base vorverarbeiteten Anweisungen auf die FC anzuwenden. Hierzu muss die HiL-Glue Schicht speziell auf die FC und deren Ausführungssemantik/-abläufe zugeschnitten werden. So stellt die Bibliothek `libSDLITL` (SDL-in-the-Loop) eine Implementierung der HiL-Glue Schicht speziell für die Anbindung von FCs auf Basis von SDL bereit. Konkret besteht die Schicht in Bezug auf SDL aus Glue-Code, welcher die Callbacks, die von der HiL-Base Schicht ausgeführt

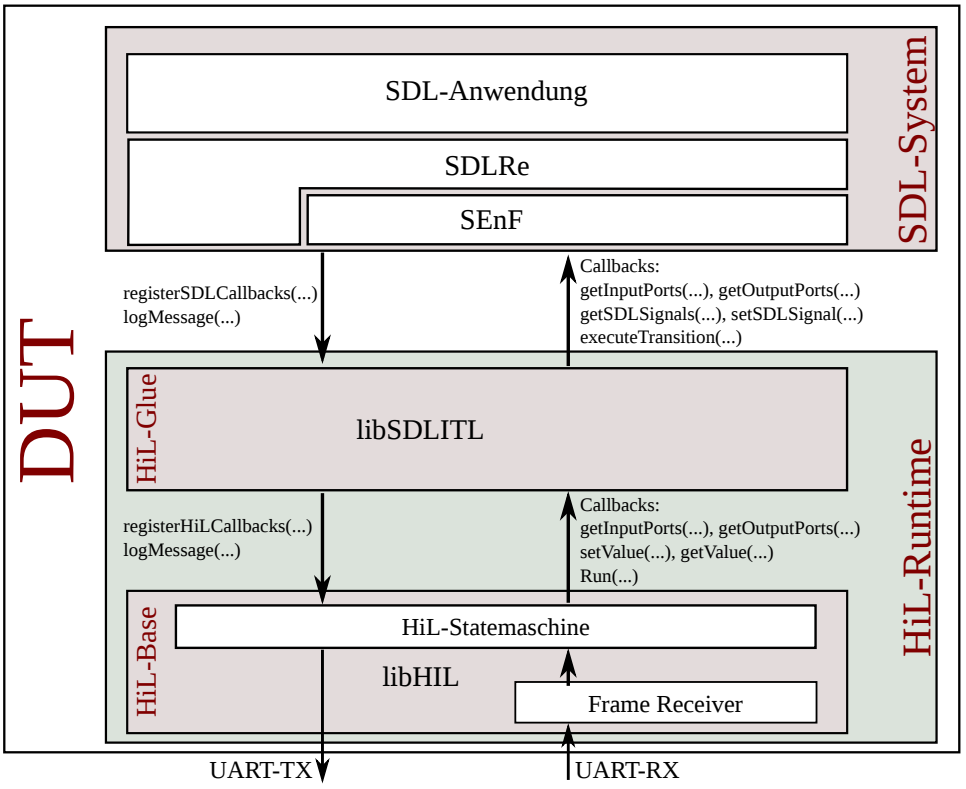


Abbildung 12.4: Diagramm des Aufbaus der HiL-Runtime.

werden, auf geeignete Funktionen innerhalb von SEnF abbildet und z. B. in Bezug auf die Interaktion mit anderen Simulationskomponenten die notwendige Konvertierung in bzw. von SDL-Signale/n über die entsprechenden SEnF-Treiber der FERAL-Integration vornimmt. Die Abbildung 12.4 verdeutlicht die Zusammenhänge noch einmal.

Soll die Unterstützung auf FCs basierend auf Matlab Simulink ausgebaut werden, genügt es, anstelle der `libSDLITL` eine andere Bibliothek zu verwenden, welche den für die Matlab-Runtime angepassten Glue-Code bereitstellt. Die HiL-Base Schicht ebenso wie der HiL-Stub können unverändert weiterverwendet werden.

Die Entkopplung zwischen den einzelnen Schichten der HiL-Runtime wird über Callbacks realisiert, deren Funktionsweise wir kurz anhand der Integration von SDL skizzieren wollen. Zunächst werden die für das SDL-System spezifischen Callbacks mit der `registerSDLCallbacks`-Methode der HiL-Glue Schicht registriert. Dies stößt wiederum die Registrierung der generischen Callbacks der HiL-Glue Schicht bei der HiL-Base Schicht (Methode `registerHiLCallbacks`) an (linke Seite der Grafik). Die hierbei registrierten Callbacks sind rechts an den nach oben zeigenden Pfeilen aufgeführt (die Pfeile zeigen die Kommunikationsrichtung an).

Die HiL-Base Schicht kontrolliert die Ausführung der FC mit Hilfe dieser Callbacks und ruft beim Empfang von Steueranweisungen die entsprechenden Callbacks der HiL-Glue Schicht (hier `libSDLITL`) auf. Diese passt, soweit notwendig, die übergebenen Parameter an (z. B. Umwandlung einer Nachricht aus FERAL in ein SDL-Signal) und ruft die spezifischen Callbacks des SDL-Systems, oder genauer die von SEnF bereitgestellten Callbacks auf. Diese leiten das eingegangene SDL-Signal an die zuständigen SEnF-Treiber weiter, welche es verarbeiten oder in aufbereiteter Form an das SDL-System weiterreichen.

Ein wichtiger Aspekt bei der HiL-Simulation ist die Berücksichtigung der real verfügbaren Ressourcen in Bezug auf die Zielplattform. In der Praxis besonders interessant sind die hieraus resultierenden

Laufzeiten für das funktionale Verhalten und inwiefern sich dieses auf das Gesamtverhalten des Systems auswirkt. Um ein realistisches Ergebnis zu erhalten, darf hierbei nur die Ausführungszeit des FC sowie dessen Runtime, nicht aber die Ausführungszeiten der HiL-Runtime quantifiziert werden. Dies wird erreicht, indem die HiL-Runtime die lokale Uhr des *Imote 2*-Knotens kontrolliert und diese so manipuliert, dass die (basierend auf dieser Uhr) gemessene Simulationszeit nur voranschreitet, sofern das FC Aktionen – im Fall von SDL Transitionen oder Funktionen von SEnF – ausführt. Während die HiL-Runtime Verarbeitungen durchführt wie z. B. die Abwicklung der Kommunikationen mit dem HiL-Stub, wird die lokale Uhr angehalten, sodass die Simulationszeit auf dem DUT nicht weiter voranschreitet.

Da die SVM nicht vollständig präamptiv arbeitet, sondern nur eine Unterbrechung nach einer vollständigen Transition erlaubt, kann (bei einer zeitgetriggerten Ausführung) das Ende einer Transitionsausführung die Dauer des vom Direktor vorgegebenen Zeitschritts überschreiten. Deshalb wird am Ende jedes Ausführungsschrittes die aktuelle Simulationszeit des DUT an den HiL-Stub übermittelt, sodass dieser eine Resynchronisation vornehmen kann (vgl. Kapitel 12.2.2). Gleichzeitig verzögert der HiL-Stub auch Nachrichten an andere Simulationskomponenten, deren Erzeugungszeitpunkt aus Sicht von FERAL in der Zukunft liegt. Auf diese Weise bleibt die Kausalität und Korrektheit gewahrt.

12.3 HiL-Simulation des linearen inversen Pendels in FERAL

Um sowohl unsere HiL-Lösung für FERAL als auch die iterative Entwicklung mittels Virtual Prototyping mit FERAL zu demonstrieren, zeigen wir dessen Anwendbarkeit anhand der Entwicklung eines verteilten, vernetzten Regelungssystems. Als Regelstrecke kommt ein inverses Pendel zum Einsatz. Hierbei handelt es sich um ein im Bereich der Regelungstechnik sehr häufig untersuchtes Problem, da das resultierende System instabil ist sowie eine nichtlineare Dynamik ausweist [AG04, Mes], woraus hohe regelungstechnische Anforderungen (Abtastrate, Reaktionszeiten, ...) resultieren.

Der schematische Aufbau eines linearen inversen Pendels ist in Abbildung 12.5 dargestellt. Es gibt verschiedene Varianten mit unterschiedlichen Freiheits- und Schweregraden. Die von uns betrachtete Variante besteht aus einem Wagen, der sich entlang einer Schiene bewegen kann. An dem Wagen ist eine starre Achse mit einem Gewicht am Ende angebracht, welche über ein Lager drehbar mit dem Wagen verbunden ist (Basispunkt). Bei dem betrachteten linearen inversen Pendel erlaubt das Lager nur Kippbewegungen in der Bewegungsebene des Wagens.

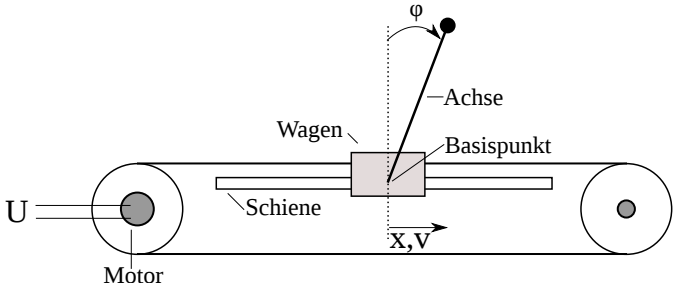


Abbildung 12.5: Aufbau eines linearen inversen Pendels.

Der Winkel ϕ beschreibt die Abweichung der Achse von der Vertikalen (in der Abbildung gestrichelt dargestellt). Die Aufgabe der Regelung besteht darin, durch geeignete Ansteuerung des Motors (welcher den Wagen bewegt) die Abweichungen des Pendels aus der Vertikalen zu minimieren. Die Regelung basiert darauf, den Basispunkt, an dem die Achse mit dem Wagen verbunden ist, unter das Gewicht zu *bewegen*. Bei der Mittelposition $\phi = 0°$ handelt es sich um eine sogenannte instabile Ruhelage, da bereits geringfügige Störungen dazu führen, dass das Gewicht diese verlässt und in eine Fallbewegung übergeht.

In diesem Fall strebt das gesamte System einen möglichst energiearmen Zustand an, welcher erreicht wird, sobald sich das Gewicht in der stabilen Ruhelage $\phi = 180°$ befindet (und die aus dem Fall resultierende kinetische Energie ebenfalls Null beträgt).

In dem hier beschriebenen Aufbau des linearen inversen Pendels wird der Wagen über einen Elektromotor (Aktuator) bewegt, welcher über einen Riemen mit dem Wagen verbunden ist. Die Ansteuerung erfolgt über das Anlegen einer Spannung U an den Motor. Das Vorzeichen der Spannung gibt die Drehrichtung des Motors und somit die Bewegungsrichtung des Wagens an. Über die Amplitude der angelegten Spannung lässt sich die Geschwindigkeit des Wagens beeinflussen. Neben dem Aktuator verfügt die Regelstrecke über drei Sensoren, zur Erfassung des Winkels ϕ und der Position x sowie der Geschwindigkeit v des Wagens.

Aus Sicht der Regelungstechnik gibt es drei Aufgabenstellungen: Die erste besteht darin, die instabile Ruhelage aus der stabilen Ruhelage, durch Aufschwingen des Pendels zu erreichen. Die zweite Aufgabenstellung zielt darauf ab, das Pendel – gegen den Einfluss von Störgrößen – in der instabilen Ruhelage, durch geeignete Ansteuerung des Aktuators zu halten. Bei der dritten muss, zusätzlich zur Stabilisierung, der Wagen in eine bestimmte Position fahren.

Wir beschränken uns auf die Stabilisierung des Systems und gehen davon aus, dass sich zum Startzeitpunkt das Pendel in der instabilen Ruhelage befindet. Die Stabilisierung kann unter Verwendung unterschiedlicher Arten von Reglern erfolgen, z. B. PID-, Kaskaden-, Zustands- oder IMC-Regler. Hierbei hat neben der Wahl und Parametrisierung des Reglers auch die Abtastrate sowie die Regelfrequenz einen Einfluss auf die Regelgüte (und somit die Stabilität) [Krä13]. Eine vollständige Beschreibung des Regelungsproblems des linearen inversen Pendels als solches sowie die Herleitung eines mathematischen Modells der Regelstrecke und der Bewegungsgleichungen finden sich, unter anderen, in [Lun10, Ala13, Mes].

12.3.1 Ausgangssituation

Da in dieser Arbeit nicht der Fokus auf der Entwicklung des Reglers für das inverse Pendel liegt, sondern hier die Anwendbarkeit von Virtual Prototyping und HiL-Simulationen im Kontext verteilter eingebetteter Echtzeitsysteme evaluiert werden soll, nutzen wir die im Rahmen des DFG-Schwerpunktprogrammes 1305[7] „Regelungstheorie digital vernetzter dynamischer Systeme“ geleisteten Vorarbeiten. Dessen Gegenstand war die Entwicklung von Konzepten für die Realisierung drahtloser, verteilter, vernetzter Regelungssysteme. Hierzu gehörte auch die Umsetzung eines verteilten, vernetzten Regelungssystems für ein lineares inverses Pendel unter Anwendung modellgetriebener Entwicklungsansätze. Ein entsprechendes Regelungssystem konnte, basierend auf den Forschungsergebnissen, entworfen, aufgebaut und erfolgreich evaluiert werden [Krä13, Ala13, CKLG09, CLKG09, CL10].

Hierzu wurde zunächst ein Matlab Modell der Regelstrecke des inversen Pendels (welches in Form eines physikalischen Aufbaus am Lehrstuhl für Automatisierungstechnik der TU-Kaiserslautern existiert) entwickelt. Auf Basis dieses mathematischen Modells wurden unterschiedliche Reglertypen entworfen und evaluiert. Anschließend erfolgten Tests an der realen Regelstrecke, wobei das Regelungssystem selbst als verteiltes Echtzeitsystem realisiert wurde. In dem konkreten Aufbau kommen fünf Sensorknoten auf Basis des *Imote 2* zum Einsatz. Hiervon übernehmen drei Knoten die Aufgabe der Sensordatenerfassung (jeweils ein Knoten für die Erfassung der Position des Wagens, dessen Geschwindigkeit sowie der Abweichung des Pendels von der Ruhelage). Die erfassten Daten werden über den integrierten CC2420-Transceiver-Transceiver des *Imote 2* an den Regler übermittelt, welcher auf dem vierten Sensorknoten läuft und anhand der Sensorwerte neue Steuergrößen für den Aktuator berechnet. Der Aktuator (in diesem Fall der Elektromotor) wird durch den fünften *Imote 2*-Knoten angesteuert. Die Steuergröße wird ebenso wie die Sensorwerte per Funk zum Aktuator übertragen.

Um ein deterministisches Kommunikationsverhalten sowie deterministische Garantien in Bezug auf auftretende Verzögerungen garantieren zu können, kommt ein TDMA-basiertes Medienzugriffsverfah-

[7] Fördernummer LI 724/15-1 und GO 503-8-1

ren zur Anwendung. Die Slotreservierungen erlauben durch ein geeignetes Duty-Cycling zudem eine Reduktion des Energiebedarfs der Knoten und somit eine längere Laufzeit bei beschränkten Energieressourcen [Krä13].

Die Bereitstellung der Sensorwerte sowie der Steuergröße wurde unter Verwendung einer dienstbasierten Middleware realisiert. Anstatt konkrete Sensorwerte von einem Knoten anzufordern, abonniert der Regler z. B. den Dienst *Winkelposition*. Welche Funktionalität letztendlich diesen Dienst erbringt und auf welchem Knoten diese ausgeführt wird, spielt für den Regler keine Rolle, die Vermittlung liegt in der Verantwortlichkeit der Middleware. Aus Anwendungssicht verbessert der Einsatz der Middleware die Abstrahierbarkeit. So ist es für die Realisierung der Anwendung unerheblich, wo die Funktionalität, die den Dienst anbietet, ausgeführt wird – lokal oder auf einem anderen Knoten. Die Middleware sorgt für eine transparente Nutzbarkeit und Bereitstellung der Daten der angebotenen Dienste.

Bei dem Regelungssystem des inversen Pendels wurde sowohl die verwendete MAC-Schicht *MacZ light* als auch die dienstbasierte Middleware NCS-CoM [Fli09, Krä13] mit Hilfe des modellgetriebenen Entwicklungsansatzes SDL-MDD erstellt (SDL Model-driven Development [Got07, KGW06]). Der Code für die verschiedenen Knoten wurde mit ConTRaST automatisch aus der SDL-Spezifikation generiert und verwendet SDLRe und SEnF. Die eigentlichen Funktionalitäten (Auslesen von Sensordaten, Regelungsalgorithmus, Anlegen neuer Steuergröße an den Aktuator) wurden von Hand implementiert und in die SDL-Systeme über eine Anwendungsschnittstelle [Krä13] integriert. Für unsere Evaluation beschränken wir uns auf die Betrachtung eines einfachen PID-Regler für die Stabilisierung des Pendels.

12.3.2 Verwendung von Virtual Prototyping bei der Entwicklung des verteilten Regelungssystems

Als das verteilte Regelungssystem für das lineare inverse Pendel im Rahmen des DFG-Projekts entwickelt wurde, standen weder FERAL noch eine vergleichbare Lösung für Virtual Prototyping zur Verfügung. Zwar kam bereits Matlab für die Entwicklung der einzelnen Reglertypen und der Regelstrecke zum Einsatz, das Zusammenspiel zwischen dem Regler und der dienstbasierten Middleware sowie die Auswirkungen von Verzögerungen (Kommunikations- und Laufzeitverzögerungen) ließen sich damit aber nicht simulieren. Aus diesen Gründen wurden Verzögerungsglieder in das Modell des Regelkreises zwischen Sensor und Regler bzw. Regler und Aktuator integriert. Doch dies liefert nur einen ersten Eindruck von dem Verhalten. Das spezifische Verzögerungsverhalten, welches durch die Kombination von MacZ light (mit TDMA) sowie die NCS-CoM entsteht, sowie die hieraus resultierenden Auswirkungen konnten so nicht evaluiert werden. Dementsprechend waren reale Integrationstests der Anwendungskomponenten (Regler, Sensor, Aktuator) zusammen mit dem Protokollstack (NCS-CoM, MacZ light) erst auf der *Imote 2*-Plattform möglich.

Der Test auf der Zielplattform gestaltete sich aufwändig und schwierig, unter anderem aufgrund von fehlender Hardware sowie beschränkten Zugangsmöglichkeiten zur physikalischen Regelstrecke. Auftretende Probleme konnten oft nicht oder nur unter Mühen reproduziert werden, sodass durchgeführte Korrekturen nur schwer überprüfbar waren. Ursachen hierfür sind sowohl Störgrößen, welche die Regelstrecke und damit den Ablauf direkt beeinflussen, aber auch sporadische Fehler bei der Kommunikation durch Einstrahlungen oder Interferenzen. Insgesamt erschwerten diese Umstände die Tests sowie die daran anschließende Auswertung, Diagnose und Fehlerbehebung. Diesbezüglich wäre eine Simulationsumgebung, welche das Gesamtsystem inkl. Regelstrecke abdeckt, eine enorme Erleichterung für die funktionale Evaluation und Fehlerbehebung gewesen, da hierbei sowohl die Testergebnisse reproduzierbar als auch Umwelteinflüsse kontrollierbar sind. Zudem bestehen bessere Möglichkeiten der Protokollierung, ohne dabei das funktionale Verhalten (z. B. durch zusätzliche Verzögerungen für die Ausgabe von Debug-Informationen) zu beeinflussen. Des Weiteren kann eine Simulation zu jedem Zeitpunkt angehalten, um den internen Zustand der verschiedenen Knoten sowie deren Anwendungen mittels eines Debuggers zu inspizieren, und danach wieder fortgesetzt werden, ein Vorgehen, welches mit dem realen Pendel nicht möglich ist.

Auch gestaltet es sich in der Praxis schwierig, den Zustand der Systeme auf verschiedenen Knoten zum gleichen Zeitpunkt zu ermitteln, um ein konsistentes Bild des Zustands des Gesamtsystems zu erhalten, wenn es um die Suche von Fehlern bei der Interaktion geht.

Diese realen Probleme, die seinerzeit bei der Entwicklung aufgetreten sind, dienen uns nun als Motivation, um zu untersuchen, inwiefern FERAL bei der Entwicklung dieses konkreten vernetzten Regelungssystems hätte verwendet werden können.

12.3.2.1 Entwurf eines Simulationssystems für das vernetzte Regelungssystem des inversen Pendels

Um FERAL verwenden zu können, haben wir in einem ersten Schritt das in Kapitel 12.3.1 beschriebene Szenario in einem Simulationssystem für FERAL nachgebaut, wobei wir einige Vereinfachungen vornahmen, um das resultierende System kompakt zu halten. Als Grundlage für die Simulation der Regelstrecke (inverses Pendel) diente das von Andreas Haupt[8] freundlicherweise bereitgestellte Simulink Modell des realen inversen Pendels des Lehrstuhls für Automatisierungstechnik der TU Kaiserslautern (vgl. [Ala13, CL10]). Dieses beinhaltet neben der Regelstrecke auch ein Modell des Aktuators (Stellglied) sowie des Sensors. Das Simulink Modell wurde mit Hilfe von FERALs Matlab Simulink Integration in eine eigenständige FSC umgewandelt, welche als Eingabe die aktuelle Steuergröße entgegennimmt (InputPort) und die drei Sensorwerte als Ausgänge bereitstellt (OutputPorts).

In unserem Simulationssystem beschränken wir uns auf drei kommunizierende Knoten. Hierbei fungiert ein Knoten als Sensor und stellt alle drei Sensorwerte zur Verfügung, welche von der Regelstrecke bereitgestellt werden. Ein weiterer Konten übernimmt die Rolle des Reglers und führt den Regelungsalgorithmus aus. Der dritte Knoten empfängt die ermittelte Steuergröße des Reglers und reicht diese wiederum an die Regelstrecke weiter. Eine zusätzliche FSC hat die Aufgabe, sämtliche ausgetauschten Rahmen zwischen den Knoten ebenso wie die Ein- und Ausgaben der FSC, die das Verhalten der Regelstrecke nachbildet, aufzuzeichnen.

Um möglichst nahe an dem real umgesetzten Regelungssystem für das invertierte Pendel zu bleiben, haben wir die im Rahmen des DFG-Projektes entstandenen SDL-Spezifikationen als Verhaltensspezifikation der FSCs der einzelnen Knoten – mit geringfügigen Modifikationen[9] – verwendet: In Bezug auf den Regler beschränken wir uns auf einen einfachen PID-Regler und verzichten auf komplexe zustandsbasierte Regler. Die drei Sensorknoten wurden durch ein einziges FSC ersetzt, welches die drei Sensorwerte zu einem einzigen Dienst zusammenfasst. Hierfür sind, dank der dienstbasierten Middleware, nur geringfüge Anpassungen an den SDL-Spezifikationen der einzelnen Knoten erforderlich. Im realen System greifen Sensor und Aktuator direkt auf die Hardware der Sensoren per ADC[10] zu oder stellen eine Steuerspannung mittels eines DAC[10] ein. Für die Simulation wurde der direkte Zugriff auf die Hardware durch eine Interaktion mit der FSC der Regelstrecke durch die Verwendung des Forward-Signals-SEnF-Treiber (Kapitel 8.4) ersetzt. Bei der Übermittlung der Sensorwerte werden bei der Simulation direkt die physikalisch gemessenen Größen genutzt, d.h., anstelle eines gemessenen Spannungswertes wird z. B. direkt der korrespondierende Winkel übertragen.

Abstraktes Simulationssystem des inversen Pendels

Die Abbildung 12.6 zeigt die Struktur des abstrakten Simulationssystems für das inverse Pendel. Die FSC *Controller* übernimmt die Aufgaben des Reglers. Die beiden FSCs *Actuator* und *Sensors* stellen Dienste zum Abonnieren von Sensorwerten bzw. der Steuergröße bereit und nutzen hierfür die NCS-CoM Middleware. Sämtliche Nachrichten (Verwaltungsnachrichten der Middleware, Sensorwerte und Steuergröße) werden zwischen den FSCs über die CSC mit der Bezeichnung `WirelessCOM` ausgetauscht. Diese

[8] Mitarbeiter des Lehrstuhls für Automatisierungstechnik der TU Kaiserslautern

[9] Hierzu gehörten, neben den beschriebenen strukturellen Anpassungen, auch die Behebung einiger in der Praxis auftretender sporadischer Fehler, die sich aufgrund der Möglichkeiten der Simulation nun reproduzieren und nachvollziehen ließen.

[10] ADC – Analog Digital Converter; DAC – Digital Analog Converter

CSC simuliert ein gemeinsames drahtloses Kommunikationsmedium der Knoten. Die FSC `Pendulum` bildet das physikalische Verhalten der realen Regelstrecke mit Hilfe des bereitgestellten Simulink Modells nach und stellt OutputPorts für die Sensorwerte sowie einen InputPort für die Steuergröße des Motors bereit. Aus diesem Grund sind die beiden FSCs `Actuator` und `Sensors` auch direkt über Links mit dem `Pendulum` verbunden, um neue Steuergrößen setzen bzw. die Sensorwerte auslesen zu können. Der `Logger` hat die Aufgabe, sowohl die Steuergröße des `Actuator` als auch die systemrelevanten Größen des `Pendulum` aufzuzeichnen.

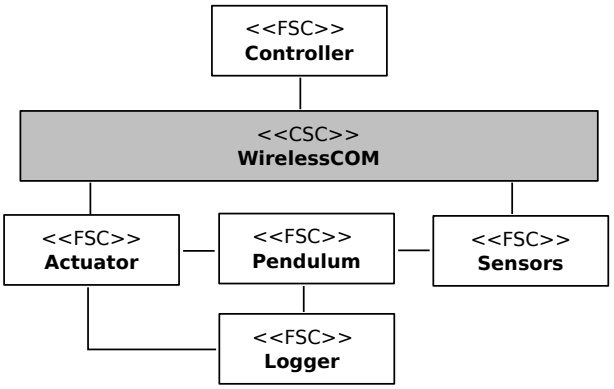


Abbildung 12.6: Abstraktes Simulationssystem für das inverse Pendel.

Konkrete Simulationssysteme für das inverse Pendel

Wir definieren für unsere Evaluation vier konkrete Simulationssysteme, welche vier unterschiedliche Entwicklungsphasen resp. Abstraktionsstufen repräsentieren. Mit jeder neuen Abstraktionsstufe erweitern wir den Detailgrad sowohl in Bezug auf den funktionalen Umfang der Verhaltensmodelle als auch in Bezug auf den Detailgrad der simulierten Umgebung der FSCs. Hierbei wollen wir einerseits zeigen, wie ein von FERAL unterstützter Entwicklungsprozess anhand dieses Beispiels aussehen kann, und andererseits zeigen, dass die häufig ignorierten nicht-funktionalen Aspekte (Laufzeit-, Kommunikationsverzögerungen) durchaus Einfluss auf das funktionale Verhalten haben können. In diesem Beispiel verwenden wir hierfür die Regelgüte als Indikator, um die Ergebnisse der Simulationen der verschiedenen konkreten Simulationssysteme zu vergleichen.

Das erste Simulationssystem dient der rein funktionalen Evaluation der einzelnen Komponenten und wird von uns als Basis verwendet, um die Parameter des PID-Reglers zu bestimmen[11]. In Bezug auf unseren Regelkreis streben wir ein Intervall von 60 ms an, d.h., alle 60 ms liefert der Sensor neue Messwerte, und der Controller bestimmt auf dieser Basis eine neue Steuergröße für den Aktuator. Bereits hier verwenden wir SDL-Spezifikationen zur Beschreibung des Verhaltens der einzelnen FSCs, allerdings bleiben auf dieser Abstraktionsstufe Verarbeitungszeiten sowie Übertragungsverzögerungen unberücksichtigt. Hierzu simulieren wir die Kommunikation mittels der PTP-CSC[12] und die SDL-Systeme mit unserem SDL-Simulator für FERAL (vgl. Kapitel 8.4). Zusätzlich wurde der Protokollstack in Bezug auf die Kommunikation reduziert. So kommt zwar noch die dienstbasierte Middleware NCS-CoM zum Einsatz, doch anstelle des zeitgetriggerten MAC-Layer *MacZ light* erfolgt die Kommunikation ereignisbasiert, ohne TDMA und reservierte Slots. Somit werden in Bezug auf die Kommunikation nur noch funktionale Aspekte berücksichtigt (Registrieren und Abonnieren von Diensten), aber keine Übertragungsverzögerungen (Propagation-Delay, Totzeiten, in denen auf den reservierten Slot gewartet wird). Dieses konkrete

[11] Dies entspricht der klassischen Vorgehensweise, die z. B. auch beim DFG-Projekt mit Simulink angewendet wurde. Der nächste Schritt bestand in einer langwierigen Integrationsphase mit Fehlerbehebungen und Optimierungen.

[12] Bei der PTP-CSC handelt es sich um einen einfacheren Vorgänger des ACOM-CSC, welcher Rahmen ohne Verzögerung weiterleitet und hierbei auch gleichzeitige Übertragungen gestattet, ohne dass es zu Interferenzen kommt.

Simulationssystem bezeichnen wir *PTP-Pendulum-Reduced*. Es gestattet die funktionale Evaluation der NCS-CoM mit realistischen Kommunikationsmustern und -abläufen und kann verwendet werden, um die Regelparameter festzulegen.

Das zweite konkrete Simulationssystem *PTP-Pendulum* integriert den vollständigen Protokollstack, bestehend aus NCS-CoM und dem zeitgetriggertem *MacZ light*. *MacZ light* implementiert ein zeitgetriggertes Kommunikationsprotokoll (TDMA) mit exklusiven Slotreservierungen, d.h., die Knoten dürfen nur innerhalb der ihnen zugewiesenen Slots kommunizieren. Dies gilt sowohl in Bezug auf die Erbringung von Diensten (z. B. Übermittlung von Sensorwerten) wie auch den Austausch von Verwaltungsnachrichten der NCS-CoM. *MacZ light* verwendet in diesem Szenario ein Resynchronisationsintervall von einer Sekunde (Makroslot). Ein Makroslot setzt sich aus 100 Mikroslots der Länge 10 ms zusammen. Der erste Slot pro Makroslot ist für die Synchronisation reserviert. Anschließend ist jeweils in direkter Abfolge ein Slot für den Sensor, einer für den Controller sowie ein Slot für den Aktuator reserviert. Diese Abfolge der Slotreservierungen wiederholt sich ununterbrochen, bis zum Ende des Makroslots. Die CSC WirelessCOM wird auch in diesem Simulationssystem weiterhin durch die PTP-CSC simuliert. In diesem Simulationssystem kann sowohl *MacZ light* als auch das Zusammenwirken zwischen NCS-CoM und einer zeitgetriggerten Kommunikation getestet werden. Gleichzeitig erlaubt die Simulation die Beurteilung der Auswirkungen einer zeitgetriggerten Kommunikation auf die Regelgüte.

Aufbauend auf diesem konkreten Simulationssystem (sowie dessen Slotkonfiguration) definieren wir zwei weitere konkrete Simulationssysteme: *CC2420-Pendulum* und *HiL-Pendulum*. *CC2420-Pendulum* unterscheidet sich von dem konkreten Simulationssystem *PTP-Pendulum* durch eine realistische Simulation der Kommunikation. Hierzu wird die *WirelessCOM*-CSC durch die ns-3-CSC simuliert, wobei die von Anuschka Igel [IG13] entwickelte ns-3-Erweiterung verwendet wird, welche die Eigenschaften des CC2420-Transceivers nachbildet. Dieser Transceiver kommt auch auf der Zielplattform zum Einsatz und ist somit für eine realistische Simulation optimal geeignet. In dieser Abstraktionsstufe werden sowohl hardwarespezifische Eigenschaften des Transceivers (Verzögerungen, Umschaltzeiten, Totzeiten) als auch das Propagation-Delay und ggf. Kollisionen berücksichtigt. Ein enormer Vorteil besteht darin, dass das simulierte Medium frei von Störungen und Interferenzen ist und sich Fehler bei der Kommunikation daher reproduzieren lassen und nicht auf sporadischen Einstrahlungen beruhen. Auch die Reichweite der Übertragungen in Relation zur Sendestärke sowie Multihop-Netzwerke können simuliert werden. Dieser Aspekt spielte für die untersuchte Variante des Regelsystems jedoch keine Rolle.

Auch das konkrete Simulationssystem *HiL-Pendulum* nutzt ns-3-CSC für die Simulation des CC2420-Transceivers. Im Gegensatz zum *CC2420-Pendulum* verwendet *HiL-Pendulum* jedoch unsere HiL-Lösungen, um das SDL-System des *Controller* direkt auf dem *Imote 2*-Knoten auszuführen. So lassen sich Einflüsse der Ausführungszeiten auf der *Imote 2*-Plattform auf das Verhalten des Gesamtsystems sowie die Regelgüte untersuchen.

Die Tabelle 12.1 gibt eine Übersicht über die konkreten Simulationssysteme und fasst deren Gemeinsamkeiten und Unterschiede zusammen. Angegeben ist jeweils der für die Komponente verwendete Simulator sowie (in Klammern) das ausgeführte Verhaltensmodell. Bei den SDL-Systemen ist jeweils der Aufbau des Protokollstacks sowie der verwendete SEnF-Treiber angegeben, welcher für die Interaktion mit FERAL (in dem Verhaltensmodell), verwendet wird. RS steht für die Verwendung der Middleware NCS-CoM mit einer ereignisgetriggerten MAC-Schicht, während die Abkürzung FS für die Kombination aus NCS-CoM und *MacZ light* mit der vorgestellten Slotkonfiguration verwendet wird. Alle drei Systeme nutzen den gleichen CC2420-ns-3-SEnF-Treiber und unterscheiden sich nicht hinsichtlich ihrer Verhaltensspezifikation. Dieser bildet die Schnittstelle des regulären CC2420-Hardwaretreibers nach, erzeugt jedoch spezielle Nachrichtentypen, die sowohl von der PTP-CSC als auch der ns-3-CSC unterstützt werden. Hierdurch unterscheiden sich die Verhaltensmodelle der FSCs nur in den oben genannten Details von den tatsächlich eingesetzten SDL-Systemen des vernetzten Regelungssystems aus dem DFG-Projekt. Zu besseren Übersicht sind die Änderungen zwischen den einzelnen Simulationssystemen rot hervorgehoben.

Simulations-komponente	Type	Simulator (Verhaltensmodell)			
Pendulum	FSC	Matlab Simulink (Simulink Modell)			
Logger	FSC	Native Simulationskomponente (Java)			
WirelessCOM	CSC	PTP-CSC (CC2420)	PTP-CSC (CC2420)	ns-3 (CC2420)	ns-3 (CC2420)
Actuator	FSC	SDL (RS, CC2420)	SDL (FS, CC2420)	SDL (FS, CC2420)	SDL (FS, CC2420)
Sensors	FSC	SDL (RS, CC2420)	SDL (FS, CC2420)	SDL (FS, CC2420)	SDL (FS, CC2420)
Controller	FSC	SDL (RS, CC2420)	SDL (FS, CC2420)	SDL (FS, CC2420)	HiL (FS, CC2420)
Bezeichner des konkr. Simulationssystems:		PTP-Pendulum-Reduced	PTP-Pendulum	CC2420-Pendulum	HiL-Pendulum

Tabelle 12.1: Übersicht der konkreten Simulationssysteme des inversen Pendels.

12.3.2.2 Ergebnisse der Simulation

Die vier vorgestellten konkreten Simulationssysteme wurden jeweils mit zwei unterschiedlichen Sätzen an Parametern für den PID-Regler simuliert, um die Auswirkungen des höheren Realismus der verschiedenen Abstraktionsstufen auf die Regelgüte in Verbindung mit dem verwendeten Regler zu demonstrieren. Die Ausgangssituation für unsere Simulationen ist ein Pendel, welches sich 4° außerhalb der instabilen Ruheposition befindet.

Ergebnisse für den PD-Regler

Der erste Parametersatz reduziert den PID-Regler zu einem PD-Regler. Die Ergebnisse der Simulation sind in der Abbildung 12.7 dargestellt. Die Auslenkung des Pendels ist in grün aufgetragen, die Steuergröße, welche durch den Regler ermittelt wird, in blau. In dem Simulationssystem *PTP-Pendulum-Reduced*, in dem die Kommunikation nicht verzögerungsbehaftet ist und ereignisbasiert abläuft, funktioniert der PD-Regler gut, wie die Abbildung 12.7a illustriert. Die Abweichung wird schnell ausgeregelt, auch wenn die Überschwinger auf eine noch nicht perfekte Parametrierung hindeuten. Auch die Steuergröße weist einen sehr sanften Verlauf auf und lässt sehr viel Spielraum in Bezug auf die maximal zulässigen Werte der Steuergröße (dargestellt als rote Linie), sodass ruckartige Lastwechsel vermieden werden und die Mechanik geschont wird.

Das Simulationssystem *PTP-Pendulum* berücksichtigt die Verzögerung, welche sich aus dem Einsatz des vollständigen Protokollstacks (NCS-CoM, MacZ light) ergeben. Dadurch zeigt sich (Abbildung 12.7b) eine deutliche Verschlechterung der Regelgüte: Der Regler benötigt deutlich länger, um die Auslenkung zu kompensieren, als im ersten Simulationssystem.

Wenn wir nun auch noch die charakteristischen Verzögerungen des CC2420-Transceivers simulieren (Simulationssystem *CC2420-Pendulum*), führt dies zu einer weiteren Verschlechterung der Regelgüte. Wie Abbildung 12.7c dokumentiert, kann der Regler die Auslenkung zwar kompensieren (erkennbar an der langsam abnehmenden Amplitude der Auslenkung), benötigt hierfür aber sehr lange. Das die

Steuergröße, trotz der langsamen Stabilisierung, den möglichen Wertebereich kaum nutzt, unterstreicht die nicht optimale Parametrierung des Reglers noch einmal.

Es verwundert nicht, dass die Simulation der Verarbeitungsverzögerungen des Reglers auf der *Imote 2*-Plattform durch das konkrete Simulationssystem *HiL-Pendulum* zu einer weiteren Verschlechterung der Regelgüte führt. Wie in der Abbildung 12.7d ersichtlich gelingt es dem Regler nicht, die Störung zu kompensieren, und das Pendel kippt nach kurzer Zeit um.

Somit ist der PD-Regler mit den gewählten Parametern nicht für die Regelstrecke geeignet, obwohl die ersten Simulationen zunächst vielversprechend aussahen (Abbildung 12.7a). Erst mit der Berücksichtigung der nicht-funktionalen Eigenschaften (Verzögerungen, Laufzeit), die durch die verteilte Natur des Systems auftreten, konnten diese Defizite identifiziert werden.

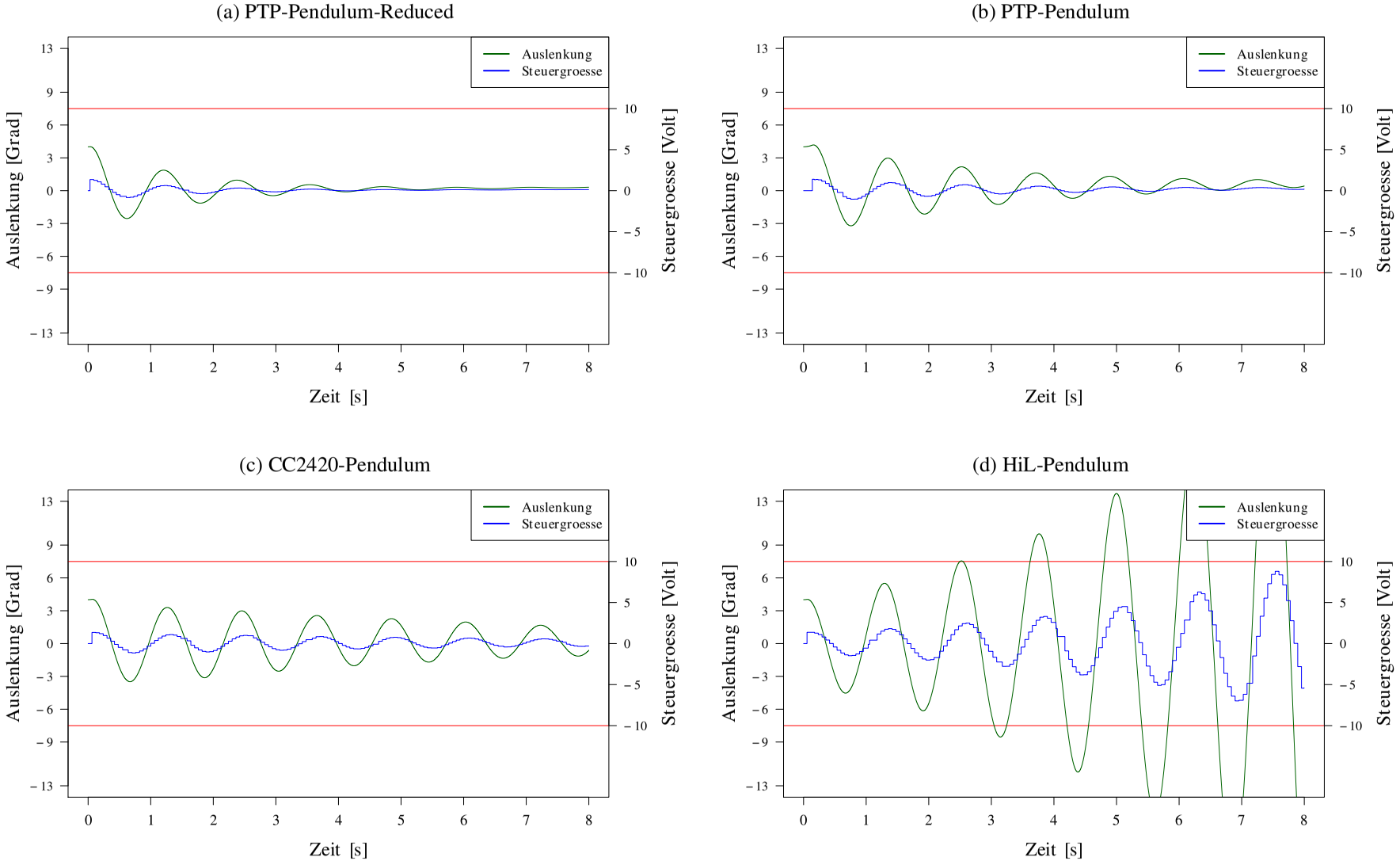


Abbildung 12.7: Darstellung der Aussteuerung des Pendels sowie der angelegten Steuergröße der verschiedenen konkreten Simulationssysteme bei Verwendung eines PD-Reglers.

Ergebnisse für den PID-Regler

Nachdem die Ergebnisse für den PD-Regler nicht optimal waren, untersuchen wir nun noch einmal einen PID-Regler. Die vorgestellten konkreten Simulationssysteme bleiben (bis auf die Parameter der Reglers) unverändert.

Wir beginnen mit der Betrachtung wieder bei dem Simulationssystem *PTP-Pendulum-Reduced* (Abbildung 12.8a). Im Vergleich zum PD-Regler fällt auf, dass der PID-Regler die Störung deutlich schneller kompensiert und dabei den Wertebereich für die Steuergröße voll ausnutzt. Die schnelle Kompensation gelingt dem PID-Regler durch sehr schnelles Umschalten der Laufrichtung des Motors. Hieraus resultieren jedoch hohe mechanische Lasten, die unter Umständen die Lebensdauer des Pendels sowie des Aktuators reduzieren.

Während beim *PTP-Pendulum-Reduced* der PID-Regler seine Aufgabe gut erfüllt, führt bei dem Simulationssystem *PTP-Pendulum* die ruckartige Modifikation der Steuergröße dazu, dass sich die Auslenkung aufschwingt und sogar größer als die ursprüngliche Störung wird (Abbildung 12.8b). Bei der Betrachtung der Steuergröße fällt auf, dass der PID-Regler (fast) ohne Zwischenstufen zwischen den Maximalwerten hin- und herspringt. Dennoch kippt das Pendel auch bei längerer Laufzeit nicht; eine Ausregelung in die instabile Ruhelage gelingt dem PID-Regler jedoch auch nicht.

Bei der Betrachtung der Simulationsergebnisse des Simulationssystems *CC2420-Pendulum* (Abbildung 12.8c) fallen die gleichen Punkte auf, die bereits für das Simulationssystem *PTP-Pendulum* gelten. Allerdings führen die zusätzlichen Kommunikationsverzögerungen (Propagation-Delay, Verzögerungen des CC2420-Transceivers) dazu, dass die Überschwinger des Pendels insgesamt kleiner sind als bei dem Simulationssystem *CC2420-Pendulum*. Dieses Ergebnis illustriert, wie schwierig es ist, die Einflüsse von Verzögerungen auf die Regelgüte oder allgemein das Gesamtverhalten von verteilten Systemen vorherzusagen.

Berücksichtigen wir nun in unserem Simulationssystem *HiL-Pendulum* zusätzlich die Verarbeitungsverzögerung auf der *Imote 2*-Plattform, so verschlechtert sich die Regelgüte noch einmal deutlich im Vergleich zu den beiden vorherigen Simulationssystemen (Abbildung 12.8d). Auch hier gelingt es dem PID-Regler nicht, das Pendel in die instabile Ruhelage auszuregeln. Aber auch hier fällt das Pendel selbst nach längerer Laufzeit, im Gegensatz zum PD-Regler, nicht um. Dennoch zeigt auch dieses Ergebnis, dass der PID-Regler für die Aufgabe in dieser Form nicht geeignet ist.

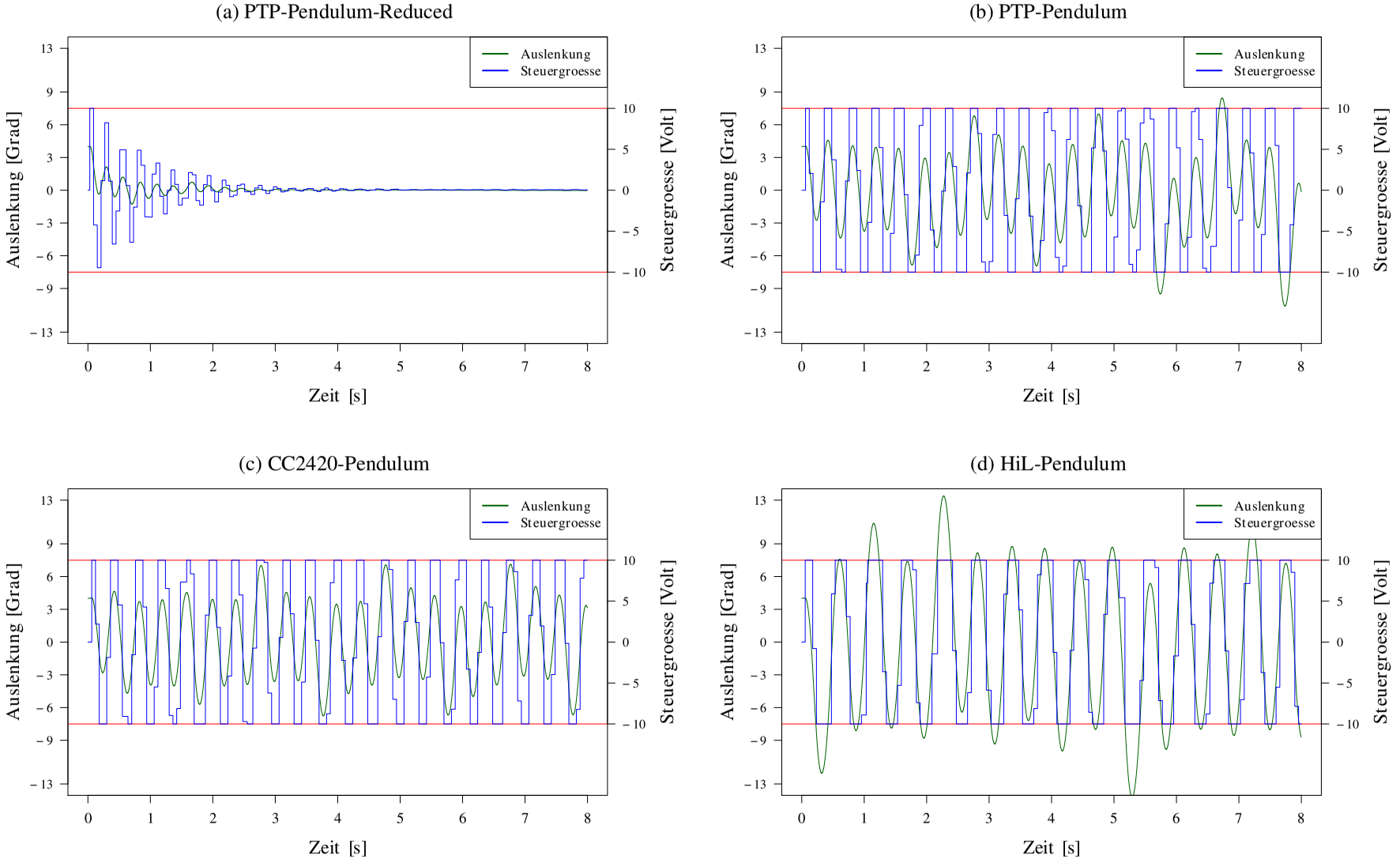


Abbildung 12.8: Darstellung der Aussteuerung des Pendels sowie der angelegten Steuergröße der verschiedenen konkreten Simulationssysteme bei Verwendung eines PID-Reglers.

Fazit

Wie die beiden ausgewählten Beispiele für Regler illustrieren, ist das verteilte Regelungssystem sehr anfällig für Verzögerungen sowohl in Bezug auf die Verarbeitung als auch die Kommunikation. Insbesondere die Berücksichtigung von Verarbeitungsverzögerungen durch die HiL-Simulation zeigt, wie groß diese Auswirkungen auf die Regelgüte und damit das Gesamtverhalten des Systems sein können. Genauere Analysen ergaben, dass die Verarbeitungsverzögerung dazu führt, dass der Controller nach dem Empfang eines Sensorwertes nicht rechtzeitig die neue Steuergröße berechnen kann, um diese direkt in dem auf den Sensorslot folgenden Slot versenden zu können. Somit ergibt sich eine zusätzliche Verzögerung von 30 ms, bis die neue Steuergröße an den Aktuator übermittelt werden kann. Die Ursachen sind sowohl im Design des Ablaufplans für die Kommunikation als auch innerhalb des SDL-Systems zu suchen. Da eine genaue Untersuchung des SDL-Systems den Rahmen sprengen würde, sei an dieser Stelle auf die Dissertation von Dennis Christmann [Chr15] verwiesen. Diese setzt sich im Detail mit Optimierungen von SDL-Systemen hinsichtlich der Echtzeitfähigkeit voraus. Seine Arbeit umfasst ebenfalls eine Performance-Evaluation der SDL-Systeme des inversen Pendels und verwendet hierfür ebenfalls HiL-Simulationen mit FERAL.

Insgesamt wird deutlich, dass sowohl beim Entwurf der Regler als auch deren Parametrisierung Kommunikations- und Ausführungsverzögerungen in geeigneter Weise adressiert und deren Auswirkungen auf das Verhalten des Gesamtsystems betrachtet und berücksichtigt werden müssen. Ein möglicher Ansatz, dies zu tun, wurde hier für das vernetzte Regelungssystem des inversen Pendels exemplarisch dargestellt. Hierbei stellt Virtual Prototyping mittels FERAL ein adäquates Hilfsmittel dar, welches beginnend beim ersten Entwurf des verteilten Regelungssystems und dessen einzelner funktionaler Komponenten bis hin zu den abschließenden Tests und auch bei einer Abstimmung der Parameter des Reglers sinnvoll eingesetzt werden kann.

In Bezug auf die beiden für die Evaluation verwendeten Regler muss festgestellt werden, dass sowohl PD- als auch PID-Regler sehr einfache Reglertypen und für dieses Szenario nicht geeignet sind. Sie dienten hier primär der Illustration des Vorgehens, ohne allzu tief in die Regelungstheorie einsteigen zu müssen. Bei den im Rahmen des DFG-Projektes entwickelten Reglern handelt es sich um deutlich kompliziertere zustandsbasierte Regler, die neben der Kompensation der Auslenkung auch die Bewegungen des Wagens und die Begrenzungen der Wegstrecke berücksichtigen. Zusätzlich arbeitet der Regelkreis beim DFG-Projekt mit einer höheren Abtastrate, um hierüber eine höhere Regelgüte bei den eingesetzten Reglertypen zu erreichen.

12.4 Zusammenfassung

Wie das Beispiel des inversen Pendels illustriert, können Verzögerungen – unabhängig davon, ob diese durch Ausführungs- oder Kommunikationsverzögerungen verursacht werden – das Gesamtverhalten verteilter Echtzeitsysteme stark beeinflussen. Beim inversen Pendel ist das komplette Spektrum der Auswirkungen sichtbar, von einer Veränderung der Regelgüte bis hin zum kompletten Versagen der Regelung, abhängig von den Regelparametern sowie den berücksichtigten Verzögerungen. Auch wenn dieses Beispiel keine Verallgemeinerung erlaubt, so verdeutlicht es dennoch, dass beim Einsatz von Virtual Prototyping, insbesondere in den späteren Entwicklungsphasen, sehr genaue Simulationsmodelle in Bezug auf Kommunikation und Plattform benötigt werden, um realistische und belastbare Simulationsergebnisse zu erhalten. Insofern stellen HiL-Simulationen eine sinnvolle Ergänzung von FERAL dar und bilden einen Zwischenschritt bei der Integration, welcher sowohl hohe Genauigkeit bietet als auch die Fehlersuche erleichtert (im Vergleich zu einer Suche im integrierten System).

Unserer Meinung nach hilft der iterative Entwicklungsansatz, gestützt durch Virtual Prototyping und Simulationen, komplexe verteilte Systeme zu realisieren und dabei das Risiko von Fehlschlägen zu minimieren. Die schrittweise Steigerung der Komplexität sowohl von Verhaltensmodellen als auch der simulierten Umgebung in Verbindung mit dem Umstand, dass jedes dieser so entstandenen konkreten

Simulationssysteme ausführbar ist, erlaubt es, Fehler früh einzugrenzen, zu identifizieren und zu beheben. Durch dieses Vorgehen lässt sich das Gesamtverhalten eines verteilten Systems über die komplette Entwicklung hinweg mit Simulationen testen und evaluieren. HiL-Simulationen helfen dabei, die abschließende Integration vorzubereiten, und bieten eine Möglichkeit, die Einflüsse von Ausführungsverzögerungen auf das Gesamtsystem sehr genau zu simulieren. Gleichzeitig bleiben die Vorteile einer Simulation, wie Reproduzierbarkeit, Ausschluss von Schäden oder Gefährdungen der physikalischen Umwelt erhalten.

Aktuell ist die Unterstützung von HiL-Simulationen in FERAL auf die *Imote 2*-Plattform sowie auf SDL-Spezifikationen beschränkt. Doch durch den modularen Aufbau der HiL-Runtime und deren Unterteilung in HiL-Base und HiL-Glue lassen sich weitere Techniken zur Beschreibung von Funktionalitäten leicht integrieren. Eine Alternative zu HiL-Simulation ist die Simulation einer kompletten Hardwareplattform. An diesem Aspekt arbeitet die Arbeitsgruppe Wehn mit ihrer Integration von Synopsis. Auch dies erlaubt sehr genaue Aussagen bzgl. auftretender Verzögerungen, erfordert hierfür jedoch sehr genaue Modelle der Plattform und ist mit entsprechend hohen Laufzeitaufwänden bei der Simulation verbunden. Ein Vorteil dieses Ansatzes besteht in der Möglichkeit, geeignete Zielplattformen vergleichend zu evaluieren, ohne dass diese bereits existieren müssen.

Da abschließende Tests jedoch auch stets auf der realen Zielplattform erfolgen müssen, z. B. auch, um Fehler in der Modellbildung oder Ungenauigkeiten auszuschließen, lassen sich HiL-Simulationen und Plattform-Simulationen sinnvoll kombinieren. Dennoch entbindet der Einsatz von Simulationen, seien es HiL-Simulationen auf der realen Hardware oder unter Verwendung von Plattform-Simulatoren, natürlich nicht davon, abschließend das Gesamtsystem auf der realen Zielplattform unter realistischen Bedingungen zu testen. Trotzdem helfen beide Techniken, einen großen Teil der Fehler bereits frühzeitig zu entdecken und zu beheben.

13 Stand der Technik

Die Kopplung von Simulatoren ist eine erprobte und sowohl in der Forschung als auch der Literatur häufig angewendete und beschriebene Strategie, wenn es um die Evaluation und Validierung komplexer Systeme geht, die unterschiedliche (interdisziplinäre) Techniken oder Forschungsgebiete miteinander verbinden. Die Kopplung spezialisierter Simulatoren kommt häufig dann zur Anwendung, wenn die exakte Simulation eines einzelnen isolierten Aspektes nicht ausreicht, da die Teilgebiete in komplexer Form zusammenwirken und das Gesamtverhalten des Systems beeinflussen. Die Verwendung spezialisierter Simulatoren erlaubt die genaue Simulation einzelner Aspekte, während die Kopplung der Simulatoren das Zusammenwirken abbildet. Der Einsatz und die Entwicklung einer Simulatorkopplung gegenüber eines eigenen Simulators, welcher alle fraglichen Aspekte direkt berücksichtigt, erweist sich dann als sinnvoll, wenn entsprechende spezialisierte Simulatoren bereits existieren sowie hinreichend etabliert und zuverlässig sind. In diesem Fall reduziert die Kopplung den Aufwand gegenüber einer Eigenentwicklung erheblich. Dies gilt insbesondere, wenn ein Framework, wie FERAL, den Entwickler bei der Kopplung der Simulatoren unterstützt. Der Aufwand für die Integration spezialisierter Simulatoren amortisiert sich dadurch, dass diese auch in anderen Simulationsszenarien (und auch in Verbindung mit anderen integrierten Simulatoren) wiederverwendet werden können.

In wissenschaftlichen Publikationen finden sich viele Beispiele, in denen zwei oder drei spezialisierte Simulatoren – häufig aus unterschiedlichen Anwendungsdomänen – miteinander gekoppelt werden, um ein bestimmtes Anwendungsszenario mittels Simulationen zu evaluieren. Allerdings beziehen sich die in diesen Arbeiten beschriebenen Ansätze stets auf die Kopplung konkreter Simulatoren mit Fokus auf dem Anwendungsszenario. Dementsprechend werden nur speziell auf das Szenario zugeschnittene Lösungen betrachtet und beschrieben. Eine Beschäftigung mit der Fragestellung, wie die allgemeinen Probleme bei der Kopplung von Simulatoren (z. B. unterschiedliche MOCCs, Interaktion, Austausch von Nachrichten zwischen den Simulatoren) gelöst werden können, erfolgt indes nicht. Gerade der spezielle Zuschnitt auf eine konkrete Problemstellung führt jedoch dazu, dass die Nutzung der entworfenen Simulatorkopplung für andere Arten von Szenarien und Problemstellungen, falls überhaupt, nur mit signifikantem Aufwand und Anpassungen möglich ist.

Das Fehlen geeigneter und etablierter Frameworks für die Kopplung von Simulatoren begünstigt natürlich die Entwicklung von speziell auf die Aufgabenstellung zugeschnittenen Kopplungen. Dies führt jedoch dazu, dass immer wieder die gleichen Problemstellungen gelöst werden müssen. Hierzu gehört die Definition von Schnittstellen für den Austausch von Informationen und Nachrichten zwischen den Simulatoren, ebenso wie die Entwicklung von Ausführungsmodellen zur Sicherstellung eines korrekten Simulationsablaufs. Zudem erlaubt ein Framework die Wiederverwendung einmal angebundener Simulatoren in anderen Szenarien, aufgrund der einheitlichen Schnittstellen. Leider sind die existierenden Frameworks, die sich mit der Kopplung von Simulatoren beschäftigen, in ihrem Anwendungsbereich stark eingeschränkt.

Die aktuell existierenden Arbeiten lassen sich im Wesentlichen in drei unterschiedliche Kategorien unterteilen: Dies sind zunächst konkrete Anwendungen von Simulatorkopplungen, bei denen die Abbildung eines spezifischen Szenarios sowie dessen Evaluation im Mittelpunkt steht (Kapitel 13.1). Kapitel 13.2 setzt sich mit dem aktuellen Stand der Forschung auseinander, in Bezug auf die für FERAL entwickelten CSCs für Feldbusse sowie der deklarativen Beschreibung von Kommunikationsverhalten, die von FERAL in Form der DBL für Bridges und Gateways genutzt wird. Abschließend betrachtet Kapitel 13.3 ausgewählte Frameworks für Simulationen und Simulatorkopplungen.

13.1 Ausgewählte Problemstellungen und Lösungsansätze für die Kopplung von Simulatoren

Wir beginnen in diesem Abschnitt mit der Beschreibung ausgewählter Beispiele für Simulatorkopplungen aus der Literatur und widmen uns dann Arbeiten, die in direktem Bezug zu Simulatoren und deren Kopplung (SDL, Matlab Simulink) stehen, deren Integration in FERAL Gegenstand dieser Arbeit ist.

Ausgewählte Beispiele für die Anwendung von Simulatorkopplungen

Erste Ansätze, die Kopplung von Simulatoren zur Lösung interdisziplinärer Aufgabenstellungen zu nutzen, finden sich in den Arbeiten von Wünsche et al. [WCSW97] aus dem Jahr 1997. In dieser Arbeit werden die beiden Simulatoren *SABER* und *ANSYS* gekoppelt, um die Performance komplexer Mikrosysteme zu evaluieren. Schwerpunkt des *Microsystems Design* ist die Integration von elektronischen, mechanischen und thermalen Komponenten innerhalb eines einzigen Chips bzw. auf sehr kleinem Raum. Die Entwicklung solcher Systeme erfordert kontinuierliche Möglichkeiten, das System während des gesamten Entwicklungsprozesses mittels Simulationen analysieren zu können, beginnend mit den ersten Modellen aus frühen Entwurfsphasen bis hin zu fertigen Prototypen.

Wünsche et al. diskutieren in [WCSW97] hierzu zwei unterschiedliche Ansätze: Der erste Ansatz besteht darin, das Verhalten eines Mikrosystems mit Hilfe eines einzigen Werkzeugs zu modellieren und anschließend zu simulieren. Dies würde jedoch die Entwicklung und Integration sämtlicher Komponenten (elektronischer, mechanischer sowie thermaler Natur) sowie deren physikalischer Eigenschaften in einem einzigen Modell notwendig machen. Die Entwicklung eines solchen Modells (inklusive des Verhaltens der einzelnen Komponenten) könnte, laut Wünsche, entweder mit VHDL-AMS oder mit der Hilfe von elektrischen Schaltkreisen (repräsentiert z. B. durch Netzlisten) erfolgen. Die Simulationen und Evaluationen wären dann mit Systemsimulatoren bzw. elektrischen Schaltungssimulatoren wie Spectre, ELDO oder SABER möglich. Wünsche et al. kommen zu dem Schluss, dass die Modellierung mit diesen Ansätzen zu aufwändig und kompliziert ist, da z. B. eine Abstraktion von den normalerweise eingesetzten partiellen Differentialgleichungen hin zu gewöhnlichen Differentialgleichungen erforderlich wird.

Aus diesem Grund entscheiden sich Wünsche et al. stattdessen für eine Lösung mittels Simulatorkopplung. Hierbei kommen für die unterschiedlichen Arten von Komponenten des Mikrosystems jeweils spezielle Simulatoren zum Einsatz, um deren Verhalten möglichst genau zu evaluieren. Die Simulation elektronischer Systemkomponenten erfolgt mit dem Schaltungssimulator *SABER*. Für mechanische Komponenten (wie z. B. mechanische Sensoren) oder thermale Effekte wird die Finite-Elemente-Methode (FEM) verwendet, hierfür kommt der *ANSYS* Simulator zum Einsatz.

Die Kopplung wird durch den Umstand ermöglicht bzw. vereinfacht, dass die beiden Simulatoren *SABER* und *ANSYS* ein identisches MOCC implementieren. So basieren beide Simulatoren auf einem zeitgetriggerten Ausführungsmodell und berechnen das Verhalten jeweils auf Basis diskreter Zeitschritte. Darüber hinaus erlaubt *ANSYS* die erneute Auswertung eines Zeitschrittes unter Berücksichtigung veränderter Stimuli. Diese beiden Umstände nutzen Wünsche et al. aus, um hierüber die Interaktion zwischen den Simulatoren und somit auch den von diesen simulierten Komponenten zu realisieren. Um eine möglichst effiziente Simulation zu gewährleisten, setzt [WCSW97] auf den von ihnen entwickelten *Automatic Time Step* Algorithmus. Dieser beruht auf der Erkenntnis, dass es aufgrund der diskreten Zeitschritte und der eigentlich vorhandenen Wechselbeziehungen der simulierten Aspekte (Leistungsaufnahme und Temperatur) bei dieser Form der Simulation zwangsläufig zu Ungenauigkeiten kommt.

Der Automatic Time Step Algorithmus passt die Schrittbreite der Simulation dynamisch an die gemessene Ungenauigkeit an. Überschreitet diese einen bestimmten Schwellwert, so wird der Zeitschritt sofort beendet und basierend auf dem Zeitpunkt der Verletzung eine neue obere Schranke für die Länge des Simulationsschrittes bestimmt. Mit dieser neuen Schranke wird der Simulationsschritt (teilweise) wiederholt. Dies führt dazu, dass bei kleineren resultierenden Ungenauigkeiten die Zeitschritte automatisch größer

gewählt werden und nur, sofern notwendig, eine feingranulare Auflösung bei der Simulation eingesetzt wird.

So konnten Wünsche et al. den Zusammenhang zwischen Leistungsaufnahme und resultierender Temperatur innerhalb von komplexen Mikrosystemen simulieren. Der Vorteil dieses Ansatzes besteht darin, die Möglichkeit klassischer Simulationsansätze von elektrischen Schaltungen mit denen der FEM zu kombinieren und auf diese Weise sehr exakte Ergebnisse erzielen zu können. Ein Nachteil, auf den Wünsche hinweist, ist allerdings die sehr hohe Laufzeit, welche insbesondere aus der Anwendung der FEM resultiert.

Die Autoren Frank und Weigel präsentieren in [FW07] ihren Ansatz für Simulationen zur Ermittlung elektromagnetischer Kompatibilität von Baugruppen. Anwendungsgebiete ergeben sich beispielsweise im Bereich von Steuergeräten im Automobilbereich. Hierfür koppeln die Autoren die beiden Simulatoren SLSim [Sim] und SMASH [Dol].

SLSim ermöglicht die Simulation von SPICE (Simulation Program with Integrated Circuit Emphasis) Netzlisten. Diese beschreiben elektrische Schaltungen, d.h. sowohl Bauelemente als auch deren elektrische Verbindungen. Mit Hilfe von SLSim lassen sich die analogen Vorgänge innerhalb der Schaltung simulieren (Spannungsverläufe, Ströme, Induktion von Magnetfeldern etc.). Dem gegenüber steht mit VHDL-AMS eine Hardwarebeschreibungssprache, welche die Modellierung des logischen Verhaltens aktiver Komponenten (Mikrocontroller, Logikbausteine) ermöglicht. Diese VHDL-Beschreibungen können mit Hilfe von SMASH simuliert werden. Durch ihre in [FW07] vorgestellte Kopplung von SLSim und SMASH ermöglichen die Autoren die integrierte Simulation kontinuierlicher SPICE-Modelle zusammen mit den diskreten VHDL-Modellen. Hierdurch gelingt es, das Gesamtverhalten einer Baugruppe inkl. der auftretenden Spannungsverläufe, Ströme, deren Auswirkungen sowie das Zusammenwirken mit den aktiven Komponenten zu simulieren.

Bei dieser Simulatorkopplung übernimmt der SLSim Simulator für SPICE-Modelle die Koordination und Steuerung der Gesamtsimulation. Der SMASH-Simulator wird über eine spezielle für SLSim entwickelte Erweiterung aufgerufen und gesteuert. Hierüber erfolgt ebenfalls die Synchronisation der beiden Simulatoren sowie der Austausch von Nachrichten (z. B. die Simulationsergebnisse eines Simulationsschritts). Die Synchronisation der Simulatoren nutzt einen eigenen Algorithmus, der, falls nötig, die Länge des Simulationszeitschrittes der Simulatoren solange verkleinert, bis die Abweichung, der von den beiden Simulatoren ermittelten Ergebnisse unterhalb eines Schwellwertes liegen. Dieser Schritt ist, wie die Autoren anmerken, mit erheblichen Laufzeitaufwänden verbunden, da Simulationen für einen Zeitschritt mehrfach ausgeführt werden müssen. Die Autoren sehen jedoch keine andere Lösung bei der verwendeten Variante der Kopplung kontinuierlicher und diskreter Modelle.

Die bereits vorgestellten Beispiele für Simulatorkopplungen haben stets Simulatoren mit unterschiedlichen Anwendungsgebieten und Schwerpunkten gekoppelt; die Arbeit von Siddique et al. [SKG07] demonstriert, dass auch die Kopplung von Simulatoren der gleichen Anwendungsdomäne gewinnbringend und sinnvoll eingesetzt werden kann. In [SKG07] stellen Sie ihre vertikale Kopplung zweier Netzwerksimulatoren vor, deren Schwerpunkte auf unterschiedlichen Schichten des OSI-Modells abzielen.

Die von ihnen entwickelte Simulatorkopplung hat zum Ziel, die als Teil des IEEE 802.11h Standards [EE03] spezifizierte Dynamic Frequency Selection (DFS) Methode für die Übertragung von Rahmen zu evaluieren. Der IEEE 802.11h Standard erweitert IEEE 802.11 (WLAN) um die Möglichkeit von Kanalwechsel sowie der Anpassung der Sendeleistung (TPC – Transmit Power Control). Auf diese Weise kann auf einen anderen Kanal gewechselt werden, sofern dieser bessere Übertragungsbedingungen bietet. TPC erlaubt es hingegen, den Signal-Rauschabstand zu optimieren. Um diesbezüglich Evaluationen und Simulationen durchführen zu können, verwenden Siddique et al. eine vertikale Kopplung des Netzwerksimulators ns-2 mit dem WRAP2-Simulator (WARP2 – Wireless Access Radio Protocol 2, [SKG+06]).

Bei dieser Kopplung ist der WRAP2 für die Simulation des MAC-Layer und des Physical-Layer eines IEEE 802.11h-Netzwerkes verantwortlich, während der ns-2 die Simulation von Logic-Link-, Network-, Transport- sowie des Application-Layer übernimmt. Für die Steuerung ist der ns-2-Scheduler

verantwortlich. Dieser wurde dahingehend modifiziert, dass er die Übertragung von Rahmen nicht mit dem eigenen MAC-Layer und Medienmodell simuliert, sondern stattdessen die erzeugten Rahmen an den Load-Generator des WARP2 übergibt. Dieser simuliert die Übertragung der Rahmen unter Verwendung seines eigenen Modells. Umgekehrt übergibt der WARP2 Simulator empfangene Rahmen an den ns-2-Scheduler, sodass dieser die Rahmen durch den eigenen Stack weiterverarbeiten lassen kann.

Auch im Bereich der Entwicklung von *Car-to-X*- oder *Car-to-Car*-Anwendungen sind Simulatorkopplungen ein häufig anzutreffender Lösungsansatz. Eine beliebte Konstellation ist die Kopplung eines Verkehrssimulators, wie SUMO [KEBB12] und des ereignisbasierten Simulators OMNeT++ [Var01], für die Netzwerksimulation [SSK09, SGD11]. SUMO simuliert die Bewegungen von Fahrzeugen innerhalb eines komplexen Straßennetzes sowie die Interaktion der Fahrzeuge.

Schuhmacher et al. [SSK09] ergänzen in ihrer Kopplung dieses Duo um ein 3D Raytracing Tool, mit dessen Hilfe sie die akkurate Ausbreitung der Funkwellen (Abschattung, Reflektionen etc.) innerhalb von urbanen Umgebungen auf Basis von 3D-Modellen der Gebäude simulieren.

Lochert et al. in [LSC+05] verwenden für ihre Forschungen im *Car-to-Car*-Bereich ebenfalls eine Simulatorkopplung, jedoch wollen diese, im Gegensatz zu [SSK09], auch die Auswirkungen der *Car-to-Car*-Kommunikation (z. B. Stau- und Unfallwarnungen) auf den Verkehrsfluss innerhalb der Simulation abbilden. Zu diesem Zweck koppeln sie einen Simulator für die Verkehrsbewegungen (VSSIM [PTVar]) sowie Matlab Simulink mit dem Netzwerksimulator ns-2. In dieser Konstellation nutzen Lochert et al. Matlab Simulink für die Entwicklung einer Anwendung (in Form eines Simulink Modells), welches in geeigneter Art das Verhalten der Fahrer widerspiegelt und auf Basis der *Car-to-Car*-Kommunikation Entscheidungen hinsichtlich der zu fahrenden Route trifft.

SDL zur Spezifikation von Verhaltensmodellen in Simulatorkopplungen

Die Kopplung von ITU-T's Specification and Description Language (SDL) [Int12] mit anderen Simulatoren findet sich in verschiedenen Arbeiten wieder. Ein Beispiel hierfür ist die in [KGGR05] beschriebene Kopplung von SDL mit dem Netzwerksimulator ns-2 [USCar]. Ziel dieser Kopplung ist die Performance-Evaluation von mit SDL spezifizierten Netzwerkprotokollen [KGGR05]. Hierzu wird die SDL-Spezifikation zunächst mit Hilfe des in der IBM Rational SDL-Suite [IBMar] enthaltenen Code Generators in C-Code übersetzt. Der generierte Code implementiert das Verhalten eines Knotens des Netzwerkes. Die Kommunikation, d.h. die Übertragung der Rahmen (Ausbreitung, Interferenzen), wird dann durch den ns-2 simuliert. Als Vorteil dieses Ansatzes führen die Autoren an, dass der gleiche C-Code sowohl für das Produktivsystem als auch für die Simulation zum Einsatz kommt und daher aussagekräftige Ergebnisse zu erwarten sind.

Bei der in [KGGR05] entworfenen Kopplung übernimmt der ns-2-Scheduler die Steuerung und Koordinierung des gesamten simulierten Systems. Der ns-2-Scheduler sequenzialisiert die Ausführung der einzelnen SDL-Systeme, sodass zu einem Zeitpunkt jeweils nur maximal ein SDL-System (Netzwerkknoten) ausgeführt wird. Für die Interaktion mit den SDL-Systemen nutzt der ns-2-Scheduler spezielle Nachrichten, die zur Synchronisation, Ausführungskontrolle und dem Austausch von Daten dienen.

Um dies zu realisieren, wird zunächst jedes SDL-System in ein eigenständiges SDL-Modul überführt. Ein solches SDL Modul besteht aus dem generierten Code des SDL-Systems, dem SDL Environment Interface sowie dem SDL-Kernel. Das SDL Environment Interface ermöglicht die Interaktion des SDL-Systems mit dem ns-2 über spezielle SDL-Signale (z. B. zum Versenden eines Rahmens), während die Aufgabe des SDL-Kernel darin besteht, die Signale innerhalb des SDL-Systems weiterzuleiten sowie die Ausführung von Transitionen zu triggern. Darüber hinaus stellt der SDL-Kernel die notwendigen Funktionen für den Austausch von Nachrichten zwischen dem ns-2 und dem SDL Environment Interface zur Verfügung. Die Ausführung des SDL-Systems wird durch den ns-2-Scheduler gesteuert, welcher hierfür Kontrollnachrichten mit dem SDL-Kernel austauscht. Hierüber wird kontrolliert, wann der SDL-

Kernel Transitionen ausführt und welche Timer innerhalb des SDL-Systems existieren, sodass der ns-2-Scheduler das SDL-System rechtzeitig ausführen kann.

Damit der ns-2-Scheduler diese Aufgaben erfüllen kann, wurden spezielle *ns-Module* entwickelt und zu der ns-2 Bibliothek hinzugefügt. Zu diesen Erweiterungen gehört der *SDL_Agent*. Dieser realisiert die Interaktion mit einem SDL-Modul in einer Form, welche für den ns-2-Scheduler selbst transparent ist. Hierzu wird für jedes SDL-Modul ein eigener *SDL_Agent* erzeugt, welcher das SDL-Modul gegenüber dem ns-2-Scheduler repräsentiert. Hierbei ist der *SDL_Agent* dafür verantwortlich, das SDL-Modul zu laden sowie die Aktionen des ns-2-Schedulers auf entsprechende Kontrollnachrichten abzubilden und an dieses zu übermitteln. Hierdurch kann der ns-2-Scheduler ohne Anpassungen verwendet werden. Weitere *ns-Module* erlauben mit SDL die Entwicklung von Protokollen auf unterschiedlichen Abstraktionsebenen des (von ns-2 simulierten) Protokollstacks. Sowohl ns-2 als auch jedes SDL-Modul werden als eigenständiger Betriebssystemprozess ausgeführt. Die Inter-Prozess-Kommunikation zwischen dem SDL-Kernel und dem jeweiligen SDL-Modul innerhalb des ns-2 (bzw. dem *SDL_Agent*) erfolgt mittels *Named Pipes*.

Diese Simulatorkopplung wurde in [KGGR05] erfolgreich für die Evaluation von DSDV (Distance-Sequenced Vector Routing) auf Basis von IEEE 802.11 genutzt. Hierbei wurde das Verhalten der Knoten, inkl. des DSDV-Routings mit SDL spezifiziert, während die Simulation des Netzwerkes (Verzögerung, Ausbreitung der Signale, Signalstärke, Reichweite sowie Kollisionen) durch den ns-2 erfolgte.

Neuere Arbeiten, die sich ebenfalls mit der Kopplung von SDL-Systemen mit Netzwerksimulatoren beschäftigen, sind in [BF10] und [BF12] beschrieben. Brumbulli und Fischer setzen für die Erstellung der SDL-Systeme sowie die Codegenerierung auf Pragmadev's Real Time Developer Studio (RTDS, [Praar]). Die mittels RTDS entwickelten SDL-Systeme werden mit dem ns-3 [ns3ar], dem offiziellen Nachfolger des ns-2, gekoppelt.

Bei der in [BF10] beschriebenen Kopplung wird, wie auch in [KGGR05], die Ausführung des Gesamtsystems durch den *ns*-Scheduler gesteuert. Auch diese Lösung verzichtet auf eine Modifikation des ns-3-Kerns (dem ns-3-Scheduler). Stattdessen haben es sich Brumbulli und Fischer zum Ziel gesetzt, den durch Paragmadev generierten C++-Code auf automatisierte Weise so zu modifizieren, dass dieser direkt als Modul innerhalb des ns-3 übersetzt und von diesem ausgeführt werden kann. Hierfür erweitern Brumbulli und Fischer den HUB Transcompiler [AEF+09, FKAE09] so, dass dieser den – aus der SDL-Spezifikation – erzeugten C++-Code so transformiert, dass das Ergebnis zusammen mit zusätzlichem Glue-Code ein vollständiges ns-3-Modul ergibt. Dieses repräsentiert einen Netzwerkknoten und rangiert innerhalb des ns-3-Protokollstacks auf der Abstraktionsebene eines Application Layer Protokolls.

Der Vorteil der in [BF10] vorgestellten Lösung im Vergleich zu der Lösung in [KGGR05] besteht darin, dass die verschiedenen SDL-Systeme direkt in den ns-3 integriert werden. Das bedeutet, ns-3 und SDL-Systeme teilen sich einen gemeinsamen Prozessraum, sodass eine direkte Interaktion über Methodenaufrufe erfolgen kann. Die aufwändige Inter-Prozess-Kommunikation über den Austausch nach Nachrichten zwischen verschiedenen Prozessen entfällt, wodurch sich der Overhead bei der Ausführung erheblich reduziert.

In [BF12] widmen sich Brumbulli und Fischer, basierend auf ihrer vorausgegangenen Arbeit in [BF10], der Fragestellung, wie sich der Aufbau eines Simulationssystems und dessen Konfiguration mit Hilfe von modellgetriebenen Techniken realisieren lässt. Dies umfasst die Festlegung der existierenden Knoten, die Zuordnung von SDL-Systemen zu den Knoten sowie den zu verwendenden Protokollstack innerhalb des ns-3. Hierbei adressieren die Autoren einen Teilaspekt, welcher auch für FERAL relevant ist – wie kann automatisiert aus einem konkreten Simulationssystem eine mit FERAL ausführbare Szenarienbeschreibung erzeugt werden (vgl. Kapitel 10.1). Brumbulli und Fischer nutzen die von RTDS bereitgestellten Deployment Diagramme, um den Aufbau eines Simulationssystems (Knoten, Zuordnung der SDL-Systeme, Protokollstack, Parameter, etc.) vollständig zu beschreiben. Die graphische Festlegung der Topologie sowie die räumliche Anordnung der Knoten werden von Brumbulli und Fischer durch die Einbindung externer Werkzeuge unterstützt.

Die Informationen des Deployment Diagramms dienen dabei als Grundlage für die Erzeugung des von ns-3 benötigten Glue-Codes. Dieser bildet zusammen mit dem ns-3-Code sowie dem generierten Code aus den SDL-Spezifikationen das vollständige Simulationssystem. Eine manuelle Erstellung des Simulationssystems – oder wie bei FERAL der Szenariobeschreibung – durch einen Entwickler wird hierdurch obsolet. Langfristig planen Brumbulli und Fischer eine Erweiterung ihres Konzepts auch für andere Simulatoren.

Die in [KGGR05] und [BF10] beschriebenen Kopplungen von SDL beschränken sich auf den Network Simulator in den Versionen 2 bzw. 3. Demgegenüber ermöglicht die Integration von SDL in unser Framework FERAL die Simulation von SDL-Systemen in Kombinationen mit beliebigen anderen, ebenfalls in FERAL integrierten Simulatoren. Das Gleiche wird für den ns-3 erreicht, welcher ebenfalls in FERAL integriert wurde (vgl. Kapitel 11). Die Integration von ns-3 und die SEnF-Treiber für ns-3 unterstützen, wie die Umsetzung aus [KGGR05], die Verwendung von SDL zur Entwicklung von Protokollen auf unterschiedlichen Abstraktionsebenen des OSI-Schichtenmodells.

Die Interaktion von SDL-Systemen mit beliebigen anderen Simulatoren wird über den entwickelten ForwardSignals-SEnF-Treiber (Kapitel 8.4) ermöglicht. Dieser ist universell und nicht auf bestimmte Simulatoren beschränkt, sodass auch eine Interaktion mit zukünftig integrierten Simulatoren gewährleistet ist. Zusätzlich bieten wir spezielle SEnF-Treiber für die CAN- und FR-CSCs an, welche eine direktere Interaktion mit einzelnen spezifischen Simulatoren erlauben und die Schnittstelle realer Hardwaretreiber sowie deren Ansteuerungslogik und Verhalten nachbilden. Hierdurch lassen sich SDL-Systeme entwickeln, die unverändert sowohl mit FERAL simuliert als auch für reale Zielplattformen übersetzt werden können.

Anwendungen von Matlab Simulink in Simulatorkopplungen

Mathworks Matlab Simulink [Mat12] ist ein in Forschung und Industrie weit verbreitetes und etabliertes Softwarepaket für die modellbasierte Entwicklung und Simulation dynamischer und eingebetteter Systeme [NL02]. Simulink kommt insbesondere beim Entwurf mathematischer Modelle sowie bei der Entwicklung und Simulation von Regelungssystemen und Reglern zum Einsatz. Der Funktionsumfang von Matlab Simulink kann darüber hinaus durch zahlreiche Toolboxes erweitert werden. Diese stellen spezielle Blockbibliotheken für unterschiedliche Anwendungsdomänen zur Verfügung, wie beispielsweise Regelungstechnik, Luftfahrt oder Automotive. Spezielle Toolboxes, wie CarMaker, CarSim oder DRIVE, erlauben die Simulation der dynamischen Eigenschaften von PKW und eigenen sich beispielsweise für die Entwicklung und Evaluation von Assistenzsystemen. Die Toolboxes (und in beschränktem Maße die S-Functions) bieten zwar die Möglichkeit, Matlab Simulink zu erweitern, allerdings sind die Entwickler von Toolboxen oder S-Functions auf die Ausführungssemantik und das Ausführungsmodell von Simulink festgelegt.

Nossal und Lang beschreiben in [NL02] einen modellbasierten Entwicklungsansatz für X-by-Wire Anwendungen, der auf Matlab Simulink basiert. Der erste Entwicklungsschritt besteht in dem Entwurf eines rein funktionalen Modells der Anwendung. Hierbei wird das Verhalten des Gesamtsystems mit Hilfe von Simulink-Blöcken beschrieben, die direkt über Signalpfade miteinander verbunden sind und interagieren. Die Bildung isolierter funktionaler Komponenten und deren Verteilung auf unterschiedliche Knoten sowie die Realisierung der Interaktion über ein Kommunikationssystem wird auf dieser Abstraktionsstufe noch nicht berücksichtigt. Im nächsten Schritt werden Blöcke zu funktionalen Komponenten zusammengefasst, die jeweils einem Knoten zugeordnet werden. Dann werden die Signalpfade zwischen den funktionalen Komponenten aufgetrennt und durch spezielle Blöcke ersetzt, welche das Verhalten eines konkreten Kommunikationssystems (z. B. CAN oder FlexRay) simulieren. Entsprechende Produkte, die Simulink um solche funktionalen Blöcke erweitern, sind z. B. xCom von DECOMSYS oder die Vehicle Network Toolbox von MathWorks selbst. Ein anderer Ansatz wird in [DDY10, Vec08] beschrieben. Hier wird Simulink mit CANoe, einem Simulator für Kommunikationssysteme aus dem Automobilbereich

der Vector Informatik GmbH, gekoppelt. Das funktionale Verhalten des dort betrachteten Steer-by-wire Systems wird mit Simulink, der FlexRay-Bus mit CANoe simuliert.

Li et al. verzichten ganz auf den Einsatz von Toolboxes und anderer Simulatoren und zeigen in [LWL08], dass die Entwicklung eines Verhaltensmodells des CAN-Busses auch direkt in Simulink möglich ist. In ihrer Arbeit nutzen sie das entwickelte Modell des CAN-Busses, um auf dieser Basis Performance-Simulationen durchzuführen. Die erhaltenen Ergebnisse wurden sowohl mit den Ergebnissen anderer Simulatoren als auch mit theoretischen Betrachtungen der Worst-Case Response Time verglichen, um deren Plausibilität zu prüfen. Eine Anwendung ihres Simulink Modells des CAN-Busses sehen Li et al. beispielsweise in der Evaluation von ECUs sowie allgemeinen Tests der CAN-Bus Konfiguration für verschiedene Szenarien.

Diesen Ansätzen [NL02, LWL08, DDY10] ist gemein, dass diese sich auf den Einsatz von Matlab Simulink als einzige Modellierungstechnik für funktionale Verhaltensmodelle beschränken. Hierdurch ist eine Simulation und Evaluation eines komplexen Systems nur möglich, sofern das gesamte System inklusive eines Modells der Umgebung oder Regelungsstrecke als Matlab Simulink Modell zur Verfügung steht oder es geeignete Erweiterungen in Form von Toolboxes gibt, welche die benötigten Modelle bereits zur Verfügung stellen. Solche Einschränkungen existieren beim Einsatz von FERAL nicht.

Bei dem in [NL02] beschriebenen Vorgehen wäre prinzipiell auch die Evaluation unterschiedlicher Kommunikationsmodelle denkbar, wie dies in FERAL möglich ist. Jedoch müssten hierbei die Blöcke, welche das jeweilige Kommunikationssystem repräsentieren, von Hand ausgetauscht und die anderen Blöcke entsprechend so modifiziert werden, dass eine Interaktion mit dem neuen Kommunikationssystem ermöglicht wird. Insgesamt wäre dieses Vorgehen, insbesondere bei der Evaluation mehrerer unterschiedlicher Kommunikationssysteme mit verschiedenen Paradigmen, ggf. mit verschiedenen Konfigurationsparametern, extrem aufwändig und fehleranfällig.

Zusammenfassung und Abgrenzung

Im Gegensatz zu den konkreten Beispielen für die Kopplung ausgewählter Simulatoren ist FERAL ein generisches Framework zur Kopplung beliebiger spezialisierter Simulatoren (vgl. Kapitel 11) und ist nicht auf ein bestimmtes Anwendungsszenario oder eine Domäne von Problemstellungen festgelegt. Einmal in FERAL integrierte Simulatoren können aufgrund der standardisierten Schnittstellen beliebig kombiniert werden, um die Anforderungen unterschiedlichster Simulationsszenarios abzubilden. Insofern unterscheidet sich FERAL von den vorgestellten Lösungsansätzen deutlich, da bei diesen in der Regel eine konkrete Problemstellung oder Klasse von Problemstellungen als Motivation für die Kopplung der Simulatoren dient. Grundlage dieser Lösungen bildet die Modifikation oder Erweiterung einer der beteiligten Simulatoren der Kopplung, sodass dieser die Ausführung der anderen Simulatoren kontrolliert und den Austausch von Nachrichten ermöglicht [SSK09, FW07, SKG07, LSC+05] oder entsprechende bereits existierende Schnittstellen genutzt wurden [NL02, DDY10]. Aufgrund der Spezialisierung der Kopplung auf eine konkrete Problemstellung ist eine Wiederverwendung in anderen Anwendungsszenarien häufig ausgeschlossen.

Infolgedessen müssen die Schwierigkeiten und Probleme bei der Kopplung (Synchronisation, Austausch von Nachrichten, Schnittstellen) von jedem Entwickler erneut gelöst werden. Im Gegensatz hierzu stellt FERAL bewährte Lösungen bereit, die bei der Integration von Simulatoren genutzt werden können. Hierbei handelt es sich sowohl um einheitliche Datentypen und Schnittstellen für die Interaktion zwischen Simulatoren als auch um Konzepte für deren Synchronisation (vgl. Kapitel 8.1, [BGFK13]). So erlaubt das Konzept der Direktoren beispielsweise die Kopplung von Simulatoren mit unterschiedlichen MOCCs und gewährleistet die semantische Korrektheit der Simulation.

Diese Wiederverwendbarkeit der einmal integrierten Simulatoren in beliebigen Simulationssystemen macht FERAL attraktiv und gestattet die Abdeckung vielfältiger interdisziplinärer Szenarien, ohne dabei auf bestimmte Anwendungsdomänen (wie z. B. die Simulation digitaler Schaltungen [WCSW97] oder

Car-to-X-Lösungen [SSK09]) beschränkt zu sein. Im Vergleich zu den vorgestellten Arbeiten gibt es bei FERAL keine Einschränkungen hinsichtlich der Art der Simulatoren, die sich mit diesem Framework koppeln lassen. So gestattet FERAL, anders als z. B. [SKG07], sowohl horizontale als auch vertikale Kopplungen.

Die Vielzahl der in FERAL bereits integrierten Simulatoren erlaubt es dem Benutzer, die für ihn und seine Aufgabenstellung am besten geeigneten Modellierungstechniken zu verwenden (wie z. B. Matlab Simulink [Mat12, NL02, LWL08]) und diese mit anderen Ansätzen, wie z. B. SDL, zu kombinieren. Aufgrund der Erweiterbarkeit von FERAL können zudem leicht weitere Simulatoren integriert werden, sofern die vorhandenen Techniken die Erfordernisse oder Wünsche nicht abdecken.

Ein besonderes Merkmal von FERAL besteht darin, innerhalb eines Simulationssystems Simulationskomponenten unterschiedlicher Abstraktionsstufen (oder aus unterschiedlichen Entwicklungsphasen) nutzen zu können. Auf diese Weise steht in allen Entwicklungsschritten ein ausführbares Simulationssystem für Evaluationen und Tests zur Verfügung. Dies gestattet es, die enthaltenen Simulationskomponenten schrittweise zu verfeinern und sukzessive durch ihre finalen Versionen zu ersetzen. Über die in Kapitel 10 vorgestellten Konzepte zur Austauschbarkeit von CSCs (Bridges und Gateways in Kombination mit der domänenspezifischen Sprache DBL zur Beschreibung des Kommunikationsverhaltens) ermöglicht FERAL zudem die einfache und effiziente Evaluation von Designalternativen hinsichtlich der Wahl von Kommunikationstechnologien. Im Gegensatz zu [NL02] und [LWL08] ersparen die von uns entwickelten Konzepte aufwändige Anpassungen der funktionalen Verhaltensmodelle.

Wie die hier vorgestellten Beispiele zeigen, ist die Kopplung von spezialisierten Simulatoren ein etablierter Lösungsansatz. Ebenso deutlich wird aber auch, dass das Fehlen geeigneter Frameworks dazu führt, dass viele Probleme der Interaktion stets aufs neue gelöst werden müssen und Kopplungen dementsprechend aufwändig sind. Erschwerend kommt hinzu, dass die hierbei entstandenen Kopplungen sich nur in geringem Maß wiederverwenden lassen, da sie auf spezifische Szenarien oder eine bestimmte Klasse von Problemstellungen zugeschnitten wurden.

13.2 Konzepte für die Simulation von Kommunikationstechnologien und Kommunikationsverhalten

Bei der Entwicklung von FERAL haben wir uns dafür entschieden, eigene Simulatoren für CAN und FlexRay zu entwickeln. Hierdurch konnte die Architektur der CSCs so entworfen werden, dass eine einfache Austauschbarkeit der CSCs innerhalb eines Simulationssystems begünstigt wird. Neben der gemeinsamen Architektur und Schnittstelle spielen Bridges und Gateways sowie unsere domänenspezifische Sprache DBL zur Spezifikation des Kommunikationsverhaltens (vgl. Kapitel 9 und 10) für die Evaluation von Designalternativen eine zentrale Rolle. Im Folgenden stellen wir diesen von uns entwickelten Konzepten bereits existierende Lösungen und Lösungsansätze für die Simulation von Feldbussen sowie die Spezifikation von Kommunikationsverhalten gegenüber.

Simulation von Kommunikationstechnologien im Bereich der Feldbusse

Im Bereich der Simulatoren für Feldbusse, insbesondere im Automobilbereich, gibt es bislang keinen ähnlich prominenten und frei verfügbaren Vertreter, wie es der ns-3 für die Simulation IP-basierter und drahtloser Netzwerke ist. Stattdessen existieren verschiedene kommerzielle Produkte, wie beispielsweise die Vehicle Network Toolbox von MathWorks oder xCom von DECOMSYS. Diese erweitern Matlab Simulink um Blockbibliotheken für die Simulation von FlexRay oder CAN-Bussen. Ein weiteres Beispiel ist TrueTime [CHL+03]. Bei diesem handelt es sich um einen freien Simulator für Feldbusse, welcher jedoch ebenfalls auf Matlab basiert.

Neben diesen vollständig auf Matlab fokussierten Lösungen existieren eigenständige Werkzeuge für die Simulation von Fahrzeugbussen. So bietet beispielsweise die Vector Informatik GmbH eine Vielzahl von ineinandergreifenden Werkzeugen an, die sowohl Entwurf, Ablaufplanung der Kommunikation, Simulationen und HiL-Lösungen involvieren und sowohl CAN als auch FlexRay unterstützen. Für die Simulation sowohl von CAN als auch FlexRay eignet sich z. B. CANoe [Vec]. Hierbei handelt es sich jedoch nicht um ein Framework, sondern einen in sich abgeschlossenen und vollständig in die Werkzeugkette von Vector integrierten Simulator, welcher zusätzlich Matlab Simulink für die Simulation von Verhaltensmodellen unterstützt [Vec08, DDY10]. Den kommerziellen Lösungen gemein ist die jeweils eng an das Portfolio des Herstellers gekoppelte Ausrichtung mit dem Schwerpunkt auf Integration und Interoperabilität der eigenen Produkte. Erweiterungen und Erweiterbarkeit stehen, im Gegensatz zu FERAL, nicht im Fokus des Interesses.

In der Literatur finden sich in Bezug auf CAN zwar häufig Ergebnisse aus Simulationen, wie z. B. in den Arbeiten [HG99, ZS97, NHN03]. Allerdings werden dort Simulationen primär zur Bestätigung der Korrektheit der zuvor beschriebenen Ansätze zur Analyse von Scheduling-Strategien, Deadlines, Worst-Case Verzögerungen oder probabilistischen Ansätzen, genutzt. Die verwendeten Simulatoren sind meist weder näher beschrieben noch verfügbar. Zudem decken diese häufig nur das jeweilige Demonstrationsszenario ab; die Simulation realistischer Netzwerke, insbesondere in Verbindung mit Verhaltensmodellen realer Komponenten, liegt nicht im Fokus dieser Arbeiten.

Für FlexRay existieren verschiedene Ansätze, die sich jeweils auf die Simulationen unterschiedlicher Komponenten eines FlexRay Busses konzentrieren. Es gibt verschiedene Simulationsmodelle für FlexRay Communication Controller in SystemC [KKAM08, KBH$^+$07] oder auch in reinem C [XZ08], die direkt auf einer PC-Plattform ausführbar sind. Zusätzlich existieren Modelle für FlexRay Transceiver von NXP oder CISC Semiconductor [KASW10], die in der Lage sind, das Verhalten konkreter Transceiver sowie Effekte auf dem Physical-Layer nachzubilden [GB07]. Karner et al. stellen diese Ansätze in [KASW10] einander gegenüber und präsentieren eine Lösung für die Simulation von FlexRay mittels *Co-Simulation Model Switching*. Hierbei werden verschiedene Simulationsmodelle mit unterschiedlichen Abstraktionsstufen gekoppelt und bei Bedarf oder einem speziellen Ereignis auf ein Modell mit höherem Detailgrad gewechselt. Die in [KASW10] vorgestellte Lösung koppelt Simulationsmodelle für den Physical-Layer (VHDL) mit schnellen, aber weniger detaillierten SystemC Modellen für den Data-Link-Layer (FlexRay Communication Controller) sowie die AutoSAR Middleware. Somit geht der Fokus dieses Simulators deutlich über die Zielsetzung der FR-CSC hinaus, eine reine Simulation auf Nachrichtenebene zu realisieren. Hierfür benötigt die Lösung aus [KASW10] jedoch auch entsprechend komplexe Simulationsmodelle sowie eine Simulationsumgebung zur Realisierung der Co-Simulation.

Neuere Ansätze, die beim Start der Entwicklung von FERAL noch nicht verfügbar waren, erweitern den ereignisbasierten Simulator OMNeT++ [Var01] um Simulationsmodelle für FlexRay [BSKS13, Eng12], CAN [MMT$^+$13, KMT14, Eng13b] und Real-time Ethernet [KBS$^+$13, SDKS11]. Im Rahmen dieser Arbeiten wurden sowohl für CAN als auch für FlexRay erste Betaversionen der entsprechenden Implementierungen veröffentlicht, sodass diese nun frei verfügbar sind. Mit FERAL vergleichbare Konzepte für die Austauschbarkeit der einzelnen Kommunikationstechnologien innerhalb eines Simulationssystems werden von diesen Arbeiten nicht adressiert, da die verschiedenen Simulatoren bzw. Erweiterungen sowie deren Funktionalität und Aufbau jeweils nur separat betrachtet werden.

Konzepte zur Beschreibung des Kommunikationsverhaltens

Zu dem von FERAL verfolgten Ansatz, mittels Bridges und Gateways funktionales und kommunikationsspezifisches Verhalten zu trennen, existiert mit dem *Fieldbus Exchange Format (FIBEX)* [Ass, ZS08, SS06, Cri07] ein ähnlicher Ansatz im industriellen Umfeld. Bei FIBEX handelt es sich um ein standardisiertes XML basiertes Datenaustauschformat zur Beschreibung von Fahrzeugboardnetzen der ASAM e.V. (Association for Standardization of Automation and Measuring Systems). Eine FIBEX-Beschreibung enthält,

neben der Konfiguration des Datenbusses, auch eine Definition der Topologie sowie deren Knoten. Neben dem statischen Aufbau erlaubt FIBEX auch die Spezifikation des Kommunikationsverhaltens. Dazu gehört die Festlegung der Eigenschaften von Frames sowie deren Payload und unter welchen Voraussetzungen ein Frame versendet/erzeugt wird (zyklisch, getriggert durch die Anwendung, etc.). Wie bei FERAL wird die funktionale Schnittstelle der einzelnen Komponenten mit Hilfe von Ports festgelegt, welche Signale (Daten, physikalische Größen) bereitstellen bzw. konsumieren. Die Payload eines Rahmens setzt sich aus Signalen zusammen, deren Kodierung innerhalb der FIBEX-Beschreibung ebenfalls festgelegt wird.

Das FIBEX-Format ist herstellerunabhängig, wird von vielen Werkzeugen (Design-, Konfigurations- und Simulationswerkzeuge) im Automobilumfeld unterstützt und gilt als Quasi-Standard zur Beschreibung des Datenaustauschs bei FlexRay. Neben FlexRay unterstützt FIBEX ebenfalls CAN, LIN und MOST und definiert für jede Kommunikationstechnologie ein eigenes XML Schema, welches den Aufbau und die Struktur der FIBEX-Beschreibung für die jeweilige Kommunikationstechnologie, unter Berücksichtigung ihrer spezifischen Eigenschaften, festlegt.

Eine solche Beschreibung spezifiziert ein PKW-Bordnetz vollständig, beginnend bei der Konfiguration des Mediums, den Knoten, ihren Funktionen und Schnittstellen, den verwendeten Rahmen sowie dem Kommunikationsverhalten der Knoten in einer XML-Datei. FERAL hingegen lagert die Konfiguration des Mediums in separate Dateien aus und beschreibt das Kommunikationsverhalten für jeden Knoten individuell in einer separaten DBL-Beschreibung. Beide Ansätze haben Vor- und Nachteile; bei FERAL lässt sich das Kommunikationsverhalten einzelner Komponenten sehr flexibel anpassen, und eine DBL-Beschreibung einer funktionalen Komponente kann in einer anderen Szenarienbeschreibung wiederverwendet werden. Demgegenüber enthält die FIBEX-Beschreibung eine vollständige Spezifikation des Aufbaus des Bordnetzes und der Kommunikationsabläufe in einer zentralen Datei. Im Gegensatz zu FIBEX handelt es sich bei der DBL um eine domänenspezifische Sprache speziell zur Beschreibung des Kommunikationsverhaltens. Dies macht die Spezifikation des Verhaltens und des Rahmenaufbaus deutlich kompakter und einfacher lesbar, im Vergleich zu einer äquivalenten FIBEX-Beschreibung in XML. Eine FIBEX-Beschreibung lässt sich ohne geeignete Werkzeuge kaum manuell erstellen oder ohne Visualisierung lesen. Die Idee bei FIBEX ist zudem die vollständige Spezifikation des Netzwerkes, d.h., die Beschreibung muss alle ausgetauschten Signale und Rahmen aller Komponenten umfassen. Die Erzeugung der Payload innerhalb einer Anwendung und Spezifikation von Extraktionsregeln sind nicht vorgesehen. In Bezug auf die Evaluation von Designalternativen ist die DBL deutlich flexibler als FIBEX, da in einer DBL-Beschreibung das Verhalten für beliebige Kommunikationstechnologien gleichzeitig beschrieben werden kann und automatisch nur die auf die aktuell gewählte Kommunikationstechnologie zutreffenden Regeln ausgewertet werden. Redundanzen werden hierbei soweit wie möglich vermieden. Bei FIBEX hingegen wird für jede Kommunikationstechnologie einer solchen Evaluation jeweils eine vollständig neue FIBEX-Beschreibungen mit vielen redundanten Informationen benötigt, welche dem jeweiligen XML Schema der Kommunikationstechnologie und den dort abgebildeten Konzepten genügt.

In Bezug auf die Ausdrucksstärke sind FIBEX und die DBL (wenn man das erweiterte Konfigurationskonzept der CSCs hinzunimmt) in etwa gleichwertig. Die DBL konzentriert sich stärker auf Aspekte der Austauschbarkeit, während FIBEX eine vollständige Spezifikation eines konkreten Busses (Bordnetzes) anstrebt. Insgesamt ist FIBEX jedoch schon alleine aufgrund seiner Verbreitung im industriellen Umfeld für FERAL interessant, sodass eine zukünftige Unterstützung für FIBEX-Beschreibungen sinnvoll erscheint, um die Interoperabilität von FERAL mit anderen Werkzeugen aus dem industriellen Umfeld und den dortigen Prozessen zu verbessern.

13.3 Existierende Frameworks für Simulatoren sowie deren Kopplung

Abschließend betrachten wir, mit Modelisar, Ptolemy und C-PartSim, ausgewählte Frameworks mit den Schwerpunkten Simulatorkopplung und Simulation und vergleichen diese mit FERAL.

Modelisar

Modelisar [Wol14, BOA+11, MOD10b, MOD10a] ist ein europäisches Projekt (2008-2011), welches mit dem Ziel gestartet wurde, das Design eingebetteter Systeme (im Speziellen die Modellierung fahrzeugbezogener Systeme), durch die Schaffung einheitlicher Schnittstellen als Basis einer offenen, herstellerunabhängigen Simulationsplattform zu verbessern. Das Ergebnis dieser Anstrengung ist die Definition des *Functional Mockup Interface* (FMI). Hierbei handelt es sich um eine standardisierte Schnittstelle, welche es ermöglicht, den von einem Modellierungswerkzeug für ein Modell generierten C-Code automatisiert und ohne Anpassungen in andere Modellierungs- und Simulationsumgebungen zu importieren und dort zu verwenden [Wol14].

Eine ausführbare Komponente, welche das FMI implementiert, wird bei Modelisar als *Functional Mockup Unit* (FMU) bezeichnet. Anhand der von FMI definierten C-Schnittstelle kann eine Simulationsumgebung eine FMU instanziieren und diese simulieren. Hierzu enthält jede FMU eine *FMI Model Description*, welche das von der FMU umgesetzte Modell sowie dessen Datenschnittstelle (in Form von Inputs und Outputs) spezifiziert. Die *FMI Model Description* liegt in XML vor und wird automatisch beim Import einer FMU in eine Simulationsumgebung oder Modellierungswerkzeug ausgewertet.

Modelisar unterstützt mit *FMI for Model Exchange* und *FMI for Co-Simulation* zwei verschiedene Ansätze, um FMUs in eine Modellierungs- oder Simulationsumgebung zu integrieren. *Model Exchange* kommt für Modelle auf Basis differentieller, algebraischer und diskreter Gleichungen im Zustandsraum zum Einsatz; die Simulation der FMU erfolgt unter Verwendung der bereitgestellten Solver des importierenden Werkzeuges. Im Gegensatz hierzu bringt das FMU bei der *Co-Simulation* einen eigenen Solver bzw. eine vollständige Simulationsumgebung mit und kann autark ausgeführt werden. Das *FMI for Co-Simulation* definiert für die *Co-Simulation* eine Schnittstelle zur Simulatorkopplung, wobei der Austausch von Daten zwischen den Simulatoren nur zu festen Zeitpunkten erfolgt (zeitgetriggerte Semantik).

Modelisar/FMI wird bereits von einer Vielzahl von Werkzeugen entweder nativ oder durch Erweiterungen unterstützt, hierzu zählen unter anderem Wolfram System Modeler, LabVIEW, Modelica, MATLAB, AMESim oder IPG CarMaker, um nur eine kleine Auswahl zu nennen. Modelisar unterstützt ausschließlich eine zeitgetriggerte Ausführungssemantik, vergleichbar mit Matlab Simulink. Die Integration ereignisgetriggerter Simulatoren ist, bezogen auf die Co-Simulation, nicht vorgesehen. Somit fehlen Modelisar, im Gegensatz zu FERAL, wichtige zentrale Konzepte für die Kopplung von Simulatoren mit unterschiedlichen MOCCs. Die Integration und Unterstützung von FMI hingegen lässt sich in FERAL über den zeitgetriggerten Direktor sowie eine spezielle Java-Kontrollkomponente realisieren (Co-Simulation). Für die Unterstützung von *FMI for Model Exchange* werden zusätzlich entsprechende nummerische Solver als Bestandteil der Java-Kontrollkomponente benötigt.

Ptolemy

Das Ptolemy Framework [BHLM02, EJL+03, Lee03, Pto14] wurde von der UC Berkeley als Umgebung für die Simulation und das Prototyping heterogener Systeme entwickelt. Die erste Version von Ptolemy wurde bereits 1990 entwickelt und basierte auf C++. Ptolemy II hingegen wurde in Java realisiert und wird seit 1996 aktiv weiterentwickelt. Bereits die ersten Versionen von Ptolemy unterstützten die Verwendung von Simulationskomponenten mit unterschiedlichen Ausführungs- und Kommunikationsmodellen (MOCCs) in einem einzigen zusammenhängenden Simulationssystem.

Hierfür unterteilt Ptolemy das Simulationssystem in semantische Domänen, die jeweils eigene MOCCs bereitstellen und implementieren, wie z. B. ein zeit- oder ereignisgetriggertes Ausführungsmodell. Die verschiedenen Simulationskomponenten (Aktoren) werden – entsprechend ihrer Ausführungssemantik – den jeweiligen Domänen zugeordnet. Jede Domäne wird durch einen Direktor kontrolliert, dieser implementiert das MOCC der Domäne und ist für die Ausführung der Simulationskomponenten (Aktoren) innerhalb der ihm unterstellten Domäne verantwortlich. Eine Besonderheit von Ptolemy ist hierbei die Bereitstellung von Methoden zur Kopplung der unterschiedlichen semantischen Domänen zu einem

gemeinsamen Simulationssystem. Die Aktoren selbst sind Softwarekomponenten und Träger von Funktionalitäten/Verhalten, deren Interaktion durch den Austausch von Nachrichten über Ports und Links erfolgt.

Im Gegensatz zu FERAL verlangt das Ptolemy Framework eine synchrone Ausführung der einzelnen Aktoren und stellt hierüber die Korrektheit der Simulation sicher. Die notwendigen Synchronisationen generieren jedoch einen entsprechend hohen Overhead, welcher sich negativ auf die Performance der Simulation auswirkt. Insbesondere schränken diese auch eine nebenläufige Ausführung der einzelnen Aktoren ein. Um eine bessere Performance erreichen zu können, nutzt FERAL ein modifiziertes Ausführungsmodell, bei dem jede Simulationskomponente ihre eigene lokale Simulationszeit verwaltet. Diese darf, im Gegensatz zu Ptolemy, auch Abweichungen zu den Zeiten der anderen Simulationskomponenten aufweisen. Durch eine regelmäßige Resynchronisation beschränkt FERAL zwar die maximale Abweichung der Simulationszeiten der Komponenten zueinander, dennoch ergeben sich hieraus ggf. Ungenauigkeiten. Je größer die Zeitabstände zwischen den Resynchronisationen gewählt werden, desto höher ist die erreichte Performance durch die Nebenläufigkeit, aber auch die ggf. auftretenden Ungenauigkeiten. Über die sogenannten Aktivitätsperioden kann der Designer den für sein System geeigneten Kompromiss zwischen Performance und Genauigkeit optimal festlegen. Insbesondere ermöglicht es FERAL, innerhalb eines Simulationssystems Domänen mit höherer Genauigkeit zu definieren. So kann der Designer für wichtige Simulationskomponenten eine sehr hohe Genauigkeit wählen, während andere Simulationskomponenten mit geringerer Genauigkeit simuliert werden (z. B. weil deren Modell ungenauer ist oder diese für das Szenario eine untergeordnete Rolle spielen).

Auch wenn FERAL viele Konzepte von Ptolemy adaptiert (Direktoren, Links, Ports, etc.), besteht der wesentliche Unterschied zwischen Ptolemy und FERAL in einer unterschiedlichen Philosophie beider Projekte. Ptolemy versteht sich selbst als Simulationsframework für heterogene Systeme, während FERAL ein Framework für die Simulatorkopplung ist. Während FERAL spezialisierte Simulatoren integriert, um mit deren Hilfe verschiedene Aspekte eines Simulationssystems abzudecken, geht Ptolemy einen anderen Weg. Hier wird die Funktionalität zur Simulation dieses Aspektes neu entwickelt und als Erweiterung direkt in Ptolemy integriert oder als Bestandteil des zu simulierenden Systems mit bereits vorhandenen Funktionen von Ptolemy realisiert.

C-PartsSim

Insgesamt versteht sich FERAL als Neuentwicklung und Verallgemeinerung von *C-PartsSim* [KG08, KB06]. *C-PartsSim* wurde bereits 2008 in der Arbeitsgruppe *Vernetzte Systeme* für die Evaluation eingebetteter Systeme durch die Kopplung spezialisierter Simulatoren entwickelt.

Um eine saubere und zeitgemäße Lösung zu entwickeln, wurde das Design von *C-PartsSim* komplett überarbeitet und im Zuge dessen, aufgrund einer besseren Plattformunabhängigkeit, von C++ zu Java gewechselt. Im Zuge des neuen Designs adoptierte FERAL viele der grundlegenden Prinzipien des Ptolemy Framework (vgl. Kapitel 7.1). Hierzu gehört insbesondere die Unterteilung eines Simulationssystems in semantische Domänen, deren Ausführung durch Direktoren kontrolliert wird. Auch die grundlegenden Schnittstellen für die Steuerung von Aktoren (in FERAL Simulationskomponenten), durch den jeweiligen Direktor, wurden im Wesentlichen übernommen. Überarbeitet wurde auch das Kommunikationsmodell; in FERAL ist der Direktor nicht nur für die Ausführung der Simulationskomponenten verantwortlich, sondern stellt auch sicher, dass die Kommunikation mit anderen Simulationskomponenten außerhalb seiner semantischen Domäne zu keiner Verletzung des MOCCs führen. Dieser Schritt ist unter anderem aufgrund des erweiterten Ausführungsmodells sowie der Unterstützung unterschiedlicher Aktivitätsperioden innerhalb eines Simulationssystems erforderlich. Hierzu werden Links zwischen den Simulationskomponenten durch spezielle Proxyports an den Domänengrenzen unterbrochen, sodass der Direktor auch die Kommunikation zwischen den semantischen Domänen kontrollieren und die kausale Korrektheit sicherstellen kann.

14 Zusammenfassung und Ausblick

Inhaltlich gliedert sich diese Arbeit in zwei Teile, die sich beide mit der effizienten Realisierung und Entwicklung komplexer verteilter (eingebetteter) Echtzeitsysteme in unterschiedlichen Anwendungsgebieten beschäftigen. Im ersten Teil liegt der Fokus auf der Verbesserung der Bandbreitennutzung zeitgetriggerter Kommunikationstechnologien durch den Einsatz von *Mode-Based Scheduling with Fast Mode-Signaling* und adressiert damit primär den Aspekt der Realisierung und Umsetzung der Kommunikation verteilter Echtzeitsysteme. Der zweite Teil beschäftigt sich damit, wie die eigentliche Entwicklung dieser Echtzeitsysteme mit Hilfe von Virtual Prototyping begleitet und Tests sowie Evaluationen des Gesamtverhaltens eines solchen Systems unterstützt werden können. In diesem Kontext stellen wir das Framework FERAL und die im Rahmen dieser Arbeit für FERAL entwickelten (und integrierten) Simulatoren für funktionales Verhalten und Kommunikationstechnologien vor. FERAL und die hier entworfenen Erweiterungen gestatten es, die Entwicklung über die verschiedenen Entwicklungsphasen (von der Designphase bis zur Integration auf der Zielplattform) hinweg mit Hilfe von Simulationen zu begleiten. Auf diese Weise werden kontinuierliche Tests und Evaluationen ermöglicht.

Sowohl in Bezug auf die untersuchten Bussysteme im ersten Teil als auch bei der Entwicklung von Simulatoren haben wir uns auf die Untersuchung von Technologien im Automobilumfeld (z. B. CAN, FlexRay, Simulink) konzentriert. Die Ergebnisse dieser Arbeit sind jedoch nicht auf dieses Anwendungsgebiet beschränkt, sondern lassen sich, wie auch die im Automobilumfeld verwendeten Technologien und Ansätze, in den unterschiedlichsten Bereichen sinnvoll einsetzen. So wird gezeigt, wie sich *Mode-Based Scheduling with Fast Mode-Signaling* im Kontext drahtloser Sensornetze umsetzen lässt, und wie FERAL und Virtual Prototyping bei der Entwicklung eines verteilten Regelungssystems für ein inverses Pendel angewendet werden kann.

Im Folgenden werden noch einmal die wichtigsten Inhalte und Ergebnisse der beiden Teile dieser Arbeit zusammengefasst.

14.1 Mode-Based Scheduling with Fast Mode-Signaling

Der erste Teil der Arbeit konzentriert sich vollständig auf den Kommunikationsaspekt bei der Entwicklung verteilter (eingebetteter) Echtzeitsysteme. In diesem Zusammenhang wird mit *Mode-Based Scheduling with Fast Mode-Signaling* ein neues Konzept für die Gestaltung zuverlässiger und effizienter zeitgetriggerter Kommunikationsprotokolle angewendet, welches weiterhin deterministische Garantien bzgl. des Kommunikationsverhaltens und der auftretenden Verzögerungen gewährt, dabei aber gleichzeitig die Bandbreite effizienter nutzt als Protokolle, die auf dem klassischen zeitgetriggerten Kommunikationsparadigma beruhen. *Mode-Based Scheduling with Fast Mode-Signaling* erreicht dies, indem es pro Zeitslot einen beschränkten kontrollierbaren Wettbewerb unterstützt, anstatt lediglich exklusive Slotreservierungen zu gestatten. Um den Wettbewerb umzusetzen, werden Nachrichtentypen Modes und *Modus-Präferenzen* (Prioritäten) zugeordnet. Die Konkurrenz mehrerer Nachrichten mit unterschiedlichen Modes um einen Zeitslot wird mittels der *Modus-Präferenzen* aufgelöst. Hierbei kommt *Fast Mode-Signaling* zum Einsatz – eine effiziente und zuverlässige Technik –, um innerhalb eines Netzwerkes einen Konsens bezüglich des Modes mit der höchsten *Modus-Präferenz* zu finden und zu propagieren. Dieser Ansatz gewährt deterministische Garantien bezüglich der auftretenden Verzögerungen für alle Nachrichtentypen mit der jeweils höchsten *Modus-Präferenz* innerhalb eines Slots. Zugleich ist es nicht wie bei klassischen zeitgetriggerten Protokollen notwendig, für sporadische Nachrichten, die deterministische Garantien benötigen, jeweils

einen Slot exklusiv zu reservieren, der bei seltenen sporadischen Nachrichten häufig ungenutzt bliebe. Bei *Mode-Based Scheduling with Fast Mode-Signaling* kann dieser Slot stattdessen für die Übertragung von Nachrichten niedrigerer Priorität genutzt werden.

Nach einer kurzen Einführung in *Mode-Based Scheduling with Fast Mode-Signaling* widmet sich Kapitel 2 zunächst der Frage nach den Einsatzmöglichkeiten von *Mode-Based Scheduling*. Hierzu wurden typische Anwendungsszenarien von *Mode-Based Scheduling* in Form von Anwendungsmustern generalisiert beschrieben. Diese Muster helfen Designern, *Mode-Based Scheduling* sinnvoll einzusetzen, und zeigen dabei gleichzeitig Optionen auf, um hierdurch die verfügbare Bandbreite besser zu nutzen und gleichzeitig die Qualität der Kommunikation durch die Unterstützung von QoS zu verbessern.

Der Schwerpunkt des ersten Teils liegt auf der Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in bereits etablierte Kommunikationstechnologien. In diesem Zusammenhang liegt die Herausforderung in der Entwicklung effizienter und zuverlässiger (deterministischer) Realisierungsmöglichkeiten von *Fast Mode-Signaling* im Rahmen der jeweiligen Technologie. Im Zuge dieser Arbeit wurden sowohl Techniken zur Realisierung von aktivem als auch passivem *Fast Mode-Signaling* entwickelt und bei der Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in TTCAN, FlexRay sowie drahtlose Netzwerke umgesetzt.

Aufgrund der konzeptionell einfachen Umsetzung erfolgte in Kapitel 3 zunächst die Integration von *Mode-Based Scheduling with Fast Mode-Signaling* in das TTCAN-Protokoll. Hierzu wurden im Rahmen dieser Arbeit die Ansätze aus [KG13] zunächst detailliert ausgestaltet und eine praxistaugliche robuste und deterministische Realisierung für *Fast Mode-Signaling* entwickelt. In diesem Zuge erfolgte die Erweiterung von TTCAN um einen neuen Typ von *Time Windows*, innerhalb derer – entsprechend der Idee von *Mode-Based Scheduling* – die Anzahl der konkurrierenden Nachrichten beschränkt wird. *Mode-Based Scheduling* fordert, dass bei mehreren konkurrierenden Nachrichten diejenige mit der höchsten *Modus-Präferenz* gewinnt. Die konzeptionelle Umsetzung des *Fast Mode-Signaling* ist einfach, da die CAN-Arbitrierung für die Realisierung genutzt werden kann (aktives *Fast Mode-Signaling*). Hierfür werden die *Modus-Präferenzen* in verträglicher Weise auf CAN-Identifier abgebildet.

Damit der Ausgang der CAN-Arbitrierung und somit des *Fast Mode-Signaling* deterministisch ist, müssen die um einen Slot konkurrierenden Knoten ihre Nachrichtenübertragung (nahezu) zeitgleich starten, sodass sich die SOF-Bits überlagern. Dieser Aspekt bereitet in der Praxis Probleme, da die Uhren der einzelnen Knoten, trotz regelmäßiger (Re-)Synchronisation, nicht perfekt synchron sind. Daher beschäftigt sich das Kapitel 3, neben der konzeptionellen Umsetzung, insbesondere mit den Möglichkeiten einer zuverlässigen und deterministischen Realisierung des *Fast Mode-Signaling* mittels der CAN-Arbitrierung. Hierzu wurden mittels theoretischer Betrachtungen die maximal zulässige Synchronisationsungenauigkeit ermittelt, bei der eine korrekte Funktionsweise noch sichergestellt ist. Die so bestimmte maximal zulässige Abweichung gestattet Rückschlüsse bzgl. zulässiger Konfigurationsparameter des Basiszyklus, welche in Form von Constraints formalisiert wurden. Der in diesem Rahmen realisierte Prototyp auf Basis handelsüblicher CAN-Controller weist nach, dass die entwickelten Konzepte korrekt arbeiten, zeigt aber auch, dass sich *Fast Mode-Signaling* auf dieser technischen Grundlage bei höheren Bandbreiten nicht zuverlässig implementieren lässt.

Um dieses Problem zu lösen, wurde in dieser Arbeit mit *Bring-your-own-Tick (BOT)* eine Erweiterung für (TT)CAN-Controller entwickelt, die es einem CAN-Controller ermöglicht, sich in eine laufende Übertragung *einzuklinken*. Hierdurch lassen sich beliebig große Synchronisationsungenauigkeiten ausgleichen, wodurch die korrekte Funktionalität von *Fast Mode-Signaling* sichergestellt wird. Durch BOT lässt sich *Fast Mode-Signaling* auf Basis der CAN-Arbitrierung zuverlässig, robust und deterministisch implementieren. Zum Nachweis der Funktionalität wurde mittels VHDL ein TTCAN-Controller mit BOT-Unterstützung spezifiziert und dieser mittels (taktgenauer) Simulationen evaluiert. Erst die Entwicklung und Verwendung der BOT-Erweiterung gestattet den zuverlässigen Einsatz von *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* im Kontext sicherheitskritischer Anwendungen.

Da die für CAN vorgestellte Lösung von *Fast Mode-Signaling* spezielle Anforderungen an die verwendete Basistechnologie (Bit-Codierung, Arbitrierung etc.) stellt, beschreibt Kapitel 4 einen alternativen Ansatz für die Realisierung von *Fast Mode-Signaling* mit Backoffslots (passives *Fast Mode-Signaling*), der im Rahmen dieser Arbeit entwickelt wurde. Diese Variante des *Fast Mode-Signaling* stellt kaum Anforderungen an die eingesetzte Basistechnologie und eignet sich somit optimal für die Integration in bestehende Kommunikationsprotokolle. Kapitel 4 entwickelt die grundlegenden Konzepte für die Realisierung von *Fast Mode-Signaling* mittels Backoffslots und definiert die notwendige Struktur für den Aufbau der Mikroslots anhand eines abstrakten Kommunikationsmodells. Die nachfolgenden Kapitel dieses Teils nutzen dieses Modell, um eine Umsetzung von *Mode-Based Scheduling with Fast Mode-Signaling* innerhalb drahtloser Netzwerke zu studieren. Hierfür wurde der bereits existierende *Black burst-Integrated Protocol Stack* (BiPS) für die *Imote 2*-Plattform um ein modusbasiertes Kommunikationsprotokoll erweitert und dieses anschließend erfolgreich evaluiert.

Kapitel 6 präsentiert mehrere (von uns patentierte [6]) Konzepte, um *Mode-Based Scheduling with Fast Mode-Signaling* in FlexRay zu integrieren. Auch hier bilden die Backoffslots die Grundlage für die Implementierung des *Fast Mode-Signaling*. In dieser Arbeit wird im Detail sowohl die Erweiterung des dynamischen Segments als auch eine Anpassung des statischen Segments vorgestellt. Während erstere keinerlei Anpassung des FlexRay-Standards erfordert und mit diesem kompatibel ist, ist eine Realisierung im statischen Segment nicht ohne Modifikationen des FlexRay-Standards möglich, dafür ist die Lösung innerhalb des statischen Segments jedoch deutlich flexibler.

14.2 FERAL

Während sich der erste Teil dieser Arbeit auf die Steigerung der Effizienz der Kommunikation konzentriert, beschäftigt sich der zweite Teil damit, wie die Entwicklung komplexer verteilter Echtzeitsysteme mit Hilfe von Virtual Prototyping und Simulationen unterstützt werden kann. In diesem Zusammenhang wird das Framework FERAL vorgestellt, welches es erlaubt, die Entwicklung eines verteilten Echtzeitsystems über die verschiedenen Entwicklungsphasen hinweg mit Hilfe von Simulationen zu begleiten. Im Rahmen dieser Arbeit wurde FERAL um Simulatoren und Konzepte erweitert, die sowohl die Simulation aktueller Kommunikationstechnologien (wie CAN oder FlexRay) im Kontext verteilter Echtzeitsysteme als auch die Evaluation von Designalternativen hinsichtlich einer geeigneten Kommunikationstechnologie gestattet.

FERAL wurde im Rahmen einer Kooperation mit dem Fraunhofer IESE entwickelt und ist selbst kein Simulator, sondern ein Framework für die Kopplung von Simulatoren. Hinter diesem Ansatz steckt die Überzeugung, dass es aufgrund der unterschiedlichen heterogenen Anforderungen bei der Simulation komplexer verteilter Echtzeitsysteme besser ist, spezialisierte Simulatoren für die Simulation verschiedener Aspekte heranzuziehen, anstatt einen Simulator zu entwickeln, der versucht, alle Aspekte eigenständig abzudecken. Deshalb konzentriert sich FERAL darauf, geeignete Mechanismen, Konzepte, Schnittstellen und Lösungen für die Kopplung von Simulatoren bereitzustellen, und ermöglicht die beliebige Zusammenstellung der einmal integrierten Simulatoren zu Simulationssystemen. Auf diese Weise lassen sich flexibel alle für das Verhalten relevanten Aspekte eines Anwendungsszenarios in der Simulation berücksichtigen.

Die Entwicklung der Kernkomponenten von FERAL erfolgte durch das Fraunhofer IESE. Ein Schwerpunkt dieser Arbeit liegt bei der Integration und Entwicklung spezieller Simulatoren für FERAL, welche die Simulation von funktionalem Verhalten (FSCs) sowie Kommunikationssystemen (CSCs) ermöglichen. Hierzu haben wir in Kapitel 8 zunächst ein Adaptionsschema entwickelt, welches die verschiedenen Schritte bei der Integration existierender Simulatoren in FERAL definiert. Dieses Schema haben wir dann angewendet, um Simulatoren für Matlab Simulink-Modelle und SDL-Spezifikationen in FERAL zu integrieren.

Aufgrund der in FERAL verfügbaren Simulatoren für Kommunikationssysteme können nicht nur einzelne Komponenten eines verteilten Systems (sowie dessen Umgebung), sondern das komplette verteilte Echtzeitsystem inkl. aller Komponenten sowie deren verteilte Interaktion (über diese Kommunikationssysteme) simuliert werden. Hierzu wurden in dieser Arbeit spezielle Simulatoren für CAN, FlexRay sowie ein abstraktes Kommunikationsmodell für FERAL entwickelt und integriert (Kapitel 9). Durch den hohen Detailgrad dieser CSCs werden das exakte Protokollverhalten sowie dessen Eigenschaften genau nachgebildet, sodass die Auswirkungen nicht-funktionaler Eigenschaften der Kommunikation (wie z. B. Verzögerungen oder Nachrichtenverluste) auf das Verhalten des Gesamtsystems untersucht werden können. Die ebenfalls entwickelten Fehlermodelle ermöglichen die Simulation des Verhaltens bei auftretenden Kommunikationsfehlern.

Die im Kontext dieser Arbeit entwickelten CSCs verwenden eine einheitliche Schnittstelle und Architektur, um deren Austauschbarkeit innerhalb eines Simulationssystems zu vereinfachen. So lassen sich mit FERAL Designalternativen hinsichtlich einer geeigneten Kommunikationstechnologie mit Hilfe von Simulationen evaluieren und objektiv anhand der Ergebnisse vergleichen. Um den Austausch von CSCs innerhalb eines Simulationssystems zu erleichtern, wurden mit Bridges und Gateways Konzepte entwickelt (Kapitel 10), um das Kommunikationsverhalten von den FSCs zu isolieren bzw. verschiedene CSCs zu koppeln. Die Bridges übernehmen für die FSCs die Aufgabe einer Kommunikationsmiddleware. So wird erreicht, dass eine CSC ausgetauscht werden kann, ohne hierfür die FSC oder ihr Verhaltensmodell anpassen zu müssen. Das Kommunikationsverhalten der Bridges bzw. das Weiterleitungsverhalten der Gateways wird mit der domänenspezifischen Sprache DBL spezifiziert. Das Vorgehen bei der Evaluation unterschiedlicher Kommunikationstechnologien sowie die Kombination der verschiedenen in FERAL integrierten Simulatoren zur Realisierung eines komplexen Anwendungsszenarios wurde anhand eines verteilten *Adaptive Cruise Control* (ACC) Systems (Kapitel 11) demonstriert.

Kapitel 12 stellt eine übergreifende Methodik vor, um FERAL und Virtual Prototyping begleitend während allen Entwicklungsphasen eines verteilten Echtzeitsystems einzusetzen, beginnend bei den frühen Designphasen bis hin zur Integration. Hierbei wird zuerst ein abstraktes Simulationssystem entworfen und dann durch die Wahl konkreter Simulatoren und Verhaltensmodelle instanziiert. Das hieraus resultierende konkrete Simulationssystem dient als Basis für die Simulationen. Durch FERALs einheitliche Schnittstelle können in einem Simulationssystem Komponenten auf unterschiedlichen Abstraktionsstufen miteinander interagieren. Dies erlaubt es, während der Entwicklung die konkreten Simulationssysteme schrittweise durch die Weiterentwicklung der einzelnen Komponenten zu verfeinern. Jedes aus einer solchen Modifikation resultierende konkrete Simulationssystem kann wieder mit FERAL simuliert werden. Somit steht während jeder Entwicklungsphase ein ausführbares Simulationssystem für Tests und Evaluationen zur Verfügung, sodass sich Fehler früh identifizieren und beheben lassen. Auch der letzte Schritt der Entwicklung, die Integration der einzelnen funktionalen Komponenten auf ihrer Zielplattform, wird durch unsere prototypische Unterstützung für HiL-Simulationen durch FERAL unterstützt. Hierdurch lassen sich die Einflüsse plattformspezifischer Verzögerungen (z. B. durch Laufzeiten) auf das Gesamtverhalten ermitteln, ohne dass es zu einer Gefährdung oder Beschädigung der realen physikalischen Umgebung kommt, da diese weiterhin simuliert wird. Dieses Vorgehen wird anhand der Entwicklung eines verteilten Regelungssystems für ein inverses Pendel demonstriert.

14.3 Ausblick

In Bezug auf *Mode-Based Scheduling with Fast Mode-Signaling* wurde bislang noch nicht untersucht, inwiefern sich optimale Ablaufpläne anhand der Kommunikationsanforderungen und Charakteristika von Nachrichten automatisch ableiten lassen. Mögliche Forschungsansätze bieten die bereits existierenden Verfahren zur Bestimmung von Ablaufplänen für zeitgetriggerte Kommunikationsprotokolle (vgl. Kapitel 2.5) oder die Untersuchung heuristischer Verfahren, wie z. B. genetischer Algorithmen. In diesem Kontext

stellt sich auch die Frage nach weiteren geeigneten Optimierungskriterien, die über die alleinige Erfüllung der Kommunikationsanforderungen hinausgehen und wie diese sich modellieren, quantifizieren und in die Ablaufplanung einbringen lassen. Ein Beispiel hierfür ist die Frage nach der Erweiterbarkeit eines Ablaufplans. Wie die Anwendungsmuster nahelegen, könnte man neben verpflichtend zu erfüllenden Kommunikationsanforderungen (z. B. minimale garantierte Bandbreite oder maximale Verzögerungen) auch zusätzliche Anforderungen definieren, die zwar gewünscht sind, jedoch nicht garantiert werden müssen oder nur auf der Basis eines stochastischen Nachrichtenmodells zusicherbar sind. Ein Beispiel hierfür wäre die Angabe einer gewünschten erwarteten Bandbreite für bestimmte Nachrichtentypen (ggf. zusätzlich zu bereits existierenden verpflichtenden Kommunikationsanforderungen). Eine zusätzliche Schwierigkeit besteht in einer geeigneten Gewichtung komplementärer Optimierungsziele (z. B. Erweiterbarkeit gegenüber kurzen Reaktionszeiten), hier sind gegebenenfalls zusätzliche Informationen oder Entscheidungen durch den Designer notwendig.

Ein weiterer wichtiger Aspekt bei *Mode-Based Scheduling with Fast Mode-Signaling* betrifft die Durchführung von Fallstudien, um zu ermitteln, welche Vorteile der Einsatz von *Mode-Based Scheduling with Fast Mode-Signaling* in realen Anwendungsszenarien bezogen auf die Bandbreitennutzung bringt. Diese Studien lassen sich dann auch für die Erprobung von Verfahren zur automatischen Ablaufplanung verwenden bzw. können zunächst als Informationsquelle dienen, um zu ermitteln, welche Arten von Optimierungskriterien relevant sind und wie diese geeignet abgebildet werden können.

Eine ebenfalls interessante Aufgabenstellung betrifft die Realisierung eines FlexRay-Kommunikationscontrollers (CC) mit Unterstützung für die verschiedenen Umsetzungsmöglichkeiten von *Mode-Based Scheduling with Fast Mode-Signaling*. Das Verhalten eines solchen CCs könnte, wie bei CAN, mit Hilfe von VHDL spezifiziert und anschließend simuliert werden. Der nächste Schritt bestünde dann in einer Realisierung als Hardware-Prototyp und dessen experimenteller Evaluation.

Auch hinsichtlich unseres mit VHDL spezifizierten TTCAN-Controllers steht noch der Aufbau eines Prototypen in Hardware sowie dessen experimentelle Evaluation aus. Hierbei geht es sowohl um die Erprobung der BOT-Funktionalität in der Praxis als auch um die korrekte Interaktion mit Standard-Controllern. In diesem Zuge müssten dann auch noch die Fehlersignalisierung sowie eine Schnittstelle für externe Mikrocontroller bzw. innerhalb des FPGAs eingebetteter Mikrocontroller realisiert werden, um diese Tests sinnvoll durchführen zu können. Auch die vollständige Realisierung der Funktionalitäten der Nachrichtenpuffer (innerhalb des TTCAN-Controllers) mit Unterstützung für *Mode-Based Scheduling* kann in diesem Rahmen erfolgen. Unter Umständen liefern auch hier konkrete Anwendungsszenarien noch Ansätze für Optimierungen und neue Funktionen.

Um solche auf VHDL basierende Prototypen in Zukunft auch im Kontext größerer Szenarien testen zu können (wie beispielsweise auch die BOT-Erweiterung für CAN), wäre zudem die Integration eines VHDL-Simulators in FERAL sinnvoll. Ein möglicher Kandidat hierfür ist der freie GHDL-Simulator, welcher bereits für die Evaluation des TTCAN-Controllers eingesetzt wurde. Auf diese Weise ließen sich Erweiterungen von Kommunikationscontrollern und deren Auswirkungen (wie etwa Verbesserungen der Effizienz) direkt innerhalb eines simulierten verteilten Echtzeitsystems erproben und evaluieren. Die Integration eines VHDL-Simulators erschließt darüber hinaus auch weitere hardwarenahe Evaluationsmöglichkeiten für FERAL und stellt deshalb, unabhängig von dem konkreten Anwendungsszenario, eine sinnvolle Ergänzung dar.

Das Konzept der Bridges als Middleware zwischen FSCs und CSCs zur Entkopplung der Verhaltensmodelle von deren Kommunikationsverhalten existiert bislang nur für FERAL. Hier stellt sich die Frage, ob und inwiefern sich die Konzepte zur Beschreibung des Kommunikationsverhaltens mit unserer domänenspezifischen Sprache DBL auch in einer realen Middleware sinnvoll einsetzen lassen. Anwendungsszenarien sind innerhalb von BiPS denkbar, etwa um einem Anwendungsentwickler die Nutzung der MAC-Protokolle zu erleichtern, indem dieser das Kommunikationsverhalten seiner Anwendung durch Regeln definiert und sich nicht mit der Realisierung sowie den Schnittstellen von BiPS auseinandersetzen muss.

Neben dieser möglichen weiteren Nutzung der Bridges außerhalb von FERAL sind auch Erweiterungen der bereits für FERAL entwickelten CSCs denkbar. Der CAN-CSC fehlt in der aktuellen Version beispielsweise noch die Unterstützung für TTCAN, *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* sowie CAN FD. Gerade die Ergänzung der CAN-CSC um *Mode-Based Scheduling with Fast Mode-Signaling für TTCAN* würde auch bei Fallstudien in Bezug auf den Einsatz von *Mode-Based Scheduling with Fast Mode-Signaling* helfen, da hierüber auch das Gesamtverhalten eines Systems und die Auswirkung bei Verwendung einer modusbasierten Kommunikation simuliert werden könnte. Gleichzeitig würden auf die Weise durchgeführte Fallstudien die Tauglichkeit und Anwendbarkeit von FERAL für praktische Szenarien weiter unterstreichen.

Anhang

A Mode-Based Scheduling with Fast Mode-Signaling for TTCAN

A.1 Signalverzögerungen bei CAN

Im Rahmen der Definition 3.3 (Kapitel 3.1.1) wurden die einzelnen Größen identifiziert, welche für die Signalverzögerung bei der Übertragung einer Flanke zwischen zwei Knoten verantwortlich sind. Die Tabelle A.1 verdeutlicht die Größenordnungen dieser Verzögerungen auf Basis von Herstellerangaben und Datenblättern [Tex08b, Tex02, Nat96] für beliebige Knoten $x \in V$, $y \in V$. Das Propagation-Delay wurde für einen CAN-Bus mit einer maximalen Ausdehnung von 40 Metern, unter Berücksichtigung einer Ausbreitungsgeschwindigkeit von $5\frac{ns}{m}$ [Ets01], berechnet. Die verwendeten 40 Meter entsprechen der maximal zulässigen Ausdehnung bei einer Übertragungsrate von 1 MBit/s.

Kenngröße	Verzögerung
$d^{x}_{rxProcDelay,\nearrow}$, $d^{x}_{rxProcDelay,\searrow}$	50-75 ns
$d^{x}_{txProcDelay,\nearrow}$, $d^{x}_{txProcDelay,\searrow}$	50-75 ns
$d^{x}_{txTransDelay,\nearrow}$	50 ns
$d^{x}_{txTransDelay,\searrow}$	40-80 ns
$d^{x}_{rxTransDelay,\nearrow}$, $d^{x}_{rxTransDelay,\searrow}$	80 ns
$d^{x,y}_{PropagationDelay,\nearrow}$, $d^{x,y}_{PropagationDelay,\searrow}$	200 ns bei 40 Metern Signalweg

Tabelle A.1: CAN-Bus Signalverzögerungen

A.2 Voraussetzungen für die Umsetzung von Mode-Based Scheduling with Fast Mode-Signaling für TTCAN

Der maximale Tickoffset $d_{maxOffset}$ zwischen den Knoten, am Ende eines Basiszyklus, ist abhängig von dessen Länge (Resynchronisationsintervall) sowie dem Clock Skew der lokalen Uhren der Knoten zueinander.

Definition A.1 (Maximaler Tickoffset $d_{maxOffset}$)

Sei d_{Cycle} die Länge des Basiszyklus und V die Menge der Knoten des TTCAN-Busses. Des Weiteren gebe $d_{maxBaseOffset}$ die maximale paarweise Ungenauigkeit der Knoten nach der Synchronisation durch den Empfang der Referencemessage an (BaseOffset). Darüber hinaus sei $r_{maxClockSkew}$ der maximale Clock Skew[1] der Knoten aus V. Dann ist der maximale Tickoffset am Ende des Basiszyklus

$$d_{maxOffset} \equiv d_{maxBaseOffset} + 2 \cdot r_{maxClockSkew} \cdot d_{Cycle}.$$

[1] Der maximale Clock Skew $r_{maxClockSkew}$ wird relativ in *ppm – parts per million* – angegeben.

Für die Durchführung der Synchronisation nutzen die Knoten den Empfangszeitpunkt des SOF-Bits der Referencemessage als Referenzzeitpunkt. Dieser Empfangszeitpunkt variiert bei den verschiedenen Knoten in Abhängigkeit von der Signalverzögerung zwischen Empfänger und Time Master. Dementsprechend definieren wir den maximalen Baseoffset $d_{maxBaseOffset}$ (unmittelbar) nach der Synchronisation als die maximale Abweichung der Empfangszeitpunkte des SOF-Bits der Referencemessage auf Basis der maximalen Signalverzögerung (Definition 3.3) zwischen den einzelnen Knoten.

Definition A.2 (Maximaler Baseoffset $d_{maxBaseOffset}$ nach der Synchronisation)

Sei V die Menge der Knoten des TTCAN-Busses und $v_T \in V$ der aktive Time Master des TTCAN-Busses. Dann ist der maximale Baseoffset $d_{maxBaseOffset}$ die maximale paarweise Ungenauigkeit der Knoten V unmittelbar nach dem Empfang der Referencemessage.
Der maximale Baseoffset $d_{maxBaseOffset}$ kann abgeschätzt werden durch,

$$\begin{aligned} d_{maxBaseOffset} &\leq \max_{v_1 \in V, v_2 \in V} |delay_{max}(v_T, v_1) - delay_{min}(v_T, v_2) + d_L^{v_1,v_2}| \\ &\leq delay_{max}^V - delay_{min}^V + \max_{v_1 \in V, v_2 \in V} d_L^{v_1,v_2} \end{aligned}$$

mit

$$d_L^{x,y} = \max(d_L^x, d_L^y)$$

$$d_L^x = \begin{cases} d_q^x, & \text{für TTCAN-Level-1} \\ d_{TimeStampPrec}^x & \text{für TTCAN-Level-2} \end{cases}$$

wobei d_q^v die konfigurierte Länge des Zeitquantums des Knotens $v \in V$ bezeichnet. Ist $v \in V$ ein Knoten mit TTCAN-Level-2 Unterstützung, so bezeichnet $d_{TimeStampPrec}^v$ die maximale Ungenauigkeit bei der Ermittlung des Zeitstempels der fallenden Flanke des SOF-Bits der Referencemessage durch den TTCAN-Controller (in Hardware).

Wie bei dem maximalen Tickoffset der modusbasierten Knoten $d_{maxBaseOffset,MB}$ (Definition 3.8) müssen wir auch in Definition A.2 zwischen TTCAN-Level 1 und TTCAN-Level 2. Für entsprechende Erläuterung siehe Kapitel 3.2.2.

Definition A.2 geht davon aus, dass der Time Master mit synchronisiert wird. Hierbei nehmen wir an, dass dieser als Referenzzeitpunkt für die Synchronisation den Empfangszeitpunkt der fallenden Flanke, des von ihm selbst gesendeten SOF-Bits, nutzt. In diesem Fall wird das *loop-delay* korrekt über den Ausdruck $delay(v_T, v_T)$ berücksichtigt. Siehe auch hierzu Kapitel 3.2.2.

Darüber hinaus liefert Definition A.2 auch eine für die Praxis nutzbare Abschätzung von $d_{maxBaseOffset}$. Der dargestellte Ausdruck kann durch Annahme $delay_{min}^V = 0$ weiter vereinfacht werden, hierdurch wird die Abschätzung jedoch ungenauer. Analog kann auch $delay_{max}^V$, wie in Abschnitt 3.1.1 beschrieben, anhand von (technischen) Maximalwerten abgeschätzt werden, um eine Unabhängigkeit von der Topologie zu erreichen.

B Realisierung von Fast Mode-Signaling mittels Backoffslots

B.1 Beweis der Verträglichkeit von slt_{mp} mit der Modus-Präferenz-Funktion mp

Der Nachweis, dass die optimale Abbildung slt_{mp} von Modes auf Backoffslots mit der *Modus-Präferenz*-Funktion verträglich ist, folgt unmittelbar aus der Definition von slt_{mp} sowie der Definition für verträgliche Abbildungen (Definition 4.3). Zur besseren Übersicht sei die entsprechende Definition 4.5 an dieser Stelle noch einmal kurz dargestellt, bevor wir uns dem eigentlichen Beweis widmen.

Definition 4.5: Sei V die Menge der Knoten eines Netzwerkes, S die Menge der Mikroslots und M die Menge der Modes. Des Weiteren sei $mp : S \times M \to \mathbb{N}_0$ eine *Modus-Präferenz*-Funktion und $SA : S \times M \to V$ die Slot-Assignment-Funktion.

Dann ordnet die partielle Funktion $slt_{mp} : S \times M \to \mathbb{N}_0$ jedem Mode $m \in M$ eines modusbasierten Mikroslots $s \in S$ einen Backoffslot zu, in dem die Übertragung eines Rahmens dieses Modes gestartet werden darf. Die partielle Funktion $slt_{mp} : S \times M \to \mathbb{N}_0$ ist wie folgt definiert:

$$\forall s \in S, \forall m \in M \textit{ mit } SA(s,m) \textit{ definiert} : slt_{mp}(s,m) \equiv |M_s \backslash M_{s,m}|$$

mit

$$M_s \equiv \{m \in M | SA(s,m) \textit{ ist definiert}\}$$
$$M_{s,m} \equiv \{m' \in M_s | mp(s,m) \leq mp(s,m')\}$$

Wir führen den Nachweis, dass slt_{mp} verträglich mit der *Modus-Präferenz*-Funktion mp.

Lemma B.1

Sei V die Menge der Knoten eines Single-Hop-Netzwerkes, S die Menge der Mikroslots und M die Menge der Modes. Des Weiteren sei $mp : S \times M \to \mathbb{N}_0$ eine Modus-Präferenz-Funktion und $SA : S \times M \to V$ die Slot-Assignment-Funktion.
Dann ist die partielle Funktion $slt_{mp} : S \times M \to \mathbb{N}_0$, gemäß Definition 4.5 verträglich mit der Modus-Präferenz-Funktion mp.

Beweis. Wir zeigen die Behauptung anhand eines Widerspruchsbeweises. Angenommen die partielle Funktion $slt_{mp} : S \times M \to \mathbb{N}_0$ ist nicht verträglich mit der *Modus-Präferenz*-Funktion mp, dann $\exists s \in S, \exists m_1 \in M, \exists m_2 \in M$ mit $m_1 \neq m_2$ und $SA(s,m_1)$ definiert, $SA(s,m_2)$ definiert, sodass $mp(s,m_1) < mp(s,m_2)$ jedoch $slt_{mp}(s,m_1) \geq slt_{mp}(s,m_2)$ ist.
Aus der Definition folgt,

$$slt_{mp}(s,m_1) = |M_s \backslash M_{s,m_1}|$$
$$slt_{mp}(s,m_1) = |M_s \backslash M_{s,m_2}|$$

Die Menge M_s enthält alle Modes, für die es eine Slotzuordnung für den Slot $s \in S$ gibt. Ordnen wir die Menge M_s anhand der zugeordneten *Modus-Präferenz* aufsteigend, ergibt sich die folgende Darstellung, da $mp(s,m_1) < mp(s,m_2)$.

$$M_s = \{m_{s_1}, \ldots, m_1, \ldots, m_2, \ldots, m_{s_n}\}$$

Dann gilt natürlich auch $M_{s,m_1} \subseteq M_s$ und $M_{s,m_2} \subseteq M_s$. Stellen wir die entsprechenden Mengen wieder aufsteigend geordnet anhand der zugeordneten *Modus-Präferenz* dar und berücksichtigen dabei, dass $mp(s,m_1) < mp(s,m_2)$ gilt, so ergibt sich

$$M_{s,m_1} = \{m_1, \ldots, m_2, \ldots, m_{s_n}\}$$
$$M_{s,m_2} = \{m_2, \ldots, m_{s_n}\}$$

Dann folgt daraus unmittelbar, dass

$$|M_{s,m_1}| > |M_{s,m_2}|$$
$$\Rightarrow |M_s \backslash M_{s,m_1}| < |M_s \backslash M_{s,m_2}|$$

und basierend auf der Definition der partiellen Funktion slt_{mp}

$$\Rightarrow slt_{mp}(s,m_1) < slt_{mp}(s,m_2)$$

Hierbei handelt es sich um einen Widerspruch zur Voraussetzung und somit ist die partielle Funktion slt_{mp} gemäß Definition 4.5 *verträglich* mit der *Modus-Präferenz*-Funktion mp. □

C Mode-Based Scheduling with Fast Mode-Signaling für BiPS

C.1 Signalverzögerungen bei der Ausführung von BiPS auf der Imote 2-Plattform

Die maximale Übertragungsverzögerung $sig\_delay_{max}^{V_{MB} \to V}$ ist, wie folgt definiert (Definition 4.7),

$$\begin{aligned} sig\_delay_{max}^{V_{MB} \to V} &= \max_{v_1 \in V_{MB}, v_2 \in V, v_1 \neq v_2} sig\_delay_{max}(v_1, v_2) \\ &= \max_{v_1 \in V_{MB}, v_2 \in V, v_1 \neq v_2} (d_{txProcDelay}^{v_1} + d_{PropagationDelay}^{v_1, v_2} + d_{rxProcDelay}^{v_2}) \end{aligned} \tag{C.1}$$

Wir bestimmen nun die einzelnen Kenngrößen des Ausdrucks (C.1) für unsere Implementierung von BiPS auf der *Imote 2*-Plattform.

$d_{txProcDelay}^{v}$**:** Gemäß unseren Vorüberlegungen befinden sich die zum BTSP zu übertragenden Daten bereits im TXFIFO. Um die eigentliche Übertragung am BTSP zu starten, wird erneut ein Hardware-Timer aufgezogen, der zu diesem Zeitpunkt abläuft und mit dem Befehl `STXON` den Sendevorgang startet. Dementsprechend setzt sich $d_{txProcDelay}^{v}$ aus der Verzögerung für die Übermittlung des `STXON`-Befehls und der Verzögerung für die Bearbeitung des Timer-Interrupts zusammen, welcher die Übertragung des `STXON`-Befehls (1 Byte über SPI) initiiert.

$$d_{STXON} \approx 5\,\mu s + 0,7\,\mu s = 5,7\,\mu s \tag{C.2}$$

$$d_{txProcDelay}^{v_1} = d_{INT} + d_{STXON} \approx 20,7\,\mu s \tag{C.3}$$

$d_{PropagationDelay}^{v_1, v_2}$**:** Das Propagation-Delay bestimmen wir für eine maximale Reichweite von 200 Metern und erhalten entsprechend

$$d_{PropagationDelay}^{v_1, v_2} = d_{maxPropagationDelay} \leq 0,7\,\mu s \tag{C.4}$$

$d_{rxProcDelay}^{v_2}$**:** Da innerhalb des Backoffslots nur der Beginn einer Übertragung detektiert werden muss genügt es, dies über den CCA-Mechanismus abzubilden. Dementsprechend muss der Backoffslot ausreichend lang sein, um auf eine steigende Flanke des CCA-Pins des PXA271 zu reagieren. Dieser löst einen Interrupt aus, dessen Verzögerung wir wieder mit der oberen Schranke von 15 µs abschätzen.

$$d_{rxProcDelay}^{v_2} = d_{CCA-Interrupt} = d_{INT} = 15\,\mu s \tag{C.5}$$

Insgesamt erhalten wir[1] für die Verzögerung $sig\_delay_{max}^{V_{MB} \to V}$ durch Einsetzen:

$$\begin{aligned} sig\_delay_{max}^{V_{MB} \to V} &= \max_{v_1 \in V_{MB}, v_2 \in V, v_1 \neq v_2} (d_{txProcDelay}^{v_1} + d_{PropagationDelay}^{v_1, v_2} + d_{rxProcDelay}^{v_2}) \\ &\leq 36,4\,\mu s \end{aligned}$$

[1] Unter der Annahme, dass alle Knoten identisch sind.

C.2 Relevante Verzögerungen für die Bestimmung der minimalen Länge der Mikroslots auf der Imote 2-Plattform

Die Teilausdrücke in Ausdruck (5.10) aus Kapitel 5.2.2.2 ergeben sich sowohl aus den Eigenschaften der Implementierung von BiPS als auch den spezifischen Verzögerungen der Hardwareplattform.

$d_{maxTransmission}$**:** Für die Ermittlung der maximalen Übertragungsdauer, gehen wir von einem Rahmen mit der maximal zulässigen Nutzlast von 127 Bytes aus (IEEE 802.15.4). Zusätzlich müssen noch ein Synchronisationsheader (5 Byte) sowie ein Header auf physikalischer Schicht (1 Byte) berücksichtigt werden [Tex07]. Insgesamt ergibt sich bei einer Übertragungsrate von 250 kBit/s eine Übertragungsverzögerung von:

$$d_{maxTransmission} \leq 4.256\,\text{ms}$$

$sig\_delay^{V}_{max}$**:** Bei der Bestimmung der maximalen Übertragungsverzögerung orientieren wir uns an Ausdruck (C.1). Allerdings müssen wir zusätzlich noch die Verzögerung für das Auslesen eines vollständigen empfangenen Rahmens aus dem RXFIFO berücksichtigen, da für diese Zeit der SPI-Bus blockiert ist und der CC2420-Transceiver somit keine neuen Befehle entgegennehmen kann. Dementsprechend setzt sich $d^{v_2}_{rxProcDelay}$ aus einer Verzögerung für die Verarbeitung des Interrupts, dem Zugriff auf den SPI-Bus sowie der Übertragung der Daten aus dem RX-FIFO zusammen:

$$d^{v_2}_{rxProcDelay} \approx 15\,\mu\text{s} + 5\,\mu\text{s} + 79,4\,\mu\text{s} = 99,4\,\mu\text{s}$$

Basierend hierauf und unter Verwendung von (C.3) und (C.4) gilt für die maximale Übertragungsverzögerung:

$$\begin{aligned} sig\_delay^{V}_{max} &= \max_{v_1 \in V, v_2 \in V, v_1 \neq v_2} sig\_delay_{max}(v_1, v_2) \\ &= \max_{v_1 \in V, v_2 \in V, v_1 \neq v_2} (d^{v_1}_{txProcDelay} + d^{v_1,v_2}_{PropagationDelay} + d^{v_2}_{rxProcDelay}) \\ &\leq 20,7\,\mu\text{s} + 0,7\,\mu\text{s} + 99,4\,\mu\text{s} = 120,8\,\mu\text{s} \end{aligned}$$

$\max_{v \in V} d^{v}_{TX \to RX}$**:** Diese Größe bezeichnet die maximale Umschaltzeit vom Sende- in den Empfangsmodus. Dieser Vorgang wird nach einer Übertragung automatisch eingeleitet, daher gilt:

$$\max_{v \in V} d^{v}_{TX \to RX} = d^{CC2420}_{tx \to rx} \tag{C.6}$$

D Integration von Mode-Based Scheduling with Fast Mode-Signaling in das FlexRay-Protokoll

D.1 Werkzeuggestützte Überprüfung der Konfigurationsparameter eines Kommunikationszyklus

Das Listing D.1 zeigt die Ausgabe unseres Werkzeuges für die Überprüfung der Konfigurationsparameter eines FlexRay-Kommunikationszyklus (unter Verwendung der Constraints des FlexRay-Standards [Fle10b]). Hier dargestellt ist die Ausgabe für den Kommunikationszyklus, der in Abschnitt 6.2 zur Anwendung kommt. Dieser Kommunikationszyklus basiert auf den von BMW in [BPS08] veröffentlichten Parametern, fehlende Angaben wurden aus den vorhandenen Informationen extrapoliert bzw. sinnvoll abgeschätzt.

Wie in der Ausgabe in Listing D.1 ersichtlich kam es zu keinen Fehlern bei der Überprüfung der Constraints – Verletztungen der Constraints würden ab Zeile 16 mit Angabe der Nummer des verletzten Constraints ausgegeben. Abschließend wird zur besseren Übersicht eine Zusammenfassung der wichtigsten Konfigurationsparameter ausgegeben (ab Zeile 20).

Listing D.1: Überprüfung der Konfigurationsparameter eines Kommunikationszyklus.

```
# Berechnung und Pruefung der FlexRay-Konfigurationsparameter

Suggestion
-------------------------------------------
 C27: gdNIT>= 9
 C37: pLatestTX should <= 250
 C18: gdNIT= 9 (if negative your cycle length is too short.)
 B.4.8 C12,C13: gdActionPointOffset (lower bound with prevention of cliques:)= 2 without
   protection 1
 B.4.8 C12,C14: minimal gdMinislotActionPointOffset= 1
 B.4.11 C18,C19: minimal gdMinislot= 2 (based on specified gdMinActionPointOffset)

These are just hints, not compared with your current configuration values.

Checking Contraints FlexRay 3.0.1
-------------------------------------------
Done

Konfigurationsparameter:
-------------------------------------------
 Basiskonfiguration
 ----------------
  Geschwindigkeit:          10Mbit/s
  Laenge eines Macroticks (gdMacrotick):    1.375 us
  Laenge eines Microticks (pdMicrotick):    25 ns
  Zykluslaenge (gdCycle):        3639 MT = 5003.625 us
   Zykluslaenge in Macroticks (gMacroPerCycle): 3639 MT
   Zykluslaenge in Microticks (pMicroPerCycle): 200145 uT
  BestCasePrecision (aBestCasePrecision):    0.3938208 us
  WorstCasePrecision (aWorstCasePrecision):   1.067698 us
  AssumePrecision (aAssumedPrecision):    0.7307592 us

 Statisches Segment
 ----------------
  Laenge des statischen Segments:       2184 MT= 3003 us
  Anzahl der statischen Slots (gNumberOfStaticSlots):  91
  Laenge eines statischen Slots (gdStaticSlot):   24 MT= 33 us
  Laenge der Payload (gPayloadLengthStatic):    16 Byte
  Gewaehlter ActionPointOffset (gdActionPointOffset):  2 MT

 Dynamisches Segment
 ----------------
  Laenge des dynamischen Segments:     1446 MT = 1988.25 us
  Anzahl der Minislots (gNumberOfMinislots):   289
  Laenge eines Minislots (gdMinislot):    5 MT = 6.875 us
  gdMinislotActionPointOffset:      1 MT
  Letzter Minislot in dem eine Uebertragung starten kann (pLatestTx): 250 -> Abbildbare
    Prioritaeten (max): 250
  Maximale Payload (dyn. Segment)(aPayloadLengthDynMax):    254 Byte
  Laenge des dynamischen Slots fuer die Uebertragung der maximalen Payload: 40 Minislots
    = 200 MT= 145.4545 us
  Laenge der adActionPointDifference:    1 MT

 SymbolWindow
 ----------------
  Laenge des SymbolWindow (gdSymbolWindow):  0 MT= 0 us

 NetworkIdleTime
 ----------------
  Laenge der NIT (gdNIT):     9 MT= 12.375 us
```

D.2 Constraints für die Konfiguration der MSSs

Um die korrekte Funktionsweise von *Fast Mode-Signaling* zu garantieren, muss die Länge der Backoffslots ebenso wie der BTSP so gewählt werden, dass alle Knoten einen Übertragungsbeginn sicher dem gleichen Backoffslot zuordnen können. Ebenso muss die Anzahl der benötigten Backoffslots bei der Konfiguration der Länge der statischen Slots berücksichtigt werden. In diesem Abschnitt leiten wir geeignete Constraints für die Konfiguration her, die dies gewährleisten.

Bei der Herleitung der Constraints gehen wir davon aus, dass alle MSSs identisch aufgebaut sind (Länge und Anzahl der Backoffslots, BTSP). Der ActionPointOffset (Abstand zwischen Beginn des statischen Slots und dessen ActionPoint) wird für MSSs und ESSs identisch konfiguriert, um sicherzustellen, dass der minimale Abstand zwischen Übertragungen in zwei benachbarten Slots – unabhängig von deren Typ –, das durch den ActionPointOffset definierte Mindestmaß nicht unterschreiten. Um auf das gemeinsame Zeitverständnis der FlexRay Knoten Bezug nehmen zu können, wird die Länge des Backoffslots d_{back} sowie der Abstand zwischen Beginn des Backoffslots und BTSP d_{BTSP} in Makroticks konfiguriert. Den Konfigurationsparameter zur Festlegung der Länge eines Backoffslots bezeichnen wir, in Anlehnung an den FlexRay-Standard, mit $gdBack \in \mathbb{N}_0$ respektive $gdBTSP \in \mathbb{N}_0$. Wie bereits erläutert fordern wir, dass der ActionPointOffset größer als der Abstand zwischen Backoffslots und des BTSP zu wählen ist:

Constraint D.1

Sei gdActionPointOffset der konfigurierte ActionPointOffset einer FlexRay Konfiguration. Der BTSP muss so gewählt werden, dass für den Abstand gdBTSP zwischen BTSP und Beginn des Backoffslots gilt, dass

$$gdBTSP\,[MT] \leq gdActionPointOffset\,[MT]$$

Bei den Constraints wurde die Schreibweise der *FlexRay Protocol Specification* [Fle10b, Fle05a] übernommen. In eckigen Klammern wird jeweils die Einheit des Parameters angegeben; [MT] steht hierbei für Makroticks. Die Festlegung des ActionPointOffsets erfolgt gemäß den Anforderungen für die klassischen statischen Slots (Constraints 12-13 aus [Fle10b]).

Die Herleitung von Constraints zur Bestimmung der minimalen zulässigen Länge eines Backoffslots sowie der Positionierung des BTSP innerhalb der Backoffslots gestaltet sich sehr einfach, da letztendlich bei der Umsetzung von *Fast Mode-Signaling* innerhalb der MSSs wieder FTDMA als Ausgangsbasis für die Realisierung dienen kann. Konzeptuell besteht jeder MSS aus einem (individuellen) in sich abgeschlossenen *dynamischen Segment* mit genau einem modusbasierten Slot. Die Backoffslots entsprechen in diesem Fall Minislots und der BTSP dem MinislotActionPoint. Für diese liefert die *FlexRay Protocol Specification* bereits entsprechende Constraints. Diese dienen uns als Grundlage, um entsprechende Constraints für die Backoffslots und den BTSP zu formulieren; als Referenz sind jeweils die Nummern der entsprechenden Constraints der Protokollspezifikation angegeben, die hierfür genutzt wurden.

Constraint D.2

Der BTSP muss so gewählt werden, dass für den Abstand gdBTSP zwischen BTSP und Beginn des Backoffslots gilt, dass

$$gdBTSP\,[MT] \geq \left\lceil \frac{aAssumedPrecision\,[\mu s] - adPropagationDelayMin\,[\mu s]}{minMacroSlot\,[\mu s/MT]} \right\rceil$$

mit

$$minMacroSlot\,[\mu s/MT] = \frac{gdMacrotick\,[\mu s/MT]}{1 + gClockDeviationMax} \tag{D.1}$$

$$aAssumedPrecision\,[\mu s] = aWorstCasePrecision\,[\mu s] \tag{D.2}$$

(vgl. Constraint 14 sowie Gleichung 13 aus [Fle10b])

Constraint D.2 bestimmt die untere Grenze für die Wahl des BTSP. Berücksichtigt wird hierbei sowohl die maximal zulässige Uhrenabweichung *aWorstCasePrecision* zwischen zwei Knoten[1] (Empfehlung des Standards für die Bestimmung der Konfigurationsparameter) als auch die minimale Dauer eines Makroticks *minMacroSlot* innerhalb des Clusters (D.1) aufgrund der maximal zulässigen Abweichungen der Taktgeber *gClockDeviationMax* der einzelnen Knoten zueinander (Clock Skew). Ein FlexRay-Knoten startet seine Übertragung jeweils *adPropagationDelayMin* vor dem Erreichen des ActionPointOffset (oder hier dem BTSP) und nutzt auf diese Weise die minimale Übertragungsverzögerung im Cluster aus (sofern diese bekannt ist, ansonsten wird $adPropagationDelayMin = 0$ gesetzt), um den Overhead zu minimieren (vgl. Kapitel 4.4.2).

Die minimale Länge eines Backoffslots lässt sich analog zur minimalen Länge eines Minislots bestimmen:

Constraint D.3

Für die Wahl des Konfigurationsparameters gdBack – der Länge eines Backoffslots in Makroticks – muss gelten, dass

$$gdBack\,[MT] \geq gdBTSP\,[MT] + \left\lceil \frac{aAssumedPrecision\,[\mu s] + adPropagationDelayMax\,[\mu s] - minFrameShort\,[\mu s]}{minMacroSlot\,[\mu s/MT]} \right\rceil$$

mit

$$minFrameShort = \min_{m \in V, n \in V} (dFrameTSSLengthChange_{m,n}\,[\mu s] + dFrameTSSEMIInfluence_{m,n}\,[\mu s]) \tag{D.3}$$

(vgl. Constraint 18 sowie Gleichung 13 in [Fle10b] und Ausdruck (D.1) sowie (D.2) in Coinstraint D.2)

Der Ausdruck aus Constraint D.3 berücksichtigt neben der Wahl des BTSP und der Abweichung der Uhren auch die maximale Übertragungsverzögerung innerhalb des Clusters. Effekte der physikalischen Schicht werden über den Ausdruck *minFrameShort* berücksichtigt und betreffen Modifikationen der Transmission Start Sequence durch die Verwendung von Sternkopplern (vgl. [Fle10b, Abschnitt 3.2.1 – Frame and symbol encoding]). Eine Umrechnung auf Makroticks erfolgt wie in Constraint D.2 durch Division mit der minimalen effektiven Makroticklänge (D.1) des Clusters. Bezüglich der Minislots führt der FlexRay-Standard zusätzlich Constraint 19 [Fle10b] ein, welches jedoch in Bezug auf Backoffslots nicht gilt, da dieses lediglich sicherstellt, dass auch das Ende einer Übertragung von allen Knoten dem

[1] Diese liegt bei einer Übertragungsrate von 10 MBit/s zwischen 8,582 µs bzw. 6,842 µs je nach Länge des größten Mikroticks innerhalb des Clusters.

gleichen Minislot zugeordnet wird. Bei MSSs genügt es zu gewährleisten, dass die Slotgrenzen nicht verletzt werden. Dies ist Gegenstand von Constraint D.4.

Für die Bestimmung der minimal zulässigen Länge der MSSs wird die maximale Anzahl *aMaxSlotModes* von Backoffslots pro MSS benötigt sowie die konfigurierte Länge der Backoffslots. Auf Basis der Constraints der FlexRay-Spezifikation für die Länge klassischer statischer Slots lässt sich dann die minimale Länge *gdStaticMode* eines MSSs bestimmen.

Constraint D.4

Für die minimale Länge gdStaticMode eines MSSs – bei dem die maximale Nutzlast auf gPayloadLengthStaticMode viele 16-Bit-Wörter begrenzt ist – mit aMaxSlotModes Backoffslots muss gelten, dass

$$\begin{aligned} gdStaticMode\,[MT] \geq\ & 2 \cdot gdActionPointOffset\,[MT] \\ & + \left\lceil \frac{(frameDuration\,[\mu s] + propDelay\,[\mu s] + idleDetect\,[\mu s])}{minMacroSlot} \right\rceil \\ & + maxModeDelay\,[MT] \end{aligned} \tag{D.4}$$

mit

$$\begin{aligned} frameDuration\,[\mu s] &= adBitMax\,[\mu s/gdBit] \cdot (gPayloadLengthStaticMode\,[gdBit] \\ &\quad - 0.5 \cdot cdFES\,[gdBit] + cChannelIdleDelimiter\,[gdBit]) && \text{(D.5)} \\ propDelay\,[\mu s] &= adPropagationDelayMin\,[\mu s] + adPropagationDelayMax\,[\mu s] && \text{(D.6)} \\ idleDetect\,[\mu s] &= adMaxIdleDetectionDelayAfterHIGH\,[\mu s] && \text{(D.7)} \\ maxModeDelay\,[MT] &= (aMaxSlotModes - 1) \cdot gdBack\,[MT] && \text{(D.8)} \end{aligned}$$

(vgl. Constraint 20 sowie Gleichung 21 in [Fle10b] und (D.1))

Constraint D.4 ergänzt Constraint 20 in [Fle10b], welches sich auf klassische statische Slots bezieht, um die zusätzlichen Länge der zusätzlichen *aMaxSlotModes* Backoffslots (D.8). Für weitere Details in Bezug auf die Bestimmung der minimalen Länge eines statischen Slots und die Ausdrücke (D.5)-(D.7) sei auf die *FlexRay Protocol Specification* [Fle10b] sowie entsprechende Sekundärliteratur [Rau08, Par12] verwiesen.

Um die Konformität zum FlexRay-Standard zu wahren, müssen die Slots des statischen Segments identisch aufgebaut sein, d.h., MSSs und ESSs müssen die gleiche Größe besitzen sowie die gleiche maximal zulässige Payload erlauben.

D.3 Kommunikationszyklus für drei Modes innerhalb eines MSSs

Das Listing D.2 zeigt einen angepassten Kommunikationszyklus, welcher bis zu drei Modes pro MSS (mit maximal 16 Bytes Nutzdaten pro FlexRay-Rahmen) unterstützt.

Listing D.2: Kommunikationszyklus für *Mode-Based Scheduling with Fast Mode-Signaling* im statischen Segment.

```
1  Basiskonfiguration
   ----------------
3    Geschwindigkeit:      10Mbit/s
     Laenge eines Macroticks (gdMacrotick):   1.375 us
5    Laenge eines Microticks (pdMicrotick):   25 ns
     Zykluslaenge (gdCycle):     3640 MT= 5005 us
7      Zykluslaenge in Macroticks (gMacroPerCycle): 3640 MT
     Zykluslaenge in Microticks (pMicroPerCycle): 200200 uT
9    BestCasePrecision (aBestCasePrecision):   0.3938208 us
     WorstCasePrecision (aWorstCasePrecision):  1.067698 us
11   AssumePrecision (aAssumedPrecision):   0.7307592 us

13 Statisches Segment
   ----------------
15   Laenge des statischen Segments:    2184 MT= 3003 us
     Anzahl der statischen Slots (gNumberOfStaticSlots): 78
17   Laenge eines statischen Slots (gdStaticSlot):  28 MT= 38.5 us
     Laenge der Payload (gPayloadLengthStatic):  16 Byte
19   Gewaehlter ActionPointOffset (gdActionPointOffset): 2 MT

21 BackoffSlots for modebased-Scheduling
   ----------------
23   Laenge eines BackoffSlots (gdBackoffSlot):  2 MT= 2.75 us
     Maximale Payload in einem modebased-Slot:  16 Bytes eingeplant
25   Gewaehlter ActionPointOffset (gdBackoffActionPointOffset): 1 MT
     Additional Idle Detection Phase per BackoffSlot (gdBackoffSlotIdlePhase) of: 1
       BackoffSlots
27
   Dynamisches Segment
29 ----------------
     Laenge des dynamischen Segments:    1441 MT= 1981.375 us
31   Anzahl der Minislots (gNumberOfMinislots):  288
     Laenge eines Minislots (gdMinislot):   5 MT= 6.875 us
33   gdMinislotActionPointOffset:    1 MT
     Letzer Minislot in dem eine Uebertragung starten kann (pLatestTx): 240 -> Abbildbare
       Prioritaeten (max): 240
35   Maximale Payload (dyn. Segment)(aPayloadLengthDynMax): 254 Byte
     Laenge des dynamischen Slots fuer die Uebertragung der maximalen Payload: 40 Minislots =
       200 MT= 145.4545 us
37   Laenge der adActionPointDifference:   1 MT

39 SymbolWindow
   ----------------
41   Laenge des SymbolWindow (gdSymbolWindow):  0 MT= 0 us

43 NetworkIdleTime
   ----------------
45   Laenge der NIT (gdNIT):     15 MT= 20.625 us

47 Using Backoff slots for the static segment assuming a payload of 16 Bytes
   ----------------
49   Minimal length of the static slot using 1 modes per static slot:  24 MT = 33 us
     Minimal length of the static slot using 2 modes per static slot:  26 MT = 35.75 us
51   Minimal length of the static slot using 3 modes per static slot:  28 MT = 38.5 us
```

E Kommunikation zwischen HiL-Stub und HiL-Runtime

Die Kommunikation zwischen HiL-Stub und HiL-Runtime (Nachrichten und Steueranweisungen) erfolgt zunächst über einen TCP/IP-Tunnel zu einem beliebigen Rechner (Gateway). Die empfangenen TCP/IP-Pakete werden dann über die serielle Schnittstelle direkt an den *Imote 2*-Knoten weitergeleitet[1].

Während die Kommunikation auf dem ersten Teilstück durch die Mechanismen von TCP/IP gegenüber Übertragungsfehlern und Paketverlusten abgesichert ist, gilt dies für die Übertragungen auf dem letzten Teilstück nicht. Daher haben wir ein minimalistisches Protokoll für die Kommunikation zwischen HiL-Stub und HiL-Runtime als auch ein spezielles HiL-Rahmenformat entwickelt. Dies erlaubt es, Fehler und Verluste von Rahmen zu detektieren und mittels Backward Error Correction zu korrigieren. Die HiL-Rahmen werden auf dem ersten Teilstück als Payload der TCP/IP Pakete übertragen. Nur HiL-Stub und HiL-Runtime prüfen die HiL-Rahmen auf ihre Korrektheit oder fordern verlorene Rahmen erneut an. Auf diese Weise benötigt der Gateway von TCP/IP zu UART keinerlei Kenntnis des verwendeten Protokolls und des HiL-Rahmenformats.

Abgesichert wird der HiL-Rahmen durch zwei Checksummen. Eine 8-Bit Checksumme (CRC8) sichert den Header und dessen Informationen und eine zusätzliche 32-Bit Checksumme (CRC32) schützt den gesamten Rahmen vor Verfälschungen (vgl. Abbildung E.1). Jeder in der Abbildung E.1 dargestellte Block entspricht einem Byte. In rot bzw. blau ist der Bereich angegeben, welcher durch die jeweilige Checksumme abgesichert wird.

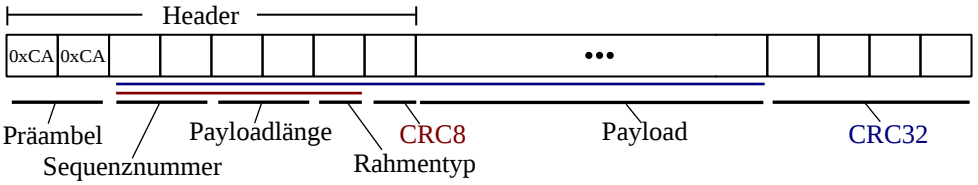


Abbildung E.1: HiL-Rahmenformat für die Kommunikation zwischen HiL-Stub und HiL-Runtime.

Jeder fehlerfrei empfangene Rahmen wird mit einem ACK-Rahmen bestätigt. Bleibt dieser aus, wird der Rahmen nach einem Timeout durch den Sender erneut übertragen. Dieser Ablauf wiederholt sich solange, bis ein ACK empfangen oder die maximale Anzahl von Versuchen erreicht und ein permanenter Kommunikationsfehler detektiert wird. Um Duplikate zu erkennen und die ACKs den jeweiligen Rahmen eindeutig zuordnen zu können, führt jeder Rahmen eine eindeutige Sequenznummer (2-Byte) im Header mit, die bei einer erneuten Übertragung nicht angepasst wird. Ebenso enthält auch das ACK (als Nutzlast) die Sequenznummer des bestätigten Rahmens.

Über das Typ-Feld wird der Rahmentyp und somit die Art der übertragenen Anweisungen sowie der Aufbau der Payload des Rahmens kodiert. Die Tabelle E.2 gibt eine Übersicht über die definierten Rahmentypen, deren Semantik sowie den Aufbau der Payload (diese ist immer ASN.1-kodiert). Zusätzlich ist in der Spalte *Richtung*, die Kommunikationsrichtung angegeben. Hierbei steht R für HiL-Runtime und S für HiL-Stub.

[1] Für die Weiterleitung der TCP/IP-Pakete über die serielle Schnittstelle verwenden wir bereits existierende, frei verfügbare Software. pySerial's TCP/IP - serial bridge als Opensource verfügbar unter http://pyserial.sourceforge.net

Typ	Anweisung	Richtung	Erläuterung	Payload
0x1	SetMessage	S→R	Übertragung einer Nachricht eines InputPorts des HiL-Stubs an die HiL-Runtime	PortValue mit der codierten Nachricht sowie Generierungszeitstempel
0x2	Run	S→R	Steueranweisung an die HiL-Runtime einen Ausführungsschritt zu starten	Aktuelle Simulationszeit und Schrittgröße (zeitgetriggert) bzw. maximale Schrittdauer (ereignisgetriggert)
0x3	Terminated	R→S	Ausführungsschritt beendet	Lokale Simulationszeit des DUT und Zeitpunkt für die nächste Ausführung
0x4	GetMessage	S→R	Abfrage der nächsten ausgehenden Nachricht, welche die HiL-Runtime für einen spezifischen OutputPort bereithält	Name des OutputPorts
0x5	Message	R→S	Übertragung einer von der HiL-Runtime gespeicherten Nachricht für einen spezifischen OutputPort	PortValue mit der codierten Nachricht sowie Generierungszeitstempel
0x6	NoMessage	R→S	Keine (weitere) Nachricht für den angefragten OutputPort gespeichert	
0x7	GetInputPorts	S→R	Abfrage der Namen der InputPorts	
0x8	InputPortNames	R→S	Liste der Namen der InputPorts	Liste der Namen der InputPorts
0x9	GetOutputPorts	S→R	Abfrage der Namen der OutputPorts	
0x10	OutputPortNames	R→S	Liste der Namen der OutputPorts	Liste der Namen der OutputPorts
0x11	LogMessage	R→S	Lognachrichten, die von FERAL ausgegeben werden sollen	Lognachricht
0xFF	ACK	beidseitig	Bestätigung eines Rahmens	Sequenznummer des bestätigten Rahmens

Tabelle E.2: HiL-Rahmentypen

Der Ablauf der Kommunikation ergibt sich aus der Bedeutung der beschriebenen Rahmentypen sowie dem in den Kapiteln 12.2.1 und 12.2.3 geschilderten Zusammenspiel zwischen HiL-Stub und HiL-Runtime. Mit Ausnahme der `LogMessage` wird die Kommunikation vollständig durch den HiL-Stub gesteuert. Dieser stellt jeweils eine Anfrage oder sendet eine Anweisung, die zunächst mit einem ACK bestätigt wird. Danach überträgt die HiL-Runtime ggf. eine entsprechende Antwort. Ein Beispiel: Beim Start des HiL-Stub fragt dieser zunächst die Input- und OutputPorts ab. Dazu sendet er einen Rahmen des Typs (0x7)

und erhält daraufhin ein ACK von der HiL-Runtime (0xFF). Anschließend überträgt die HiL-Runtime die Liste der InputPorts (0x8). Dieser Rahmen wird vom HiL-Stub bestätigt (0xFF). Analog wird bei der Abfrage der OutputPorts verfahren. Dieses Muster prägt die gesamte Kommunikation und erlaubt eine einfache Realisierung der Interaktion.

Publikationsliste

[1] BRAUN, Tobias ; GOTZHEIN, Reinhard ; WIEBEL, Matthias: Integration of FlexRay into the SDL-Model-Driven Development Approach. In: KRAEMER, Frank A. (Hrsg.) ; HERRMANN, Peter (Hrsg.): *System Analysis and Modeling: About Models – SAM 2010, 6th International Workshop on System Analysis and Modeling, Oslo, Norway, October 4–5, 2010, Co-located with MODELS 2010* Bd. 6598, Springer, 2010 (LNCS), S. 56–71

[2] KRÄMER, Marc ; BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard: Real-Time Signaling in SDL. In: OBER, I. (Hrsg.) ; OBER, I. (Hrsg.): *15th International SDL Forum* Bd. 7083, Springer, 2011 (LNCS), S. 184–199

[3] BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; IGEL, Anuschka: Model-driven Engineering of Networked Ambient Systems with SDL-MDD. In: *Procedia Computer Science* 10 (2012), S. 490 – 498. – ISSN 1877–0509. – ANT 2012 and MobiWIS 2012

[4] BRAUN, Tobias ; GOTZHEIN, Reinhard ; FORSTER, Thomas ; KUHN, Thomas: FERAL – Framework for Simulator Coupling on Requirements and Architecture Level. In: *Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign, 18-20 Oktober*, 2013, S. 11–22

[5] BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; IGEL, Anuschka ; FORSTER, Thomas ; KUHN, Thomas: Virtual Prototyping with Feral – Adaptation and Application of a Simulator Framework. In: *The 24th IASTED International Conference on Modelling and Simulation*, 2013

[6] GOTZHEIN, Reinhard ; BRAUN, Tobias: *Zeit- und Prioritätsgesteuerter Sende/Empfangsknoten für FlexRay und LIN*. Oktober 2013. – Patent DE 102012200475 B4

[7] CHRISTMANN, Dennis ; BRAUN, Tobias ; GOTZHEIN, Reinhard: SDL Real-Time Tasks - Concept, Implementation, and Evaluation. In: KHENDEK, Ferhat (Hrsg.) ; TOEROE, Maria (Hrsg.) ; GHERBI, Abdelouahed (Hrsg.) ; REED, Rick (Hrsg.): *SDL Forum* Bd. 7916, Springer, 2013 (Lecture Notes in Computer Science). – ISBN 978–3–642–38910–8, S. 239–257

[8] BRAUN, Tobias ; GOTZHEIN, Reinhard ; KUHN, Thomas: Mode-based Scheduling with Fast Mode-Signaling – A Method for Efficient Usage of Network Time Slots. In: *ICCSIT 2013: 6th International Conference on Computer Science and Information Technology, Paris, France*, 2013

[9] BRAUN, Tobias ; GOTZHEIN, Reinhard ; KUHN, Thomas: Mode-based Scheduling with Fast Mode-Signaling – A Method for Efficient Usage of Network Time Slots. In: *Journal of Advances in Computer Networks (JACN)* Vol. 2 (2014), S. 48–57

[10] BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; IGEL, Anuschka ; KUHN, Thomas: Virtual Prototyping of Distributed Embedded Systems with FERAL. In: *International Journal of Modelling and Simulation* Vol. 34 (2014)

[11] BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; MATER, Alexander: SDL Implementations for Wireless Sensor Networks - Incorporation of PragmaDev's RTDS into the

Deterministic Protocol Stack BiPS. In: *System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014, Valencia, Spain, September 29-30, 2014. Proceedings*, 2014, 271–286

[12] BRAUN, Tobias ; CHRISTMANN, Dennis: Simulating Distributed Systems with SDL and Hardware-in-the-Loop. In: *17th International SDL Forum*, Springer, Oktober 2015 *(Accepted Paper)*

Literaturverzeichnis

[AEF+09] AHRENS, Klaus ; EVESLAGE, Ingmar ; FISCHER, Joachim ; KÜHNLENZ, Frank ; WEBER, Dorian: The Challenges of Using SDL for the Development of Wireless Sensor Networks. In: REED, Rick (Hrsg.) ; BILGIC, Attila (Hrsg.) ; GOTZHEIN, Reinhard (Hrsg.): *SDL 2009: Design for Motes and Mobiles*, Bd. 5719. Springer Berlin Heidelberg, 2009. – ISBN 978–3–642–04553–0, S. 200–221

[AG04] ARACIL, Javier ; GORDILLO, Francisco: The Inverted Pendulum: A Benchmark in Nonlinear Control. In: *Automation Congress, 2004. Proceedings. World*, Bd. 16 IEEE, 2004, S. 468–482

[AGAT11] ANSSI, Saoussen ; GÉRARD, Sébastien ; ALBINET, Arnaud ; TERRIER, François: Requirements and Solutions for Timing Analysis of Automotive Systems. In: KRAEMER, Frank A. (Hrsg.) ; HERRMANN, Peter (Hrsg.): *System Analysis and Modeling: About Models*, Bd. 6598, Springer Berlin Heidelberg, 2011 (Lecture Notes in Computer Science). – ISBN 978–3–642–21651–0, S. 209–220. http://dx.doi.org/10.1007/978-3-642-21652-7_13

[Ala13] ALAIN TIERRY CHAMAKEN KAMDE: *Model-Based Cross-Design for Wireless Networked Control Systems*, Diss., 2013. – XI, 136 S.

[Alt08] ALTERA CORPORATION: *Cyclone II Device Handbook, Volume 1*. 2008

[Alt09] ALTERA CORPORATION: Understanding Metastability in FPGAs. 2009. – Forschungsbericht

[Alt14a] ALTERA CORPORATION: *ModelSim*. http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html, 2014. – Letzter Abruf 19.03.2014

[Alt14b] ALTERA CORPORATION: *Quartus II*. http://www.altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html, 2014. – Letzter Abruf 19.03.2014

[ARM05] *ARMv5 Architecture Reference Manual, Revision I.* https://silver.arm.com/download/download.tm?pv=1073121. Version: 2005

[Ass] ASSOCIATION FOR STANDARDISATION OF AUTOMATION AND MEASURING SYSTEMS (ASAM): *Field Bus Exchange Format*. http://wwww.asam.net. – Letzter Abruf 24.04.2015

[ASSC02] AKYILDIZ, I. F. ; SU, W. ; SANKARASUBRAMANIAM, Y. ; CAYIRCI, E.: Wireless Sensor Networks: A Survey. In: *Comput. Netw.* 38 (2002), März, Nr. 4, 393–422. http://dx.doi.org/10.1016/S1389-1286(01)00302-4. – ISSN 1389–1286

[AUT14a] AUTOSAR: Release 4.1. Version: 2014. http://autosar.org. 2014. – Specification

[AUT14b] AUTOSAR: Specification of the FlexRay Interface. Version: 2014. http://autosar.org. 2014. – Specification

[BBG05] BEYDEDA, S. ; BOOK, M. ; GRUHN, V.: *Model-driven software development*. Springer, 2005. – ISBN 9783540256137

[BBJM07] BABAOGLU, O. ; BINCI, T. ; JELASITY, M. ; MONTRESOR, A.: Firefly-inspired Heartbeat Synchronization in Overlay Networks. In: *Self-Adaptive and Self-Organizing Systems, 2007. SASO '07. First International Conference on*, 2007, S. 77–86. http://dx.doi.org/10.1109/SASO.2007.25

[BCG09] BECKER, Philipp ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard: Model-Driven Development of Time-critical Protocols with SDL-MDD. In: REED, Rick (Hrsg.) ; BILGIC, Attila (Hrsg.) ; GOTZHEIN, Reinhard (Hrsg.): *14th International SDL Forum*, Bd. 5719, Springer, 2009 (LNCS). – ISBN 978–3–642–04553–0, S. 34–52

[BCG+13] BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; IGEL, Anuschka ; FORSTER, Thomas ; KUHN, Thomas: Virtual Prototyping with Feral – Adaptation and Application of a Simulator Framework. In: *The 24th IASTED International Conference on Modelling and Simulation*, 2013

[BCG+14] BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; IGEL, Anuschka ; KUHN, Thomas: Virtual Prototyping of Distributed Embedded Systems with FERAL. In: *International Journal of Modelling and Simulation* , Vol. 34, (2014)

[BCGM14] BRAUN, Tobias ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; MATER, Alexander: SDL Implementations for Wireless Sensor Networks - Incorporation of PragmaDev's RTDS into the Deterministic Protocol Stack BiPS. In: AMYOT, Daniel (Hrsg.) ; CASAS, Pau F. (Hrsg.) ; MUSSBACHER, Gunter (Hrsg.): *System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014, Valencia, Spain, September 29-30, 2014. Proceedings*, Bd. 8769, Springer, 2014 (Lecture Notes in Computer Science). – ISBN 978–3–319–11742–3, 271–286. http://dx.doi.org/10.1007/978-3-319-11743-0_19

[BF10] BRUMBULLI, Mihal ; FISCHER, Joachim: SDL Code Generation for Network Simulators. In: [KH10], S. 144–155

[BF12] BRUMBULLI, Mihal ; FISCHER, Joachim: Simulation Configuration Modeling of Distributed Communication Systems. In: HAUGEN, Øystein (Hrsg.) ; REED, Rick (Hrsg.) ; GOTZHEIN, Reinhard (Hrsg.): *SAM*, Bd. 7744, Springer, 2012 (Lecture Nodes in Computer Science (LNCS) LNCS 7744). – ISBN 978–3–642–36756–4, S. 198–211

[BGFK13] BRAUN, Tobias ; GOTZHEIN, Reinhard ; FORSTER, Thomas ; KUHN, Thomas: FERAL – Framework for Simulator Coupling on Requirements and Architecture Level. In: *Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign, 18-20 Oktober*, 2013, S. 11–22

[BGK14] BRAUN, Tobias ; GOTZHEIN, Reinhard ; KUHN, Thomas: Mode-based Scheduling with Fast Mode-Signaling – A Method for Efficient Usage of Network Time Slots. In: *Journal of Advances in Computer Networks (JACN)* , Vol. 2, (2014), S. 48–57

[BGW10] BRAUN, Tobias ; GOTZHEIN, Reinhard ; WIEBEL, Matthias: Integration of FlexRay into the SDL-Model-Driven Development Approach. In: [KH10], S. 56–71

[BHLM02] BUCK, Joseph ; HA, Soonhoi ; LEE, Edward A. ; MESSERSCHMITT, David G.: Readings in Hardware/Software Co-design. Version: 2002. http://dl.acm.org/citation.cfm?id=567003.567050. Norwell, MA, USA : Kluwer Academic Publishers, 2002. – ISBN 1–55860–702–1, Kapitel Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, 527–543

[BJL+01] BUFORD, James A. Jr. ; JOLLY, Alexander C. ; LETSON, Kenneth R. ; MOBLEY, Scott B. ; RAY, Jerry A. ; SHOLES, William J.: HWIL weapon system simulations in the U.S. Army Aviation and Missile Command (USAAMCOM). In: *Proc. SPIE* 4366 (2001), S. 82–89. http://dx.doi.org/10.1117/12.438104

[BKPS07] BROY, M. ; KRUGER, IH. ; PRETSCHNER, A ; SALZMANN, C.: Engineering Automotive Software. In: *Proceedings of the IEEE* 95 (2007), Feb, Nr. 2, S. 356–373. http://dx.doi.org/10.1109/JPROC.2006.888386. – ISSN 0018–9219

[BOA+11] BLOCHWITZ, T. ; OTTER, M. ; ARNOLD, M. ; BAUSCH, C. ; CLAUSS, C. ; ELMQVIST, H. ; JUNGHANNS, A. ; MAUSS, J. ; MONTEIRO, M. ; NEIDHOLD, T. ; NEUMERKEL, D. ; OLSSON, H. ; PEETZ, J. v. ; WOLF, S.: The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In: *Proceedings of the 8th International Modelica Conference*, 2011

[BPC+07] BUI, Bach D. ; PELLIZZONI, Rodolfo ; CACCAMO, Marco ; CHEAH, Chin F. ; TZAKIS, Andrew: Soft Real-Time Chains for Multi-Hop Wireless Ad-Hoc Networks. In: *IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE Computer Society, 2007. – ISBN 978–0–7695–2800–7, S. 69–80

[BPG00] BERGWANGER, Josef ; PELLER, Martin ; GRIESSBACH, Robert: byteflight – A New Protocol for Safety Critical Applications. In: *FISITA World Automotive Congress, Seoul, Korea*, 2000

[BPS08] BERGWANGER, Josef ; PETERATZINGER, Martin ; SCHEDL, Anton: FlexRay startet durch – FlexRay Boardnetz für Fahrdynamik und Fahrerassistenzsysteme. In: *Elektronik Automotive, Sonderausgabe BMW 7er*, 2008

[BRSR99] BOOT, R. ; RICHERT, J. ; SCHUTTE, H. ; RUKGAUER, A.: Automated Test of ECUs in a Hardware-in-the-Loop Simulation Environment. In: *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, 1999, S. 587–594. http://dx.doi.org/10.1109/CACSD.1999.808713

[BRW07] BURRI, N. ; RICKENBACH, P. von ; WATTENHOFER, R.: Dozer: Ultra-Low Power Data Gathering in Sensor Networks. In: *6th International Symposium on Information Processing in Sensor Networks (IPSN), 2007.*, 2007, S. 450–459. http://dx.doi.org/10.1109/IPSN.2007.4379705

[BSKS13] BUSCHMANN, Stefan ; STEINBACH, Till ; KORF, Franz ; SCHMIDT, Thomas C.: Simulation based Timing Analysis of FlexRay Communication at System Level. In: *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. New York : ACM-DL, März 2013. – ISBN 978–1–4503–2464–9, S. 285–290

[CAN05] CAN IN AUTOMATION E.V.: *CiA 301 V4.2.0 – CANopen application layer and communication profile*. 2005

[CBG11] CHRISTMANN, Dennis ; BECKER, Philipp ; GOTZHEIN, Reinhard: Priority Scheduling in SDL. In: OBER, I. (Hrsg.) ; OBER, I. (Hrsg.): *15th International SDL Forum*, Bd. 7083, Springer, 2011 (LNCS), S. 200–215

[CBG13] CHRISTMANN, Dennis ; BRAUN, Tobias ; GOTZHEIN, Reinhard: SDL Real-Time Tasks - Concept, Implementation, and Evaluation. In: KHENDEK, Ferhat (Hrsg.) ; TOEROE, Maria (Hrsg.) ; GHERBI, Abdelouahed (Hrsg.) ; REED, Rick (Hrsg.): *SDL Forum*, Bd. 7916, Springer, 2013 (Lecture Notes in Computer Science). – ISBN 978–3–642–38910–8, S. 239–257

[CES04] CULLER, D. ; ESTRIN, D. ; SRIVASTAVA, M.: Guest Editors' Introduction: Overview of Sensor Networks. In: *Computer* 37 (2004), Aug, Nr. 8, S. 41–49. http://dx.doi.org/10.1109/MC.2004.93. – ISSN 0018–9162

[CG12] CHRISTMANN, Dennis ; GOTZHEIN, Reinhard: Real-time Tasks in SDL. In: HAUGEN, Øystein (Hrsg.) ; REED, Rick (Hrsg.) ; GOTZHEIN, Reinhard (Hrsg.): *SAM 2012*, Springer, 2012 (Lecture Notes in Computer Science (LNCS) 7744), S. 53–71

[CGR12] CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; ROHR, Stephan: The Arbitrating Value Transfer Protocol (AVTP) - Deterministic Binary Countdown in Wireless Multi-Hop Networks. In: *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, 2012, S. 1 –9. http://dx.doi.org/10.1109/ICCCN.2012.6289227

[CGSW14] CHRISTMANN, Dennis ; GOTZHEIN, Reinhard ; SIEGMUND, Stefan ; WIRTH, Fabian: Realization of Try-Once-Discard in Wireless Multi-hop Networks. In: *Industrial Informatics, IEEE Transactions on* 10 (2014), Nr. 1, S. 17–26. http://dx.doi.org/10.1109/TII.2013.2281511. – ISSN 1551–3203

[CHL+03] CERVIN, Anton ; HENRIKSSON, Dan ; LINCOLN, Bo ; EKER, Johan ; ÅRZÉN, Karl-Erik: How Does Control Timing Affect Performance? Analysis and Simulation of Timing Using Jitterbug and TrueTime. In: *IEEE Control Systems Magazine* 23 (2003), Juni, Nr. 3, S. 16–30

[Chr15] CHRISTMANN, Dennis: *Distributed Real-time Systems – Deterministic Protocols for Wireless Networks and Model-Driven Development with SDL*, Diss., 2015

[CiA14a] CIA (CAN IN AUTOMATION): *CAN history*. http://www.can-cia.de/index.php?id=systemdesign-can-history. Version: 2014. – Letzter Abruf 21.02.2014

[CiA14b] CIA (CAN IN AUTOMATION): *CANopen device and application profiles*. http://www.can-cia.de/index.php?id=systemdesign-profiles. http://www.can-cia.de/index.php?id=systemdesign-profiles. Version: 2014. – Letzter Abruf 21.02.2014

[Cinar] CINDERELLA APS: *Cinderella*. http://www.cinderella.dk/, 2016

[CKLG09] CHAMAKEN, A. ; KRÄMER, M. ; LITZ, L. ; GOTZHEIN, R.: Model-based C3-Cross-Design for Wireless Networked Control Systems. In: *NES|TCOC Symposium on Recent Trends in Networked Systems and Cooperative Control (NESCOC) and Workshop on Network Induced Constraints in Control (NETCOC), Stuttgart, Germany*, 2009

[CL10] CHAMAKEN, A. ; LITZ, L.: Joint Design of Control and Communication in Wireless Networked Control Systems: A Case Study. In: *American Control Conference (ACC)*, 2010. – ISSN 0743–1619, S. 1835 –1840

[CLKG09] CHAMAKEN, A. ; LITZ, L. ; KRÄMER, M. ; GOTZHEIN, R.: Cross-Layer Design of Wireless Networked Control Systems with Energy Limitations. In: *European Control Conference 2009 (ECC'09)*. Budapest, Hungary, 08 2009

[CMDM07] COPPOLINO, Luigi ; MAURIZIO DI MEGLIO, Enrica R. Nicola Mazzocca M. Nicola Mazzocca: Dependability aspects of automotive x-by-wire technologies. In: *Applied Electronics (AE), 2010 International Conference on*, 2007

[CNM10] CHEN, Deji ; NIXON, Mark ; MOK, Aloysius: *WirelessHART: Real-Time Mesh Network for Industrial Automation*. 1st. Springer Publishing Company, Incorporated, 2010. – ISBN 1441960465, 9781441960467

[Cri07] CRIEGEE, T.: FIBEX Theorie und Praxis. In: *Göpel Automotive Days*, 2007

[Cro07] CROSSBOW TECHNOLOGY INC.: *Imote2 Hardware Reference Manual*. 2007

[CV04] CENA, G. ; VALENZANO, A.: Performance Analysis of Byteflight networks. In: *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, 2004, S. 157–166. http://dx.doi.org/10.1109/WFCS.2004.1377701

[DDY10] DUAN, Jianmin ; DENG, Hualin ; YU, Yongchuan: Co-Simulation of SBW and FlexRay Bus Based on CANoe_MATLAB. In: *IEEE International Conference on Automation and Logistics (ICAL)*, 2010, S. 202–206

[DH98] DEERING, S. ; HINDEN, R.: *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460 (Draft Standard). http://www.ietf.org/rfc/rfc2460.txt. Version: Dezember 1998 (Request for Comments). – Updated by RFCs 5095, 5722

[DK12] DINH, Nguyen Q. ; KIM, Dong-Sung: Performance Evaluation of Priority CSMA-CA Mechanism on ISA100.11a Wireless Network. In: *Computer Standards & Interfaces* 34 (2012), Nr. 1, 117-123. http://dblp.uni-trier.de/db/journals/csi/csi34.html#DinhK12

[DMTT05] DING, Shan ; MURAKAMI, Naohiko ; TOMIYAMA, Hiroyuki ; TAKADA, Hiroaki: A GA-based Scheduling Method for FlexRay Systems. In: *Proceedings of the 5th ACM International Conference on Embedded Software*. New York, NY, USA : ACM, 2005 (EMSOFT '05). – ISBN 1–59593–091–4, 110–113. http://dx.doi.org/10.1145/1086228.1086249

[Dol] DOLPHIN INTEGRATION: *SMASH – Logic, Analog and Mixed simulation*. http://www.dolphin.fr,

[ECG14] ENGEL, Markus ; CHRISTMANN, Dennis ; GOTZHEIN, Reinhard: Implementation and Experimental Validation of Timing Constraints of BBS. In: N. TRIGONI, A.L. M. (Hrsg.): *EWSN*, Bd. 8354, Springer, 2014 (Lecture Notes in Computer Science). – ISBN 978–3–319–04650–1, S. 84–99

[EE03] ELECTRICAL, Institute of ; ENGINEERS, Electronics: *IEEE Standard 802.11h-2003: Spectrum and Transmit Power Management Extension in the 5 GHz Band in Europe*. New York, NY, USA : IEEE Computer Society, 2003

[Eis12] EISELE, Harald: The Benefits of CAN for In-Vehicle Networking. In: *13th International CAN Conference 2012, Hambach (Germany)* Adam Opel AG, 2012

[EJL+03] EKER, J. ; JANNECK, J. W. ; LEE, E. A. ; LIU, J. ; LIU, X. ; LUDVIG, J. ; NEUENDORFFER, S. ; SACHS, S. R. ; XIONG, Y.: Taming Heterogeneity - The Ptolemy Approach. In: *Proceedings of the IEEE* 91 (2003), Nr. 1, S. 127–144

[Eng12] ENGLER, Stefan B.: *OMNeT++ basierte Simulation von FlexRay Netzwerken zur Analyse von Automotive Anwendungen*. Hamburg, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorsthesis, November 2012

[Eng13a] ENGEL, Markus: *Optimierung und Evaluation Black Burst-basierter Protokolle unter Verwendung der Imote 2-Plattform*, TU Kaiserslautern, Diplomarbeit, 2013

[Eng13b] ENGLER, Jonas: *Ein Framework zu einer OMNeT++ basierten Simulation von CAN-Netzwerken auf der Sicherungsschicht*. Hamburg, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorsthesis, Mai 2013

[Ets01] ETSCHBERGER, K.: *Controller Area Network: Basics, Protocols, Chips and Applications*. IXXAT Press, 2001 http://books.google.de/books?id=n1TJAAAACAAJ. – ISBN 9783000073762

[FAF+06] FERREIRA, J. ; ALMEIDA, L. ; FONSECA, J.A ; PEDREIRAS, P. ; MARTINS, E. ; RODRIGUEZ-NAVAS, G. ; RIGO, J. ; PROENZA, J.: Combining Operational Flexibility and Dependability in FTT-CAN. In: *Industrial Informatics, IEEE Transactions on* 2 (2006), May, Nr. 2, S. 95–102. http://dx.doi.org/10.1109/TII.2005.875508. – ISSN 1551–3203

[Fel09] FELSER, Max: *Profibus Handbuch*. epubli.de Verlag, 2009

[FFHS06] FATHY, Hosam K. ; FILIPI, Zoran S. ; HAGENA, Jonathan ; STEIN, Jeffrey L.: Review of Hardware-in-the-Loop Simulation and Its Prospects in the Automotive Area. In: *Proc. SPIE* 6228 (2006). http://dx.doi.org/10.1117/12.667794

[FGJ+05] FLIEGE, Ingmar ; GERALDY, Alexander ; JUNG, Simon ; KUHN, Thomas ; WEBEL, Christian ; WEBER, Christian: Konzept und Struktur des SDL Environment Framework (SEnF) / TU Kaiserslautern. 2005 (341/05). – Forschungsbericht

[FGW06] FLIEGE, Ingmar ; GRAMMES, Rüdiger ; WEBER, Christian: ConTraST – A Configurable SDL Transpiler and Runtime Environment. In: [GR06], S. 216–228

[FHR+04] FOREST, T. ; HEDENETZ, B. ; RAUSCH, M. ; TEMPLE, C. ; EISELE, H. ; ELEND, B. ; UNGERMANN, J. ; KUHLEWEIN, M. ; BELSCHNER, R. ; LOHRMANN, P. u. a.: *Method for transmitting data within a communication system*. https://www.google.com/patents/US20040081193. Version: April 29 2004. – US Patent App. 10/613,088

[FKAE09] FISCHER, Joachim ; KÜHNLENZ, Frank ; AHRENS, Klaus ; EVESLAGE, Ingmar: Model-based Development of Self-organizing Earthquake Early Warning Systems. In: TROCH, I. (Hrsg.) ; BREITENECKER, F. (Hrsg.) ; Vienna University of Technology (Veranst.): *Proceedings MathMod Vienna 2009* Vienna University of Technology, 2009 (Argesim Report 35). – ISBN 978-3-901608-35-3

[Fle05a] FLEXRAY CONSORTIUM: *FlexRay Communication System Protocol Specification V2.1 Rev.A*. FlexRay Consortium, 2005

[Fle05b] FLEXRAY CONSORTIUM: *FlexRay Requirements Specification Version 2.1*. FlexRay Consortium, 2005

[Fle06] FLEXRAY CONSORTIUM: *FlexRay Communication System Electrical Physical Layer Specification V2.1 Rev.B*. 2006

[Fle10a] FLEXRAY CONSORTIUM: *FlexRay Communication System Electrical Physical Layer Specification Version 3.0.1*. 2010

[Fle10b] FLEXRAY CONSORTIUM: *FlexRay Communication System Protocol Specification Version 3.0.1*. 2010

[Fli09] FLIEGE, Ingmar: *Component-based Development of Communication Systems.*, University of Kaiserslautern, Diss., 2009. – 1–228 S. – http://d-nb.info/994854005

[FMD+00] FÜHRER, Thomas ; MÜLLER, Bernd ; DIETERLE, Werner ; HARTWICH, Florian ; HUGEL, Robert: CAN Network with Time Triggered Communication – Time Triggered CAN-TTCAN. In: *7th International CAN Conference, Amsterdam (Netherlands)*, 2000

[FPAF02a] FERREIRA, J. ; PEDREIRAS, P. ; ALMEIDA, L. ; FONSECA, J.: Achieving fault tolerance in FTT-CAN. In: *Factory Communication Systems, 2002. 4th IEEE International Workshop on*, 2002, S. 125–132. http://dx.doi.org/10.1109/WFCS.2002.1159709

[FPAF02b] FERREIRA, J. ; PEDREIRAS, P. ; ALMEIDA, L. ; FONSECA, J.A: The FTT-CAN Protocol for Flexibility in Safety-Critical Systems. In: *Micro, IEEE* 22 (2002), Jul, Nr. 4, S. 46–55. http://dx.doi.org/10.1109/MM.2002.1028475. – ISSN 0272–1732

[Fuj07] FUJITSU LIMITED: *FlexRay ASSP MB88121C, Preliminary Hardware Manual*. 2007

[FW07] FRANK, F. ; WEIGEL, R.: Co-Simulation of SPICE Netlists and VHDL-AMS Models via a Simulator Interface. In: *International Symposium on Signals, Systems and Electronics*, 2007

[GB07] GERKE, Thorsten ; BOLLATI, David: Development of the Physical Layer and Signal Integrity Analysis of FlexRay Design Systems. In: *Simulation & Modelling Mechatronics (SP-2111)*, 2007

[GBP00] GRIESSBACH, Robert ; BERGWANGER, Josef ; PELLER, Martin: byteflight – neues Hochleistungs-Datenbussystem für sicherheitsrelevante Anwendungen. In: *Sonderausgabe Automotive Electronics*, 2000

[GHJV09] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns – Elements of Reusable Object-Oriented Software*. 37. print. Boston [u.a.] : Addison-Wesley, 2009. – XV, 395 S. : Ill., graph. Darst.. – ISBN 0–201–63361–2

[GK08] GOTZHEIN, Reinhard ; KUHN, Thomas: Decentralized Tick Synchronization for Multi-Hop Medium Slotting in Wireless Ad Hoc Networks Using Black Bursts. In: *5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks. SECON'08, San Francisco*, IEEE, June 2008, S. 422–431

[GK11] GOTZHEIN, Reinhard ; KUHN, Thomas: Black Burst Synchronization (BBS) – A Protocol for Deterministic Tick and Time Synchronization in Wireless Networks. In: *Computer Networks* 55 (2011), Nr. 13, S. 3015–3031

[Got07] GOTZHEIN, Reinhard: Model-driven by SDL – Improving the Quality of Networked Systems Development (Invited Paper). In: *Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), Marrakesh, Morocco*, 2007, S. 31–46

[GR06] GOTZHEIN, Reinhard (Hrsg.) ; REED, Rick (Hrsg.): *System Analysis and Modeling - Language Profiles*. Bd. *4320*. Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–68371–2

[Gro12] GROH, Robert: *Beiträge zur Integration von CC2420 und SDL in ns-3*, Technische Universität Kaiserslautern, Fachbereich Informatik, Bachelorarbeit, 2012

[HAR07] HART COMMUNICATION FOUNDATION: *HART Field Communication Protocol Specification, Rev. 7*. 2007

[Har12] HARTWICH, Florian: CAN with Flexible Data-Rate. In: *13th International CAN Conference 2012, Hambach (Germany)* Robert Bosch GmbH, 2012

[HB99] HARTWICH, Florian ; BASSEMIR, Armin: The Configuration of the CAN Bit Timing. In: *6th International CAN Conference, 2-4 November, Turin (Italy)*, 1999

[Hei12] HEINZ, Matthias: *Modellbasierte Entwicklung und Konfiguration des zeitgesteuerten FlexRay Bussystems*. KIT Scientific Publishing, 2012

[HG99] HOFSTEE, J.W. ; GOENSE, D.: Simulation of a Controller Area Network-based Tractor — Implement Data Bus according to {ISO} 11783. In: *Journal of Agricultural Engineering Research* 73 (1999), Nr. 4, 383 - 394. http://dx.doi.org/http://dx.doi.org/10.1006/jaer.1999.0432. – ISSN 0021–8634

[HMFH02] HARTWICH, Florian ; MÜLLER, Bernd ; FÜHRER, Thomas ; HUGEL, Robert: Timing in the TTCAN Network. In: *8th International CAN Conference, Las Vegas, Nevada (USA)*, 2002

[HRP+06] HWANG, Taehun ; ROH, Jihoon ; PARK, Kihong ; HWANG, Jeongho ; LEE, Kyu H. ; LEE, Kang-Won ; LEE, Soo-Jin ; KIM, Young-Jun: Development of HILS Systems for Active Brake Control Systems. In: *SICE-ICASE, 2006. International Joint Conference*, 2006, S. 4404–4408. http://dx.doi.org/10.1109/SICE.2006.314663

[IBMar] IBM CORP.: *Rational SDL Suite*. http://www-01.ibm.com/software/awdtools/sdlsuite/, 2016

[IEC07] IEC: *Industrial communication networks: Fieldbus specifications*. Geneva : IEC, 2007

[IEE99] IEEE: *IEEE Standard 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE Computer Society, 1999 http://standards.ieee.org/getieee802/download/802.11-2007.pdf

[IEE00] IEEE-SA STANDARDS BOARD: *IEEE Standard VHDL Language Reference Manual*. Stuttgart, Germany, 2000

[IEE02] *IEEE 802.15.1-2002 IEEE Standard for Information Technology - Telecommunication and Information Exchange between Systems - LAN/MAN - Part 15.1: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specications for Wireless Personal Area Networks (WPANs)*. 2002

[IEE03] IEEE: *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*. New York, NY, USA : IEEE Computer Society, 2003 http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf

[IEE07] IEEE: *IEEE Standard 802.11e, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 8: Medium Access Control (MAC) Quality of Service Enhancements*. IEEE Computer Society, 2007

[IG13] IGEL, Anuschka ; GOTZHEIN, Reinhard: A CC2420 Transceiver Simulation Module for ns-3 and its Integration into the FERAL Simulator Framework. In: *The Fifth International Conference on Advances in System Simulation*, 2013, S. 156–164

[Ige15] IGEL, Anuschka: *Entwicklung von TDMA-basierten QoS-Routing-Protokollen und Simulationskomponenten für Ad-Hoc-Netze*, Diss., 2015

[Ins12] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS: *IEEE Standard for Ethernet*. New York, NY, USA : IEEE Computer Society, 2012

[Int00] INTERNATIONAL TELECOMMUNICATION UNION (ITU): *ITU-T Recommendation Z.100 Annex F: Formal Semantics Definition*. http://www.itu.int/rec/T-REC-Z.100-200011-I!AnnF1; http://www.itu.int/rec/T-REC-Z.100-200011-I!AnnF2; http://www.itu.int/rec/T-REC-Z.100-200011-I!AnnF3, 2000

[Int06] INTEL CORPORATION: *Intel Mote 2 – Engineering Platform Data Sheet*. 2006

[Int12] INTERNATIONAL TELECOMMUNICATION UNION (ITU): *ITU-T Recommendation Z.100 (12/11) - Specification and Description Language - Overview of SDL-2010*. http://www.itu.int/rec/T-REC-Z.100/en, 2012

[ISA09] ISA-100.11A-2009 STANDARD: *Wireless Systems for Industrial Automation: Process Control and Related Applications*. 2009

[ISO03a] ISO: Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling / International Organization for Standardization. Geneva, Switzerland, 2003 (11898-2:2003). – ISO

[ISO03b] ISO: Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit / International Organization for Standardization. Geneva, Switzerland, 2003 (11898-2:2003). – ISO

[ISO04] ISO: Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication / International Organization for Standardization. Geneva, Switzerland, 2004 (11898-4:2004). – ISO

[Jau05] JAUCH QUARTZ GMBH: *Quarz Crystal JXE75*. 2005

[Joh92] JOHN R. VIG: Introduction to Quartz Frequency Standards. 1992. – Research and Development Report, Army Research Laboratory, Electronics and Power Sources Directorate

[JTN05] JOHANSSON, Karl H. ; TÖRNGREN, Martin ; NIELSEN, Lars: Vehicle Applications of Controller Area Network. In: *Handbook of Networked and Emedded Control Systems*, Springer, 2005

[KASW10] KARNER, Michael ; ARMENGAUD, Eric ; STEGER, Christian ; WEISS, Reinhold: Holistic Simulation of FlexRay Networks by Using Run-Time Model Switching. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010. – ISSN 1530–1591, S. 544–549. http://dx.doi.org/10.1109/DATE.2010.5457145

[KB06] KUHN, Thomas ; BECKER, Philipp: A Simulator Interconnection Framework for the Accurate Performance Simulation of SDL Models. In: [GR06], S. 133–147

[KBH+07] KRAUSE, M. ; BRINGMANN, O. ; HERGENHAN, A. ; TABANOGLU, G. ; ROSENTIEL, W.: Timing Simulation of Interconnected AUTOSAR Software-Components. In: *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, 2007, S. 1–6. http://dx.doi.org/10.1109/DATE.2007.364638

[KBS+13] KARFICH, Oleg ; BARTOLS, Florian ; STEINBACH, Till ; KORF, Franz ; SCHMIDT, Thomas C.: A Hardware/Software Platform for Real-time Ethernet Cluster Simulation in OMNeT++. In: *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. New York : ACM-DL, März 2013. – ISBN 978–1–4503–2464–9, S. 334–337

[KEBB12] KRAJZEWICZ, Daniel ; ERDMANN, Jakob ; BEHRISCH, Michael ; BIEKER, Laura: Recent Development and Applications of SUMO - Simulation of Urban MObility. In: *International Journal On Advances in Systems and Measurements* 5 (2012), December, Nr. 3&4, 128–138. http://elib.dlr.de/80483/

[KG94] KOPETZ, Hermann ; GRÜNSTEIDL, Günter: TTP-A Protocol for Fault-Tolerant Real-Time Systems. In: *Computer* 27 (1994), Januar, Nr. 1, 14–23. http://dx.doi.org/10.1109/2.248873. – ISSN 0018–9162

[KG08] KUHN, Thomas ; GOTZHEIN, Reinhard: Model-Driven Platform-Specific Testing through Configurable Simulations. In: SCHIEFERDECKER, Ina (Hrsg.) ; HARTMAN, Alan (Hrsg.): *ECMDA-FA*, Bd. 5095, Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–69095–5, S. 278–293

[KG13] KUHN, Thomas ; GOTZHEIN, Reinhard: *Zeit- und prioritäts-gesteuerter Sende/empfangsknoten*. 2013. – Patent DE 102010039488 B4

[KGGR05] KUHN, T. ; GERALDY, A. ; GOTZHEIN, R. ; ROTHLÄNDER, Florian: *ns*+SDL – The Network Simulator for SDL Systems. In: PRINZ, A. (Hrsg.) ; REED, R. (Hrsg.) ; REED, J. (Hrsg.): *SDL 2005*, Springer, 2005 (Lecture Notes in Computer Science (LNCS) 3530)

[KGV83] KIRKPATRICK, S. ; GELATT, C. D. ; VECCHI, M. P.: Optimization by Simulated Annealing. In: *Science* 220 (1983), Nr. 4598, 671-680. http://dx.doi.org/10.1126/science.220.4598.671

[KGW06] KUHN, Thomas ; GOTZHEIN, Reinhard ; WEBEL, Christian: Model-Driven Development with SDL – Process, Tools, and Experiences. In: NIERSTRASZ, Oscar (Hrsg.) ; WHITTLE, Jon (Hrsg.) ; HAREL, David (Hrsg.) ; REGGIO, Gianna (Hrsg.): *MoDELS*, Bd. 4199, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–45772–0, S. 83–97

[KH10] KRAEMER, Frank A. (Hrsg.) ; HERRMANN, Peter (Hrsg.): *SAM 2010, 6th Workshop on System Analysis and Modelling*. Bd. *6598*. Springer, 2010 (LNCS)

[KKAM08] KIM, Woo S. ; KIM, Hyun A. ; AHN, Jin-Ho ; MOON, Byungin: System-Level Development and Verification of the FlexRay Communication Controller Model Based on SystemC. In: *Second International Conference on Future Generation Communication and Networking, 2008. FGCN '08*, Bd. 2, 2008, S. 124–127. http://dx.doi.org/10.1109/FGCN.2008.149

[KMT14] KAWAHARA, Keigo ; MATSUBARA, Yutaka ; TAKADA, Hiroaki: A Simulation Environment and Preliminary Evaluation for Automotive CAN-Ethernet AVB Networks. In: *Proceedings of the 1st OMNeT++ Community Summit (Successor of the International Workshop on OMNeT++)*, 2014

[Kop91] KOPETZ, Hermann: Event-Triggered Versus Time-Triggered Real-Time Systems. In: *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*. London, UK : Springer-Verlag, 1991. – ISBN 3–540–54987–0, S. 87–101

[Kop97] KOPETZ, Hermann ; STANKOVIC, John A. (Hrsg.): *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997

[Krä13] KRÄMER, Marc: *Modellgetriebene Entwicklung von Kommunikationsprotokollen für drahtlos vernetzte Regelungssysteme*, Diss., 2013. – V, 181 S.

[Kra14] KRAMER, Christopher: *Drahtlose Kommunikationssysteme für den Produktionsbereich*, TU Kaiserslautern, Masterarbeit, 2014

[Kuh09] KUHN, Thomas: *Model Driven Development of MacZ – A QoS Medium Access Control Layer for Ambient Intelligence Systems*, University of Kaiserslautern, Diss., 2009

[Lee03] LEE, Edward A.: Overview of the Ptolemy Project / University of California, Berkeley. 2003. – Forschungsbericht

[LGTM09] LUKASIEWYCZ, Martin ; GLASS, Michael ; TEICH, Jürgen ; MILBREDT, Paul: FlexRay Schedule Optimization of the Static Segment. In: ROSENSTIEL, Wolfgang (Hrsg.) ; WAKABAYASHI, Kazutoshi (Hrsg.): *CODES+ISSS 2009: Grenoble, France*, ACM, 2009. – ISBN 978–1–60558–628–1, 363-372. http://dblp.uni-trier.de/db/conf/codes/codes2009.html#LukasiewyczGTM09

[LIN10] LIN CONSSORTIUM: LIN Specification Package, Revision 2.2A. 2010. – Specification

[LMV02] LODI, Andrea ; MARTELLO, Silvano ; VIGO, Daniele: Recent advances on two-dimensional bin packing problems. In: *Discrete Applied Mathematics* 123 (2002), Nr. 1–3, 379 - 396. http://dx.doi.org/http://dx.doi.org/10.1016/S0166-218X(01)00347-X. – ISSN 0166–218X

[LSC+05] LOCHERT, C. ; SCHEUERMANN, B. ; CALISKAN, M. ; BARTHELS, A. ; CERVANTES, A. ; MAUVE, M.: Multiple Simulator Interlinking Environment for IVC. In: *2nd ACM International Workshop on Vehicular Ad-Hoc Networks*, ACM, 2005. – ISBN 1–59593–141–4. http://dx.doi.org/10.1145/1080754.1080771

[Lun10] LUNZE, Jan: *Regelungstechnik 1*. Springer, 2010 (Springer-Lehrbuch Bd. 1). – ISBN 978–3–642–13808–9

[LWL08] LI, Fang ; WANG, Lifang ; LIAO, Chenglin: CAN (Controller Area Network) Bus Communication System Based on Matlab/Simulink. In: *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08. 4th International Conference on*, 2008, S. 1–4. http://dx.doi.org/10.1109/WiCom.2008.1004

[Mar12] MARK NIXON: *A Comparison of WirelessHART and ISA100.11a*. Round Rock, USA, 2012

[Mat12] MATHWORKS: *Simulink - Simulation and Model-Based-Design*. http://www.mathworks.com/products/simulink/, 2012

[MEMar] MEMSIC INC.: *Imote2 Datasheet*. http://vs.cs.uni-kl.de/downloads/Imote2NET_ED_Datasheet.pdf, 2016

[Mes] MESSNER, BILL AND TILBURY, DAWN: *Control Tutorials for Matlab and SIMULINK – Inverted Pendulum: System Modeling*. http://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling#1. – Letzter Abruf 06.10.2014

[Mic12a] MICROCHIP: AN754: Understanding Microchip's CAN Module Bit Timing. 2012. – Application Note

[Mic12b] MICROCHIP: *MCP2551: High-Speed CAN Transceiver*. 2012

[MKHC07] MONTENEGRO, G. ; KUSHALNAGAR, N. ; HUI, J. ; CULLER, D.: *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944 (Proposed Standard). http://www.ietf.org/rfc/rfc4944.txt. Version: September 2007 (Request for Comments)

[MMT+13] MATSUMURA, Jun ; MATSUBARA, Yutaka ; TAKADA, Hiroaki ; OI, Masaya ; TOYOSHIMA, Masumi ; IWAI, Akihito: (A Simulation Environment based on OMNeT++ for Automotive CAN–Ethernet Networks). In: *Proceedings of the 4th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS2013)*, 2013, S. 1–6

[MOD10a] MODELISAR CONSORTIUM: Functional Mock-up Interface for Co-Simulation. (2010), October

[MOD10b] MODELISAR CONSORTIUM: Functional Mock-up Interface for Model Exchange. (2010), January

[Mot00] MOTOROLA, INC.: *SPI Block Guide, v03.06*. 2000

[Mtr05] MTRONPI: *Understanding Quartz Crystals*. http://www.mtronpti.com/sites/default/files/files/understanding-quartz-crystals.pdf, 2005 Letzter Abruf 30.04.2014

[Nat96] NATIONAL SEMICONDUCTOR: DS36C250 Controller Area Network (ISO/DIS 11898) Transceiver. 1996. – Datasheet

[Nau05] NAUGHTON, M.: SMART-plan: A New Message Scheduler for Real-Time Control Networks. In: *IET Conference Proceedings* (2005), January, 302-307(5). http://digital-library.theiet.org/content/conferences/10.1049/cp_20050328

[NHN03] NOLTE, T. ; HANSSON, H. ; NORSTROM, C.: Probabilistic Worst-Case Response-Time Analysis for the Controller Area Network. In: *Proceedings. The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003.*, 2003. – ISSN 1545–3421, S. 200–207. http://dx.doi.org/10.1109/RTTAS.2003.1203052

[NL02] NOSSAL, Roman ; LANG, Roland: Model-Based System Development: An Approach to Building X-by-Wire Applications. In: *IEEE Micro* 22 (2002), Juli, Nr. 4, 56–63. http://dx.doi.org/10.1109/MM.2002.1028476. – ISSN 0272–1732

[NP12] NAVET, Nicolas ; PERRAULT, Herve: CAN in Automotive Applications: A Look Forward. In: *13th International CAN Conference 2012, Hambach (Germany)*, 2012

[ns3ar] *The ns-3 Network Simulator*. http://www.nsnam.org, 2016. – Project Homepage, Letzter Abruf 01.08.2014

[NSL08] NAVET, Nicolas ; SIMONOT-LION, Francoise: *Automotive Embedded Systems Handbook*. 1st. Boca Raton, FL, USA : CRC Press, Inc., 2008. – ISBN 084938026X, 9780849380266

[NSSLW05] NAVET, N. ; SONG, YeQiong ; SIMONOT-LION, F. ; WILWERT, C.: Trends in Automotive Communication Systems. In: *Proceedings of the IEEE* 93 (2005), June, Nr. 6, S. 1204–1223. http://dx.doi.org/10.1109/JPROC.2005.849725. – ISSN 0018–9219

[NXP13] NXP: *LPC11Cx2/Cx4 – Rev. 3.1 – Product data sheet*. http://www.nxp.com/documents/data_sheet/LPC11CX2_CX4.pdf. http://www.nxp.com/documents/data_sheet/LPC11CX2_CX4.pdf. Version: 2013. – Letzter Abruf 23.02.2014

[Obj22] OBJECT MANAGEMENT GROUP (OMG): *Unified Modelling Language (UML) Specification: Version 2.4.1, ISO/IEC 19505*. http://www.omg.org/spec/UML/, 20122

[Obj03] OBJECT MANAGEMENT GROUP: *MDA Guide Version 1.0.1*. http://www.omg.org/cgi-bin/doc?omg/03-06-01, 2003. – www.uml.org

[OPA11] OPAL-RT TECHNOLOGIES: *About Hardware in the Loop and Hardware in the loop Simulation*. http://www.opal-rt.com/about-hardware-in-the-loop-and-hardware-in-the-loop-simulation. Version: 2011. – Letzter Abruf 22.07.2014

[Ope10] OPENCORES: Wishbone B4 – WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores. 2010. – Specification

[Ora] ORACLE: *Java Native Interface 6.0 Specification*. https://docs.oracle.com/javase/6/docs/technotes/guides/jni/. – Letzter Abruf 29.07.2014

[OSG14] OSGI ALLIANCE: OSGi Specifications Release 6. Version: June 2014. http://www.osgi.org/Download/Release6. 2014. – Specification

[Par12] PARET, D.: *FlexRay and Its Applications: Real Time Multiplexed Network*. Wiley, 2012. – ISBN 9781119979562

[PAT07] PEREIRA, Nuno ; ANDERSSON, Björn ; TOVAR, Eduardo: WiDom: A Dominance Protocol for Wireless Medium Access. In: *IEEE Trans. Industrial Informatics* 3 (2007), Nr. 2, S. 120–130

[PB15] PATIL, Prajakta ; BHOSALE, Snehal: Review on Hardware-in-Loop Simulation used to Advance Design Efficiency and Test Competency. In: *International Journal of Science and Research (IJSR)* 4 (2015), Nr. 3

[PC11] PETERSEN, S. ; CARLSEN, S.: WirelessHART Versus ISA100.11a: The Format War Hits the Factory Floor. In: *Industrial Electronics Magazine, IEEE* 5 (2011), Dec, Nr. 4, S. 23–34. http://dx.doi.org/10.1109/MIE.2011.943023. – ISSN 1932–4529

[PDB08] PAQUIN, Jean-Nicolas ; DUFOUR, Christian ; BÉLANGER, Jean: A Hardware-In-the-Loop Simulation Platform for Prototyping and Testing of Wind Generator Controllers. In: *CIGRÉ Canada, Conference on Power Systems, Winnipeg, October 19-21*, 2008, S. 107–112

[PDO02] PAL, Abhishek ; DOGAN, Atakan ; ÖZGÜNER, Füsun: MAC Layer Protocols for Real-Time Traffic in Ad-Hoc Wireless Networks. In: *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing*. Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7695–1677–7, S. 539

[PGAB05] PEDREIRAS, P. ; GAI, P. ; ALMEIDA, L. ; BUTTAZZO, G.C.: FTT-Ethernet: A Flexible Real-Time Communication Protocol That Supports Dynamic QoS Management on Ethernet-Based Systems. In: *Industrial Informatics, IEEE Transactions on* 1 (2005), Aug, Nr. 3, S. 162–172. http://dx.doi.org/10.1109/TII.2005.852068. – ISSN 1551–3203

[PI00] POLLINI, L. ; INNOCENTI, M.: A Synthetic Environment for Dynamic Systems Control and Distributed Simulation. In: *Control Systems, IEEE* 20 (2000), Apr, Nr. 2, S. 49–61. http://dx.doi.org/10.1109/37.833640. – ISSN 1066–033X

[PM12] POPOVICI, K. ; MOSTERMAN, P.J.: *Real-Time Simulation Technologies: Principles, Methodologies, and Applications*. CRC Press, 2012 (Computational Analysis, Synthesis, and Design of Dynamic Systems). https://books.google.de/books?id=GvlzMPbuZqAC. – ISBN 9781439847237

[Pos80] POSTEL, J.: *User Datagram Protocol*. RFC 768 (Standard). http://www.ietf.org/rfc/rfc768.txt. Version: August 1980 (Request for Comments)

[Pra13] PRAGMADEV SARL: *Real Time Developer Studio: User Manual*. http://www.pragmadev.com, 2013

[Praar] PRAGMADEV SARL: *Real Time Developer Studio*. http://www.pragmadev.com, 2016

[Pto14] PTOLEMAEUS, Claudius (Hrsg.): *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014 http://ptolemy.org/books/Systems

[PTVar] PTV PLANUNG TRANSPORT VERKEHR AG: *PTV Vissim traffic simulator*. http://vision-traffic.ptvgroup.com/de/produkte/ptv-vissim/, 2016

[PXA13] *Marvell Technology Group Ltd.: Marvell PXA Family*. http://www.marvell.com/application-processors/pxa-family/. Version: 2013. – Letzter Abruf 16.10.2014

[Rau08] RAUSCH, M.: *FlexRay: Grundlagen, Funktionsweise, Anwendung*. Hanser, 2008. – ISBN 9783446412491

[Rey13] REYES, Victory ; INC., Synopsys (Hrsg.): *Virtual Hardware „In-the-Loop“: Earlier Testing for Automotive Applications*. http://www.synopsys.com/. Version: 2013

[Rob91] ROBERT BOSCH GMBH: *CAN Specification Version 2.0*. Stuttgart, Germany, 1991

[Rob99] ROBERT BOSCH GMBH: *VHDL Reference CAN – User's Manual Revision 2.2*. Stuttgart, Germany, 1999

[Rob07] ROBERT BOSCH GMBH: *E-Ray, FlexRay IP-Module, User's Manual*. November 2007

[Rob12] ROBERT BOSCH GMBH: *CAN with Flexible Data-Rate 1.0*. 2012

[Rus01] RUSHBY, John: A Comparison of Bus Architectures for Safety-Critical Embedded Systems, Springer-Verlag, 2001, S. 306–323

[SBB+02] SHULL, Forrest ; BASILI, Vic ; BOEHM, Barry ; BROWN, A. W. ; COSTA, Patricia ; LINDVALL, Mikael ; PORT, Dan ; RUS, Ioana ; TESORIERO, Roseanne ; ZELKOWITZ, Marvin: What We Have Learned About Fighting Defects. In: *Proceedings of the 8th International Symposium on Software Metrics*. Washington, DC, USA : IEEE Computer Society, 2002 (METRICS '02). – ISBN 0–7695–1339–5, 249–. http://dl.acm.org/citation.cfm?id=823457.824031

[SDKS11] STEINBACH, Till ; DIEUMO KENFACK, Hermand ; KORF, Franz ; SCHMIDT, Thomas C.: An Extension of the OMNeT++ INET Framework for Simulating Real-time Ethernet with High Accuracy. In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. New York : ACM-DL, März 2011. – ISBN 978–1–936968–00–8, S. 375–382

[SGD11] SOMMER, C. ; GERMAN, R. ; DRESSLER, F.: Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. In: *Mobile Computing, IEEE Transactions on* 10 (2011), Jan, Nr. 1, S. 3–15. http://dx.doi.org/10.1109/TMC.2010.133. – ISSN 1536–1233

[Sim] SIMLAB SOFTWARE GMBH: *PCBMod / SLSim*. http://www.simlab.de,

[SIN] *SINNODIUM - Software Innovations For the Digital Company* . http://www.software-cluster.org/en/research/projects/joint-projects/sinnodium. – Letzter Abruf 16.10.2014

[SK96] SOBRINHO, J.L. ; KRISHNAKUMAR, A.S.: Real-Time Traffic over the IEEE 802.11 Medium Access Layer. In: *Bell Labs Technical Journal* Autumn 1996 (1996), autumn, S. 172–187

[SKG+06] SIDDIQUE, Mohammad M. ; KOENSGEN, Andreas J. ; GOERG, Carmelita ; HIERTZ, Guido R. ; MAX, Sebastian: Extending IEEE 802.11 by DARPA XG Spectrum Management: A Feasibility Study. In: *Wireless Conference 2006 - Enabling Technologies for Wireless Multimedia Communications (European Wireless), 12th European*, 2006, S. 1–7

[SKG07] SIDDIQUE, M. ; KÖNSGEN, A. ; GÖRG, C.: Vertical Coupling between Network Simulator and IEEE 802.11 Based Simulator. In: *International Conference on Information & Communication Technology (ICICT)*, 2007, S. 127–130

[sma] *Technologie-Initiative SmartFactory KL*. http://www.smartfactory-kl.de/. – Letzter Abruf 16.10.2014

[SS06] STROOP, J. ; STOLPE, R.: Prototyping of Automotive Control Systems in a Time-Triggered Environment Using Flexray. In: *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, 2006, S. 2332–2337

[SS07] SCHMIDT, K. ; SCHMIDT, E.G.: Systematic Message Schedule Construction for Time-Triggered CAN. In: *Vehicular Technology, IEEE Transactions on* 56 (2007), Nov, Nr. 6, S. 3431–3441. http://dx.doi.org/10.1109/TVT.2007.906413. – ISSN 0018–9545

[SS09a] SCHMIDT, E.G. ; SCHMIDT, K.: Message Scheduling for the FlexRay Protocol: The Dynamic Segment. In: *Vehicular Technology, IEEE Transactions on* 58 (2009), Jun, Nr. 5, S. 2160–2169. http://dx.doi.org/10.1109/TVT.2008.2008653. – ISSN 0018–9545

[SS09b] SCHMIDT, K. ; SCHMIDT, E.G.: Message Scheduling for the FlexRay Protocol: The Static Segment. In: *Vehicular Technology, IEEE Transactions on* 58 (2009), Jun, Nr. 5, S. 2170–2179. http://dx.doi.org/10.1109/TVT.2008.2008654. – ISSN 0018–9545

[SSK09] SCHUMACHER, Henrik ; SCHACK, Moritz ; KÜRNER, Thomas: Coupling of Simulators for the Investigation of Car-to-X Communication Aspects. In: *4th IEEE Asia-Pacific Services Computing Conference, IEEE APSCC 2009, Singapore, December 7-11 2009*, 2009, S. 58–63

[SSP06] SCHELER, Fabian ; SCHRÖDER-PREIKSCHAT, Wolfgang: Time-Triggered vs. Event-Triggered: A matter of configuration? In: DULZ, Winfried (Hrsg.) ; SCHRÖDER-PREIKSCHAT, Wolfgang (Hrsg.): *MMB Workshop Proceedings*. Berlin, 2006. – ISBN 978–3–8007–2956–2, 107–112. http://www4.cs.fau.de/Publications/2006/scheler_06_mmb-nfp.pdf

[STM11] STMICROELECTRONICS: Reference Manual 0090: STM32F40xxx, STM32F41xxx, STM32F42xxx, STM32F43xxx advances ARM-based 32-bit MCUs. 2011. – Reference Manual

[STM12] STMICROELECTRONICS: UM1472 User Manuel: STM32F4Discovery. 2012. – User Manual

[Tan03] TANENBAUM, Andrew S.: *Computer Networks*. 4. Prentice Hall International, Inc., 2003

[Tan07] TANENBAUM, Andrew S.: *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2007. – ISBN 9780136006633

[TB94] TINDELL, K. ; BURNS, A.: Guaranteeing Message Latencies on Controller Area Network (CAN). In: *1st International CAN Conference*, 1994

[Tem98] TEMPLE, C.: Avoiding the Babbling-Idiot Failure in a Time-Triggered Communication System. In: *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, 1998. – ISSN 0731–3071, S. 218–227. http://dx.doi.org/10.1109/FTCS.1998.689473

[Tex02] TEXAS INSTRUMENTS: SN65HVD230, 3.3-V CAN Transceiver. 2002. – Datasheet

[Tex07] TEXAS INSTRUMENTS: *CC2420 Datasheet*. http://focus.ti.com/lit/ds/symlink/cc2420.pdf, 2007. – Revision SWRS041b

[Tex08a] TEXAS INSTRUMENTS: Introduction to the Controller Area Network (CAN). 2002, Revised 2008. – Application Report

[Tex08b] TEXAS INSTRUMENTS: Controller Area Network Physical Layer Requirements. 2008. – Application Report

[Tex14] TEXAS INSTRUMENTS: *Tiva C Series ARM Microcontrollers*. 2014

[TH08] TRIKALIOTIS, Spiro ; HIERTZ, Guido: *Verkehrssteuerung im WLAN – Quality of Service in Funknetzen*. http://www.heise.de/netze/artikel/Verkehrssteuerung-im-WLAN-221449.html, 2008. – Letzter Abruf 25.04.2014

[The10] THE HAMMERSMITH GROUP: *The Internet of things: Networked object and smart devices*. New York, USA, 2010

[Tri14] TRISTAN GINGOLD: *GHDL – VHDL simulator*. http://gna.org/projects/ghdl/, 2014. – Letzter Abruf 19.03.2014

[TTA04] TTA-GROUP: *TTP – Time-Triggered Protocol TTP/C High-Level Specification Document Protocol Version 1.1*. 2004

[TTA05] TTA-GROUP: *TTP – Frequently Asked Questions*. http://www.ttagroup.org/ttp/faq.htm. http://www.ttagroup.org/ttp/faq.htm. Version: 2005. – Letzter Abruf 18.07.2014

[TTT02] TTTECH: CAN - Byteflight - FlexRay - TTP/C – Technical comparison of protocol properties with focus on safety related applications. 2002. – Technology Training

[USCar] USC INFORMATION SCIENCES INSTITUTE: *The Network Simulator – ns-2*. http://www.isi.edu/nsnam/ns/, 2016

[Var01] VARGA, András: The OMNeT++ Discrete Event Simulation System. In: *Proceedings of the European Simulation Multiconference (ESM'2001)* (2001), June

[VDE12] VDE: *Hardware In the Loop Simulation für den entwicklungsbegleitenden Test von Steuergeräten*. https://www.vde.com/DE/REGIONALORGANISATION/BEZIRKSVEREINE/NORDBAYERN/MITTEILUNGEN/2012/2-2012/Seiten/Hardware%20In%20the.aspx. Version: 2012. – Letzter Abruf 22.07.2014

[Vec] VECTOR INFORMATIK GMBH: *CANoe*. http://vector.com

[Vec08] VECTOR INFORMATIK GMBH, MARKUS SCHWAGER: *Using MATLAB with CANoe*. 2008

[WCSW97] WUENSCH, S. ; CLASS, C. ; SCHWARZ, P. ; WINKLER, F.: Microsystem Design Using Simulator Coupling. In: *Europa Design and Test Conference*, 1997, S. 113–118

[WG92] WALLS, F.L. ; GAGNEPAIN, Jean-Jacques: Environmental sensitivities of quartz oscillators. In: *Ultrasonics, Ferroelectrics, and Frequency Control, IEEE Transactions on* 39 (1992), March, Nr. 2, S. 241–249. http://dx.doi.org/10.1109/58.139120. – ISSN 0885–3010

[Wie10] WIEBEL, Matthias: *Entwicklung eines Treibers zur Integration von FlexRay in das SDL-MDD Diplomarbeit*, Fachbereich Informatik, Technische Universität Kaiserslautern, Diplomarbeit, 2010

[WL88] WELCH, Jennifer L. ; LYNCH, Nancy: A New Fault-Tolerant Algorithm for Clock Synchronization. In: *Information and Computation*, Bd. 77, 1988, S. 1–36

[WNS+05] WILWERT, Cédric ; NAVET, Nicolas ; SONG, Ye-Qiong ; SIMONOT-LION, Françoise u. a.: Design of Automotive X-by-Wire systems. In: *The Industrial Communication Technology Handbook* (2005)

[Wol14] WOLFRAM RESEARCH: *Functional Mock-up Interface (FMI)*. https://reference.wolfram.com/system-modeler/UserGuide/ModelCenterFunctionalMockupInterface.html. Version: 2014

[Xil06] XILINX: *Spartan-3E Starter Kit – Board User Guide*. 2006

[Xil13] XILINX: *Spartan-3E FPGA Family Data Sheet*. 2013

[XZ08] XU, C. ; ZHANG, Y.: Simulation of FlexRay Communication Using C Language. In: *Computer Science and Computational Technology, 2008. ISCSCT '08. International Symposium on*, Bd. 2, 2008, S. 272–276. http://dx.doi.org/10.1109/ISCSCT.2008.292

[YYH03] YOU, Tiantong ; YEH, Chi-Hsiang ; HASSANEIN, H.: A New Class of Collision Prevention MAC Protocols for Wireless Ad hoc Networks. In: *Communications, 2003. ICC '03. IEEE International Conference on*, Bd. 2, 2003, S. 1135 – 1140 vol.2. http://dx.doi.org/10.1109/ICC.2003.1204542

[ZS97] ZUBERI, K.M. ; SHIN, K.G.: Scheduling Messages on Controller Area Network for Real-Time CIM Applications. In: *IEEE Transactions on Robotics and Automation* 13 (1997), Apr, Nr. 2, S. 310–316. http://dx.doi.org/10.1109/70.563654. – ISSN 1042–296X

[ZS08] ZIMMERMANN, Werner ; SCHMIDGALL, Ralf: *Bussysteme in der Fahrzeugtechnik: Protokolle und Standards*. 3. Wiesbaden : Vieweg, 2008. – ISBN 978–3–8348–0447–1

[ZWT09] ZIERMANN, T. ; WILDERMANN, S. ; TEICH, J.: CAN+: A new backward-compatible Controller Area Network (CAN) protocol with up to 16x higher data rates. In: *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009. – ISSN 1530–1591, S. 1088–1093. http://dx.doi.org/10.1109/DATE.2009.5090826

Tobias Braun

Lebenslauf

Schulbildung

1989–1993	Grundschule Langenlonsheim
1993–2002	Gymnasium am Römerkastell, Bad Kreuznach

Studium

seit 04/2003	Studium der Informatik an der Technischen Universität Kaiserslautern
	Projektarbeit: "Energieverbrauch von Suchalgorithmen"
	Diplomarbeit: "Automatisierte Analyse von Algorithmen auf Basis von Spursprachen"
03/2009	Abschluss des Studiums mit Titel Diplom-Informatiker

Beruflicher Werdegang

05/2009	Wissenschaftliche Hilfskraft
seit 05/2009	Wissenschaftlicher Mitarbeiter bei der Arbeitsgruppe *Vernetzte Systeme* der Technischen Universität Kaiserslautern