Programmnetzlisten: Ein formales Modell für die Verifikation von Hardwarenaher Software in Eingebetteten Systemen

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

von
Dipl.-Ing. Bernard Schmidt
geboren in Lyon (Frankreich)

D 386

Dekan:	Prof. Dr.-Ing. Ralph Urbansky
Vorsitzender der Prüfungskommission:	Prof. Dr. Gerhard Fohler
Gutachter:	Prof. Dr.-Ing. Dr. rer. nat. Wolfgang Kunz
	Prof. Dr. Christoph Grimm
Tag der Einreichung:	25. Oktober 2019
Tag der Disputation:	28. Februar 2020

Kaiserslautern, 2020

Zusammenfassung

In dieser Arbeit wird ein formales Modell zur Beschreibung von hardwarenaher Software vorgestellt: Die Programmnetzliste. Die Programmnetzliste (PN) besteht aus Instruktionszellen die in einem gerichteten azyklischen Graph verbunden sind und dabei alle Ausführungspfade des betrachteten Programms beinhaltet. Die einzelnen Instruktionszellen repräsentieren eine Instruktion oder eine Instruktionssequenz. Die PN verfügt über eine explizite Darstellung des Programmablaufs und eine implizite Modellierung des Datenpfads und ist als Modell für die Verifikation von Software nutzbar. Die Software wird dabei auf Maschinencode-Level betrachtet.

Die Modellgenerierung besteht aus wenigen und gut automatisierbaren Schritten. Als Grundlage dient ein – ggf. unvollständiger – Kontrollfluss Graph (CFG), der aus der Software generiert werden kann. Die Modellgenerierung besteht aus zwei Schritten.

Der erste Schritt ist die Erzeugung des expliziten Programmablaufs, indem der CFG abgerollt wird. Dabei wird ein sogenannter Execution-Graph (EXG) erzeugt, der alle möglichen Ausführungspfade des betrachteten Programms beinhaltet. Um dieses Modell so kompakt wie möglich zu halten, werden unterschiedliche Techniken verwendet – wie das Zusammenführen gemeinsamer Pfade und das Erkennen von “toten” Verzweigungen im Programm, die an der entsprechenden Stelle niemals ausgeführt werden.

Im Anschluss wird im zweiten Schritt der Execution-Graph in die Programmnetzliste (PN) übersetzt. Dabei werden alle Knoten im EXG durch eine entsprechende Instruktionszelle ersetzt. Die Kanten des Graphen entsprechen dabei dem Programmzustand. Der Programmzustand setzt sich aus den Variablen im Speicher wie auch dem Architekturzustand des unterliegenden Prozessors zusammen.

Ergänzt wird der Programmzustand in der Programmnetzliste um ein sogenanntes Active-Bit, welches es ermöglicht den aktiven Pfad in der Netzliste zu markieren. Das ist notwendig, da die Software immer nur einen Pfad gleichzeitig ausführen kann, aber die PN alle möglichen Pfade beinhaltet. Auf der Programmnetzliste können dann mit Hilfe von Hardware Property Checkern basierend auf BMC oder IPC diverse Eigenschaften bewiesen werden.

Zusätzlich wird die Programmnetzliste um die Fähigkeit zur Interruptmodellierung erweitert.

Danksagung

Zuerstmöchte ich vor allem Prof. Wolfgang Kunz danken, der mir die Promotion an seinem Lehrstuhl ermöglicht hat und mich im Studium mit dem Thema der formalen Verifikation vertraut machte. In der ganzen Zeit hatte ich immer seine Unterstützung. Prof. Dominik Stoffel danke ich dafür, dass er mich motivierte hier am Lehrstuhl mit der Promotion zu beginnen und mich während der ganzen Promotion immer positiv unterstützte.

Ebenso danke ich auch Prof. Christoph Grimm, der sich als Zweitgutachter zur Verfügung stellte und Prof. Gerhard Fohler der den Vorsitz der Prüfungskommision übernommen hat. Einen besonderen Dank gilt auch Dr. Jörg Bormann.

Allen Kollegen und Mitarbeitern am Lehrstuhl danke ich für die Unterstützung. Danke auch an Binghao, Oliver, Max, Mingh, Thomas, Christian und Carlos für die produktiven und konstruktiven Diskussionen, zu denen sie immer bereit waren, und auch für die gemeinsame Zeit, die wir nach der Arbeit zusammen verbracht haben. Besonders Carlos Villarraga hat viele Stunden mit mir über die Herausforderungen und Ansätze geknobelt und wir haben gemeinsam das EXG-Tool aus der Taufe gehoben.

Danken möchte ich auch meinen Eltern und Klaus mit deren Engelsgeduld während meines Studium und auch Alexandra, die mich ein langes Stück bei der Promotion begleitet hat.

Danke!

Inhaltsverzeichnis

Zusammenfassung **iii**

Danksagung **iv**

Inhaltsverzeichnis **vi**

1 Einleitung **1**
1.1 Vergleich mit dem Stand der Technik 3
1.2 Motivation zur Arbeit 6
1.3 Aufbau der Arbeit 7

2 Grundlagen **9**
2.1 Graphen 9
2.2 Automatenmodelle 12
2.3 Hardware 13
2.3.1 Prozessoren 13
2.3.2 Interrupts und Kontextwechsel 16
2.3.3 Kommunikationsbusse 17
2.3.4 Hardware Modellierung 18
2.4 Software 18
2.4.1 Hardwarenahe Software 19
2.4.2 Software-Modellierung 20
2.4.3 Interrupt-getriebene Software 22
2.5 Nicht-formale Verifikationstechniken (NFV) 23
2.5.1 NFV für Hardware 23
2.5.2 NFV für Software 26
2.6 Formale Verifikationstechniken 27
2.6.1 FV für Hardware 27
2.6.2 FV für Software 31
2.7 HW/SW Co-Verifikation 37
2.8 FV hardwarenaher Software mit Interrupts 39

3 Basisidee für das Modell **41**
3.1 Phase 1: Generierung des Programmflusses 43
3.2 Phase 2: Generierung des Gesamtmodells für die Verifikation 45
3.2.1 Modellierung der Programmsemantik 45
3.3 Interrupt-Modellierung 49
3.4 Vergleich mit vorhandenen Ansätzen 50

4 Modellgenerierung **56**
4.1 Herleitung des Modells 56
4.1.1 Kompaktieren durch Zusammenführen 64
4.2 Modellgenerierung 65
4.2.1 Extraktion des Kontrollflussgraphen 66
4.2.2 Ausrollen des Kontrollflussgraphen 67
4.2.3 Kompaktieren durch Vermeidung toter Verzweigungen 68
4.2.4 Zusammenführen zweier Knoten 69
4.2.5 Unbekannte Verzweigungsziele 71
4.2.6 Infinite Schleifen 72
4.3 Instruktionszellen 72
4.3.1 Generierung der Instruktionszellen 73
4.3.2 Erweiterung der Instruktionszellen-Logik zur Modellierung des Kontrollflusses 74
4.4 Speichermodellierung 75
4.5 Modellierung von Interrupts 76
4.5.1 Vereinfachung mittels “transparenter Interrupts” 77
4.5.2 Interrupt-Timing 79
4.5.3 Zusätzliche hardwareabhängige Aspekte 80

5 Anwendungen der Programmnetzliste **82**
5.1 Property Checking 82
5.2 Equivalence Checking 83
5.3 Überprüfung der Programmnetzlistenerstellung 84
5.4 Unbestimmte Schleifen 86
5.4.1 Behandlung von endlos langen Schleifen 88
5.4.2 Beweis von Idle Funktionen 89

6 Tool **91**
6.1 Modellierung von Speicher 93

7 Experimentelle Ergebnisse **95**
7.1 Seriell-zu-Parallel-Konverter 95
7.2 Software-basierter Selbsttest 96
7.3 Sensorplattform 99
7.4 LIN Master 101

8 Ergebnisse und Ausblick **108**
8.1 Zusammenfassung 108
8.2 Ergebnisse 109
8.3 Ausblick 111
8.3.1 Neue Anwendungsszenarien 111
8.3.2 Erweiterungen des Modells 112

A Implementierung der Instruktionszellen **114**
A.1 Zyklenzähler innerhalb der Instruktionen 119

B Zusätzliche Logik für das Property Checking **122**

C Code-Beispiele **126**
C.1 Eigenschaft: Selbsttest Stuck-at-Fehler erkennen 126
C.2 Eigenschaft: LIN Frame Empfangen von 4 Datenbytes 132

D Abbildungsverzeichnis **134**

E Tabellenverzeichnis **136**

F Literaturverzeichnis **137**

Kapitel 1

Einleitung

In den letzten zwei Jahrzehnten ist der Anteil der Geräte und Produkte, in denen mikroprozessorbasierte Computersysteme als Teil des technischen Systems verwendet werden, stark angestiegen. Solche Systeme werden gemeinhin als *eingebettete Systeme* (engl. "embedded systems") bezeichnet. Die Anwendungsgebiete eingebetteter Systeme reichen von der Kaffeemaschine über das Auto bis hin zur Internationalen Raumstation (ISS). Gleichzeitig sind die Systeme immer komplexer geworden. Während früher viele Steuerungsaufgaben mit reinen Hardwarelösungen realisiert wurden, steigt der Anteil der Systeme mit programmierbaren Komponenten kontinuierlich an.

Die Software wird nicht nur für zusätzliche Komfortfunktionen und größerer Flexibilität eingesetzt, sondern ist vor allem auch ein entscheidender Bestandteil sicherheitsrelevanter Systeme. Zum Beispiel im Bereich der Automobilindustrie hat der Anteil der Software in den eingebetteten Systemen stark an Bedeutung gewonnen [Gri05]. Neben Multimedia- und Komfortanwendungen wie Navigationssystemen und Freisprecheinrichtungen befindet sich auch Software in sicherheitsrelevanten Systemen wie Airbags, Antiblockiersystemen und elektronischen Stabilitätsprogrammen, die zum Teil aktiv und korrigierend in das Fahrverhalten des Fahrzeugs eingreifen können. Generell ist in allen sicherheitsrelevanten Anwendungen das korrekte Zusammenwirken von Hardware und der zugehörigen hardwarenahen Software, die die Kommunikation zwischen der Hardware und der Applikationssoftware realisiert, wesentlich. Erschwerend kommt hinzu, dass Hardware- und Software-Komponenten von unterschiedlichen Firmen entwickelt werden können, was einen erhöhten Entwicklungs- und Verifikationsaufwand für ein reibungsloses und zuverlässiges Zusammenwirken der einzelnen Komponenten erfordert.

Der Einsatz von Software beschränkt sich dabei nicht auf reine anwendungsspezifische Applikationen, die auf einem Prozessor ausgeführt werden, sondern Software kann auch innerhalb eines Hardwaremoduls auftreten, das als programmierbares Subsystem seinerseits in ein größeres eingebettetes System integriert ist. Nach außen hin verhält sich dieses Modul wie eine reine Hardware-Implementierung, es besitzt aber auch einen Prozessor und die entsprechende eingebettete Software, auch Firmware genannt. Solche Prozessoren können stark anwendungsspezifisch sein. ASIPs (*Application-Specific Instruction Set Processors*) sind Prozessoren, deren Instruktionssatz genau auf eine spezifische

Anwendung hin entworfen und optimiert wurde. Diese Optimierung bringt mit sich, dass die zugehörigen Programme bestimmten Einschränkungen unterliegen und nicht beliebige Algorithmen damit implementiert werden können (engl. “weak programmability”, siehe z.B. [LWS+12]). Eine weitere Klasse programmierbarer Subsysteme besteht in Hardwaremodulen für Signalverarbeitung, die mit Hilfe von digitalen Signalprozessoren (DSPs) implementiert sind. Als Beispiel seien MP3-Decoder genannt, z.B. [20112]. Die Software ist fest in einem ROM platziert.

Der Anteil an programmierbaren Systemen nimmt kontinuierlich zu, da diese leichter zu entwickeln sind und relativ einfach aktualisiert werden können - selbst noch nach dem Einbau in ein größeres System. Solche programmierbaren eingebetteten Systeme sind kombinierte Hardware- und Software-Systeme. Das Gesamtverhalten dieser Systeme ergibt sich aus dem Zusammenspiel der kombinierten Hardware- und Software-Komponenten. Dieses Zusammenspiel ist eine neue potenzielle Fehlerquelle, wenn die Zusammenarbeit der beiden Seiten nicht korrekt implementiert wurde.

Eines der bekanntesten Beispiele, bei dem sowohl die Software als auch die Hardware korrekt waren, aber im Zusammenspiel fehlerhaftes Verhalten erzeugt haben, ist der Start des Flugs 501 der Ariane 5 am 4. Juni 1996 [Lio96]. 40 Sekunden nach dem Start wurde die Rakete zerstört, weil sie unerwartet vom Kurs abkam. Die verwendete Software war gut erprobt, übernommen von der Vorgänger-Trägerrakete Ariane 4. Die Hardware der Ariane 5 war fehlerfrei. Der Fehler entstand durch das nicht korrekte Zusammenspiel der Software mit der neuen Hardware aus der Ariane 5 und wurde im Vorfeld durch eine unzureichende Verifikation nicht erkannt. Ursächlich für den Fehler war das Auftreten von größeren Werten in einer 64 Bit Float-Variable, die bei der Konvertierung zu einem 16-Bit Integer nicht mehr darstellbar waren. Diese Werte waren erst durch die neue Hardware der Ariane 5 möglich geworden. Ausgehend von diesem Fehler entstand eine Kette von weiteren Fehlern, die zusammen das Flugverhalten der Rakete veränderte und schließlich zur Sprengung der Ariane 5 führte.

Eine formale Verifikation solcher hardwareabhängigen Software (engl.: hardware-dependent software) zusammen mit der zugehörigen Hardware hätte den Fehler schon vor dem Verlust der Rakete aufdecken können.

Dieses und weitere Beispiele werden auch in [Lig02] und [Der] aufgegriffen, wo ein guter – wenn auch nicht mehr aktueller – Überblick über mögliche Softwarefehler gegeben wird.

Entsprechend der Komplexität der eingebetteten Systeme und der komplexen Interaktion von Hardware und Software nimmt der Aufwand zu, diese Systeme fehlerfrei zu halten. Gerade im Bereich sicherheitskritischer Anwendungen (engl. *safety-critical applications*), bei denen funktionale Fehler Gefahr für Leib und Leben bedeuten können, ist eine ausgereifte und zuverlässige Verifikation unerlässlich.

Die Akzeptanz von formalen Ansätzen bei der Verifikation von funktionalem Verhalten eingebetteter Systeme steigt. Es sind heutzutage ausgereifte kommerzielle Werkzeuge verfügbar, die eine formale Verifikation der Hardware eines Systems-on-Chip (SoC) durchführen können. Dabei werden überwiegend Techniken für den Äquivalenzvergleich oder der Eigenschaftsprüfung eingesetzt. Sie betrachten aber ausschließlich die Hardwa-

rekomponenten. Die meisten SoCs enthalten hingegen einen oder mehrere Prozessoren, so dass nicht nur der Anteil der Software in solchen Systemen wächst, sondern die Software auch das Gesamtverhalten des Systems wesentlich prägt. Daraus ergibt sich die Notwendigkeit, die hardwarenahe Software, die mit der Hardware interagiert, in der Verifikation besonders zu berücksichtigten.

Dabei nimmt die Verzahnung der Hardware und Software zu, so dass es nicht ausreichend ist, nur die Schnittstellen zwischen der Hardware und Software zu kennen und bei der Verifikation zu berücksichtigen. Zusätzlich müssen die Kommunikationssequenzen und auch das "Gedächtnis" der einzelnen Komponenten im Zusammenspiel betrachtet werden. Daher sind separate Untersuchungen der Hardware und der hardwarenahen Software nicht ausreichend, um das Gesamtverhalten des eingebetteten Systems zu beschreiben und zu verifizieren.

In dieser Arbeit wird ein neues Berechnungsmodell für hardwarenahe Software vorgestellt, welches es ermöglicht, die Verifikation von Low-Level-Software in eine vorhandene Hardwareverifikationsumgebung zu integrieren, so dass die Low-Level-Software in ihrer zugehörigen Hardwareumgebung formal verifiziert werden kann.

1.1 Vergleich mit dem Stand der Technik

Das in den Kapiteln 3 und 4 vorgestellte Modell ermöglicht die formale Verifikation der auf dem Prozessor ausgeführten hardwarenahen Software in Bezug zur umgebenden Hardware. Diese kombinierte HW/SW-Sichtweise wurde aus den Bereichen der reinen Hardware- und der reinen Softwareverifikation heraus inspiriert. Entsprechend wird in diesem Kapitel der Stand der Technik in drei Kategorien aufgeteilt. Die erste Kategorie bilden die Methoden, die reine Softwareverifikation betreiben. Die zweite Kategorie beinhaltet die Methoden zur formalen Verifikation von Hardware. Die kombinierten Ansätze, die sowohl Hardware als auch Software in ihrem Verifikationsansatz integrieren, ergeben die verbleibende Gruppe.

Das Feld der formalen Verifikation von Software ist in den letzten Jahrzehnten schon intensiv erforscht worden. Eine gute Übersicht über klassische Ansätze, vor allem solche, die relevant sind im Bereich des Entwurfs eingebetteter Systeme, bieten die Arbeiten [JM09] und [DKW08].

Typisch in diesen Ansätzen ist, dass die Software - bestehend aus einfachen Programmen - als eine Art Finite-State-Transition-System aufgefasst werden kann. Diese können mit Verfahren der Modellprüfung (engl. *model checking*, MC) und ähnlichen Verfahren verarbeitet werden. Die zu verifizierenden Programme sind im Allgemeinen in einer Hochsprache wie C beschrieben. Die Modellprüfungsverfahren werden in zwei Gruppen aufgeteilt: Zum einem in die Methoden, die ein unbeschränktes Modell verwenden, zum anderen diejenigen, die ein endliches Modell besitzen (engl. *bounded model*). Methoden mit unbeschränkten Modellen werden in [BPR01, BHJM07, God05, HP00, Hol97, Dil96, Sch10] verwendet. Endliche Modelle werden in [BH08, IYG+08, CKY03] eingesetzt. Ein ausführlicher Überblick über die einzelnen Methoden wird in Kapitel 2 gegeben.

Unterschiede zwischen den dort vorgestellten Tools resultieren vor allem aus den zugrunde gelegten Beweismethoden. So wird zwischen symbolischen und enumerativen Methoden unterschieden. Die enumerativen Verfahren lassen sich wiederum in zwei Kategorien unterteilen: Die eine Gruppe berechnet die Zwischenzustände und speichert diese ebenfalls, die andere berechnet zwar die Zustände, hält aber nur den aktuellen Zustand vor.

Die meisten Softwareverifikationstechniken, über die in der Literatur berichtet wird, sind hardwareunabhängig. Die Semantik eines Programms wird dort in einer Hochsprache definiert; das Verhalten wird unabhängig von der verwendeten Zielplattform untersucht.

Das wichtigste Merkmal, wodurch sich der in dieser Arbeit vorgeschlagene Ansatz von den vorher aufgezählten Publikationen unterscheidet, ist der Fokus auf hardwareabhängige Low-Level-Software. In der hardwareabhängigen Softwareverifikation wird das Verhalten eines Programms unter Berücksichtigung des Einflusses der konkreten Hardware untersucht. Dabei ist die konkrete Hardware die Plattform, auf der das Programm ausgeführt wird. So eine Plattform besteht mindestens aus einem Prozessor, Speicher, Ein-, Ausgabegeräten und dem Systembus, der die ganzen Komponenten verbindet.

Die hardwareabhängige Betrachtungsweise der Software ist wichtig, wenn ein genaues Verhalten auf der Hardware/Software-Schnittstelle im eingebetteten System betrachtet werden soll. Die Schnittstelle umfasst alle Zugriffe der Software auf die Hardwareumgebung, wie zum Beispiel Zugriffe auf die Ein- und Ausgabegeräte (kurz: E/A). Die Schnittstelle verfügt über eine gegebene Spezifikation, durch die die Kommunikation geregelt wird. Der Inhalt der Kommunikation zwischen der Software und der Hardware – und das entsprechende resultierende Gesamtverhalten des Systems – wird von der Spezifikation nicht abgedeckt. Um zum Beispiel eine softwarebasierte Implementierung eines Kommunikationsprotokolls zu verifizieren, ist es entscheidend, das genaue Verhalten der Software unter Berücksichtigung der korrekten Signale für die verwendete Hardware zu analysieren. Abhängig von diesen Signalen erzeugt die hardwarebasierte E/A-Komponente die Signale, die die Protokollspezifikation erfüllen müssen. Daher beinhaltet das in dieser Arbeit vorgestellte Modell Objekte, die genau diese notwendigen Signale für die Hardware-Architektur repräsentieren.

Mittels dieses Modells ist es möglich, Eigenschaften über das Softwareverhalten zu prüfen und sich dabei dennoch auf konkrete Hardwaresignale im System zu beziehen. Dieselben Hardwaresignale können in Hardware Property Checking referenziert werden und gelten somit auch in der umgebenden Hardware des eingebetteten Systems. Dadurch wird erreicht, dass sowohl das Verhalten der Software als auch das der Hardware in einem Modell abgedeckt werden können.

Der in dieser Arbeit vorgeschlagene Ansatz wurde durch viele Elemente von existierenden Software- und Hardware-Verifikationstechniken beeinflusst.

Die Neuheit des Modells ergibt sich aus der Art, wie diese Elemente kombiniert werden, um ein Berechnungsmodell zu erzeugen, welches besser für die formale Verifikation von hardwareabhängiger Software geeignet ist als die bisherigen Techniken.

Ein direkter Ansatz zur hardwareabhängigen Softwareverifikation basiert auf klassischer Hardwareverifikation mittels Bounded Model Checking (BMC) [BCC+99]. Dabei

wird eine Register-Transfer-Level-Beschreibung (kurz: RTL-Beschreibung) eines Prozessors und seines Speichers, in dem das betrachtete Programm gespeichert ist, für eine endliche Anzahl an Zeitschritten ausgerollt. Diese Anzahl entspricht dabei der Anzahl von Taktzyklen, die benötigt werden, um den längsten Ausführungspfad im Programm darstellen zu können [GKD06, NWSK11]. Auf diese Weise wird ein Modell generiert, das die vollständige Programmsemantik in Hinblick auf die zugrunde liegende Hardware darstellt. Dieser "Brute-Force"-Ansatz ist wegen der vielen auszurollenden Taktzyklen des Prozessors in der Regel nicht für realistische Hardware/Software-Systeme handhabbar. Aus diesem Grund wurde in [NWSK11] ein spezieller Abstraktionsmechanismus für diesen Ansatz vorgeschlagen, um die Komplexität zu reduzieren.

Trotz der Komplexität ist BMC-basierte Hardwareverifikation dennoch attraktiv für die hardwareabhängige Softwareverifikation, da die Hardware leicht in dem gewünschten Detailgrad repräsentiert werden kann. Je nach Verifikationsziel kann der Prozessor mit der zugehörigen Hardware als konkrete RTL-Implementierung dargestellt oder abstrakter auf der "Instruction Set Architecture" (ISA) modelliert werden. Die hardwareabhängige Sicht des Programms ist für die kombinierte HW-/SW-Verifikation notwendig und wird von der hier vorgestellten Methodik ermöglicht, da das komplette Verhalten des Programms implizit mittels ausgerollter Hardware dargestellt wird. Dabei ist zu beachten, dass die Komplexität des Verifikationsproblems nicht ausschließlich von der reinen Größe des ausgerollten Modells abhängt.

Ein entscheidender Nachteil dieses Ansatzes entsteht aus der Tatsache, dass nicht nur die Programmsemantik des individuellen Programmschritts, sondern auch der komplette Programmfluss des Programms ausschließlich implizit durch die Hardware modelliert wird. Der so formulierte BMC-Ansatz hat den Nachteil, dass der zur Beweisführung verwendete SAT-Solver das explizite Wissen über die Pfade des Programmflusses aufwendig mittels Backtracking, Klauseln lernen und ähnlichen Konzepten errechnen muss.

Zum Vergleich verfolgen Software-Verifikationstechniken üblicherweise einen pfadorientierten Ansatz, in dem die einzelnen Pfade explizit durchlaufen und repräsentiert werden. Dies ist zum Beispiel bei Symbolic Execution der Fall, vgl. z.B. [PV09,AEO+08]. Symbolic Execution enumeriert die einzelnen Ausführungspfade des Programms und überprüft die Zustände für die Verifikation entlang dieser Pfade mittels spezieller Verifikationsalgorithmen. Es werden symbolische Formeln erstellt, die eine explizite Repräsentation aller möglichen Zustände - bedingt durch alle möglichen Eingabeszenarien - entlang des betrachteten Pfades darstellen. Diese dienen als Grundlage für die formale Verifikation.

Während eine explizite Repräsentation des Programmflusses ein gewichtiger Vorteil des Ansatzes von Symbolic Execution ist, hat diese Darstellungsform auch gleichzeitig einen Nachteil: Die Programmsemantik in dieser Repräsentation muss explizit mittels Formeln dargestellt und somit explizit berechnet werden. Besonders im Kontext von hardwareabhängiger Softwareverifikation können diese Formeln sehr komplex werden und dadurch die Effizienz der Verifikationsalgorithmen während der Beweisführung negativ beeinflussen.

Verglichen mit dem in dieser Arbeit vorgestellten Ansatz kann davon ausgegangen

werden, dass es zum Beweisen einer zu prüfenden Eigenschaft von Nachteil ist, dass die Evaluierung des Programmflusses und die Programmsemantik in der Darstellungsform im Modell miteinander vermischt sind. In einer hardwareabhängigen Umgebung kann es notwendig sein, das Verifikationsziel zusammen mit einer großen Anzahl an Pfaden zu untersuchen. Das kann dazu führen, dass die symbolischen Formeln unter Umständen "explodieren", d.h. nicht mehr dargestellt werden können.

CBMC [CKY03] ist ein anderer, ebenfalls hardwareunabhängiger Ansatz. Das zu verifizierende C-Programm, welches verifiziert werden soll, wird in ein ausgerolltes Modell überführt, das einem Satz rein kombinatorischer Formeln entspricht, die wie bei BMC in eine SAT-Instanz überführt werden können. Dies kombiniert teilweise die Vorteile von BMC mit der pfadorientierten Sichtweise aus der Softwareverifikation. Allerdings arbeiten die von CBMC generierten SAT-Formeln direkt auf der Hochsprache C. Die Semantik der einzelnen C-Anweisungen ist direkt in das generierte Modell hinein codiert. Dies ist eine adäquate Darstellung für eine hardwareunabhängige Softwareverifikation, wie von CBMC beabsichtigt. Ein Hindernis entsteht allerdings, wenn hardwareabhängige Software auf Basis von Maschineninstruktionen betrachtet wird. Statt direkt auf der Hardware-Beschreibungs-Ebene zu arbeiten, wie es bei dem hier in der Arbeit vorgestellten Ansatz der Fall ist, müsste bei CBMC und anderen hardwareunabhängigen Verifikationsmethoden sowohl das Maschinenprogramm als auch die dazugehörige Hardware in der hohen Programmiersprache modelliert werden.

1.2 Motivation zur Arbeit

Die Techniken, die in dieser Arbeit vorgestellt werden, entstanden als Antwort auf den drängenden Bedarf nach formalen Verifikationstechniken für hardwarenahe Software, die die zugrundeliegende Hardware mitberücksichtigen kann. Es gibt bereits viele Arbeiten für die funktionale Verifikation sowohl von reiner Software als auch reiner Hardware. Im Zusammenspiel von Hardware und Software gibt es jedoch Herausforderungen, bei denen sich das Gesamtverhalten des Systems nicht oder nur mit großem Aufwand mit den klassischen Hardware- oder Softwareverifikationsmethoden verifizieren lässt.

Dabei ist die Schnittstelle, an der das Softwareverhalten in der Hardware sichtbar wird, der Systembus zwischen Prozessor und Hardwareperipherie. Diese Schnittstelle ist in der Regel klar definiert und lässt sich mit klassischer Hardwareverifikation auch formal verifizieren. Beim Zusammenspiel zwischen der Software und anderen Hardwarekomponenten definiert diese Schnittstelle aber nur, wie Informationen übertragen werden, nicht jedoch, ob der Inhalt der Kommunikation korrekt ist. Der Inhalt der Übertragung entscheidet, ob das Zusammenspiel zwischen der Hardware und der Software korrekt ist. Zum Beispiel entscheidet der Inhalt der Daten und Adressen, ob die Daten an die korrekte Adresse transferiert werden und die richtige Sendereihenfolge der Daten eingehalten wird. Diese Eigenschaften lassen sich nur unter gemeinsamer Berücksichtigung von Hardware und Software verifizieren.

Es können dabei unterschiedliche Ansätze verfolgt werden. Eine Möglichkeit besteht

darin, alle Randbedingungen der einen Komponente (zum Beispiel der Seite der Hardware) zu spezifizieren und diese dann auf der anderen Komponente formal zu verifizieren, um zu prüfen, ob diese Bedingungen erfüllt werden. Werden die Randbedingungen von der Seite der Hardware spezifiziert, müssen sie auf der Softwarekomponente formal verifiziert werden. Umgekehrt können die Randbedingungen der Software ebenfalls auf der Hardware-Komponente untersucht werden.

Alternativ ist eine echte Hardware-Software-Co-Verifikation möglich, in der beide Komponenten zusammen formal verifiziert werden, wie es zum Beispiel bei [NWSK11] vorgeschlagen wird. Techniken dieser Art skalieren jedoch schlecht, bzw. es lassen sich nur mit hohen manuellen Aufwand Abstraktionen erzeugen, auf denen die Verifikation durchgeführt werden kann. Eine weitere echte Co-Verifikation ist [MPPK17]. Dort wird das Zusammenspiel von Hardware und Software mittels spezieller Instruktionen erreicht, die wohl überlegt und manuellen Aufwand in den Software Teil eingefügt werden müssen.

Die Ergebnisse der vorliegenden Arbeit und der hier vorgestellten Techniken adressieren die Herausforderung, eine echte Hardware-Software-Co-Verifikation zu ermöglichen und dabei gleichzeitig besser zu skalieren und den manuellen Aufwand möglichst gering zu halten. Der erste Schritt besteht darin, die Software zu modellieren, um eine effiziente funktionale Verifikation mit formalen Techniken zu ermöglichen. Weiterhin soll sich das Modell als Grundlage für eine echte Hardware-Software-Co-Verifikation eignen indem sich das Softwaremodell mit einem existierenden Hardwareverifikationsmodell wieder kombinieren lässt. Dies wird unter anderem in [Vil16] und [VSK18] betrachtet.

1.3 Aufbau der Arbeit

Im folgenden Kapitel 2 werden die für diese Arbeit notwendigen Grundlagen gelegt. Im Anschluss werden gängige Methoden zur formalen Verifikation diskutiert und die im Kapitel 1.1 erwähnten Arbeiten den Techniken nochmals zugeordnet und genauer erörtert.

Danach wird das neue Modell für die Verifikation hardwarenaher Software beschrieben. Die grundlegende Idee wird zunächst in Kapitel 3 vorgestellt. Daran anschließend wird die Idee mit dem Stand der Technik verglichen.

In Kapitel 4 wird das Modell formal beschrieben und dargestellt, inwiefern es das Verhalten der modellierten Software korrekt darstellt. Sodann werden die Details über die Modellgenerierung näher erklärt, die wesentlich für das Modell sind oder zu dessen Optimierung beitragen.

In Kapitel 5 werden wesentliche Anwendungen des neuen Modells vorgestellt, und es wird gezeigt, welche Eigenschaften am Modell formal verifiziert werden können.

Um das hier vorgestellte Modell anwenden zu können, wurde im Rahmen dieser Arbeit ein Tool entworfen, welches in Kapitel 6 beschrieben wird.

Kapitel 7 beschreibt mehrere mit dem neuen Modell durchgeführte Experimente. Die einzelnen Experimente bestehen aus unterschiedlichen Problemstellungen und Beschreibungen, wie mit dem Modell und formaler Eigenschaftsprüfung die Aufgaben gelöst werden.

Im letzten Kapitel wird die Arbeit nochmals zusammengefasst und dabei die wesentlichen Punkte des Modells und der Modellgenerierung aufgezählt. Die Vorzüge und Limitierungen des Modells werden dann im Zusammenhang mit den Experimenten diskutiert. Zum Abschluss werden auf dieser Dissertation aufbauende Forschungsarbeiten und -ideen erörtert, wie zum Beispiel die Nutzung des Modells bei der Äquivalenzprüfung.

Kapitel 2

Grundlagen

Dieses Kapitel beschäftigt sich mit den für diese Arbeit relevanten Grundlagen. Dabei gliedert sich das Kapitel in drei Teile: Den ersten Teil bildet eine kurze Übersicht zu relevanten Themen aus der Graphentheorie und deren Einsatz in dieser Arbeit. Der zweite Teil beschreibt grundlegende Konzepte von Hardware (Kap. 2.3) und Software (Kap. 2.4). Dabei liegt der Fokus auf den Bereichen, die relevant zum Verständnis der Arbeit sind.

Der dritte Teil beschreibt die Verifikationstechniken und erläutert die gängigen Methoden mit dem Schwerpunkt auf der formalen Verifikation. Dabei wird zwischen formalen (Kap. 2.6) und nicht formalen Verifikationstechniken (Kap. 2.5) unterschieden. Die Kapitel selbst unterscheiden nochmals zwischen Hardware- und Softwareverifikation.

Im Anschluss daran werden in Kapitel 2.7 formale Ansätze für die kombinierte Hardware/Software-Co-Verifikation vorgestellt. Zum Abschluss werden in Kapitel 2.8 formale Ansätze für Interrupt-basierte Systeme behandelt.

2.1 Graphen

Ein Graph G ist ein geordnetes Paar (V, E), wobei V eine nichtleere Menge aus Knoten im Graph G ist und $E \subseteq V \times V$ eine Menge von Kanten. Eine Kante $e \in E$ verbindet jeweils zwei Knoten $v, w \in V$. Wird eine Kante $e \in E$ als ungeordnetes Tupel $\{v, w\}$ dargestellt, dann ist der Graph G ein ungerichteter Graph. Ein gerichteter Graph besitzt geordnete Paare der Form $e = (v, w)$, die gerichtete Kanten genannt werden.

Eine Kantenfolge im Graphen G ist eine endliche Folge von adjazenten Kanten der Form $\{\{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_n, v_m\}\}$. Besteht die Kantenfolge nur aus paarweise verschiedenen Kanten wird sie Kantenzug genannt. Ein Pfad ist ein Kantenzug mit paarweisen verschiedenen Knoten, d.h. ein Pfad passiert nicht zweimal denselben Knoten. Ein Pfad in dem der Anfangsknoten v_1 gleich dem Endknoten v_m ist heißt Zyklus. Ein Graph ohne Zyklus wird azyklisch genannt.

Gibt es für jedes beliebige Knotenpaar in einem ungerichteten Graph einen Pfad, der beide Knoten verbindet, so wird der Graphen zusammenhängend genannt. Gilt diese Bedingung ebenfalls in einem gerichteten Graph – mit Pfaden bestehend aus gerichteten Kanten – ist es ein stark zusammenhängender Graph. Gilt sie hingegen nur, wenn die

gerichteten Kanten durch ungerichtete ersetzt werden, ist der Graph schwach zusammenhängend.

Ein zusammenhängender und azyklischer Graph wird Baum genannt. Dieser verfügt über einen explizit markierten Wurzelknoten. Knoten, die nur einen Nachbarn besitzen, werden als Blattknoten oder Blätter bezeichnet. Im Falle eines Baums mit gerichteten Kanten besitzt der Wurzelknoten keine Vorgängerknoten und alle Knoten ohne Nachfolger sind Blätter.

Beispiel: In Abbildung 2.1 a) ist Knoten a der Wurzelknoten. Alle Knoten, die folgen, bilden die inneren Knoten. Die letzten Knoten d, h und i im Graphen, die selbst keine Nachfolger mehr haben, sind die sogenannten Blattknoten.

Gerichtete Graphen wie in Abbildung 2.1 b), die ebenfalls keine Zyklen beinhalten, werden *gerichtete azyklische Graphen* (DAG, engl.: *directed acyclic graphs*) genannt.

In der weiteren Arbeit wird bei Graphen immer von gerichteten Graphen ausgegangen, sofern es nicht ausdrücklich Anderes angegeben wird.

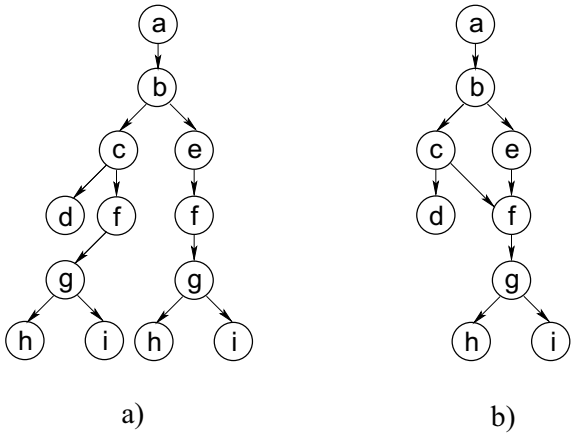

Abbildung 2.1: Gerichtete Graphen: a) Baum, b) DAG

Graphen werden später zur Darstellung von Abläufen oder Strukturen verwendet. Die semantische Bedeutung der einzelnen Elemente wird an den entsprechenden Stellen der Arbeit erläutert.

Weitere Informationen über Graphen können in der Standardliteratur, zum Beispiel in [Nit09], nachgeschlagen werden.

Graphenalgorithmen

Um Graphen verarbeiten zu können werden Suchverfahren verwendet. Mit Hilfe solcher Verfahren können nicht nur bestimmte Knoten gefunden, sondern auch Eigenschaften von Graphen analysiert werden [Sed98]. Die häufigsten Verfahren sind die Tiefensuche (engl. *depth-first search*, DFS) und die Breitensuche (engl. *breadth-first search*, BFS). Bei Ausführung von DFS und BFS in Listing 2.1 und 2.2 werden die Nachfolgeknoten jeweils von dem aktuell bearbeiteten Knoten benötigt. Dafür müssen die Nachfolger – aus den

Knoten V und Kanten E errechenbar – in einer maschinenlesbaren Form abgespeichert werden. Die Eingabe für die Algorithmen ist der Graph in Form einer Adjazenzstruktur Adj, die als Listenelemente in einem Array umgesetzt ist. Da die Berechnung bei beiden Algorithmen identisch ist, wurde die Berechnung nicht explizit dargestellt und der Zugriff auf die Information mittels des Aufruf Adj[v] abstrahiert. Zusätzlich gibt es eine Liste "visited[]", die für jeden Knoten einen Booleschen Wert beinhaltet, der anzeigt ob der jeweilige Knoten schon besucht wurde. Ein Knoten wird, nachdem er besucht wurde, als 'besucht' markiert.

In der Tiefensuche wird zuerst ein Pfad von der Wurzel bis zu einem Pfadende in einen Blattknoten durchlaufen. Wenn eine Verzweigung auf dem Pfad liegt, wird die Verzweigung gespeichert und ein gewählter Zweig weiter abgearbeitet. Erst wenn alle nachfolgenden Knoten in einem Zweig durchlaufen sind, wird der zweite Zweig gewählt. Dieses Verhalten wird durch den rekursiven Aufruf der DFS_VISIT Funktion aus Listing 2.1 erzeugt. Die Funktion DFS_VISIT prüft den aktuellen Arbeitsknoten (engl. *working node*) auf direkte Nachfolger mittels Adj[v] und ruft sich rekursiv für jeden gefundenen Nachfolger wieder auf. Dabei wird der Nachfolgerknoten zum aktuellen Arbeitsknoten. Wenn der Baum aus der Abbildung 2.1 a) durchlaufen wird, dann entsteht bei der Tiefensuche unter der Annahme, dass immer der linke Zweig zuerst bearbeitet wird, folgende Sequenz: a–b–c–d–f–g–h ...

DFS kann auch iterativ mittels eines Stacks implementiert werden.

Listing 2.1: Algorithmus für die Tiefensuche (DFS)

```
1   DFS(Nodes V, Edges E, Startnode s) {
2     for each v ∈ V {
3          visited[v] = false;
4     }
5     DFS_VISIT( V, E, s);
6   }

    DFS_VISIT(Nodes V, Edges E, workingnode v){
9     for each w ∈ Adj[v] {
10       if visited[w] = false {
11         DFS_VISIT(V, E, w);
12       }
      }
      visited[v] = true;
15  }
```

Bei der Breitensuche hingegen werden die Knoten in Abhängigkeit von der Entfernung zum Wurzelknoten durchlaufen. Das bedeutet, dass in diesem Verfahren die Knoten verschiedener Pfade nacheinander bearbeitet werden, ohne einen Pfad abgeschlossen zu haben. Dieses Verhalten wird mit Hilfe einer Schlange (engl. *queue*) queueQ in Listing 2.2 erzeugt. Als erstes wir der Startknoten s in Zeile 5 der queueQ hinzugefügt. Anschließend wird, solange die Schlange nicht leer ist ein neuer Arbeitsknoten aus der Schlange genommen (engl. *dequeue*). Während der Bearbeitung des Knoten werden alle

seine Nachfolgeknoten in die Queue gespeichert (engl. *enqueue*). und im Anschluss nacheinander abgearbeitet. Wenn der Baum aus der Abbildung 2.1 a) durchlaufen wird, dann entsteht bei der Breitensuche mit Annahme, dass mit dem linken Zweig begonnen wird, folgende Sequenz: a–b–c–e–d–f–f ...

Listing 2.2: Algorithmus für die Breitensuche (BFS)

```
1  BFS(Nodes V, Edges E, Startnode s) {
2    for each v ∈ V ∧ v ≠ s {
3      visited[v] = false;
4    }
5    queueQ.enqueue(s);
     while queueQ.not_empty() {
7      v = queueQ.dequeue();
       for each w ∈ Adj[v] {
9        if visited[w] = false {
10         queueQ.enqueue(w);
11       }
         visited[v] = true;
13     }
     }
15 }
```

2.2 Automatenmodelle

Das Verhalten von Hardware wie auch von Software lässt sich durch sogenannte *Automatenmodelle* beschreiben. Ein Automat ist ein Gebilde, das einen internen Zustand besitzt und dessen Ausgabe zu einem bestimmten Zeitpunkt nicht nur von der momentanen Eingabe, sondern auch vom internen Zustand und damit von der Vorgeschichte abhängt.

Man unterscheidet zwischen *Mealy*- und *Moore*-Automaten. Ein endlicher Automat ist definiert durch $M = (I, O, S, S_0, \delta, \lambda)$. Dabei steht I für das Eingabealphabet, welches eine endliche Menge an Eingabesymbolen zulässt. O ist das Ausgabealphabet, eine endliche Menge möglicher Ausgabesymbole. S ist die Menge aller Zustände im Automaten. S_0 ist die Menge aller möglichen Startzustände; es gilt: $S_0 \subseteq S$.

δ definiert eine Übergangsfunktion der Form: $\delta : S \times I \rightarrow S$. λ definiert die Ausgabefunktion und hängt beim Moore-Automaten vom aktuellen Zustand S ($\lambda : S \rightarrow O$) ab und beim Mealy-Automaten zusätzlich auch von der Eingabe I ($\lambda : S \times I \rightarrow O$). Abbildung 2.2 a) zeigt das Blockschaltbild für einen Mealy-Automaten, mit der – für Mealy-Automaten typischer – Verbindung zwischen der Eingabe I und der Ausgabefunktion λ. Diese fehlt einem Moore-Automaten in Abb. 2.2 b). In Teil c) wird ein Automat vereinfacht dargestellt, der das Verhalten des Mealy-Automaten besitzt, der in der weiteren Arbeit verwendet wird. Die Register in den Abbildungen dienen zur Rückkopplung des Systemzustands S und sind an einen Taktgeber angeschlossen. Der Taktgeber dient dazu, das ganze System zu synchronisieren und somit nur einen Gesamtwert rückzukoppeln,

bei dem alle einzelnen rückgekoppelten Signale einen stabilen Wert angenommen haben. Die Übergangsfunktion δ und die Ausgabefunktion λ sind dabei rein kombinatorische Schaltnetze.

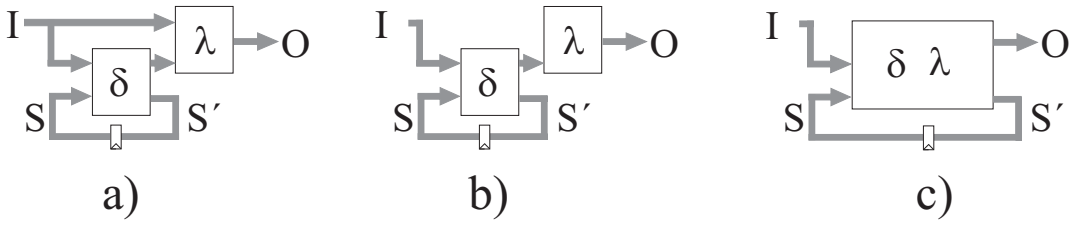

Abbildung 2.2: Automatenmodell: a) Mealy, b) Moore, c) Vereinfacht

2.3 Hardware

Für die Verifikation von hardwarenaher Software ist es entscheidend zu sehen, wie die Software mit der Umgebung interagiert. Ihr Verhalten wird zwar durch das Softwareprogramm spezifiziert, aber letztendlich ist es die Hardware des Prozessors, die die Interaktion mit der Umgebung realisiert. Aus diesem Grund ist es notwendig, die Software in Kombination mit der verwendeten Plattform und dem dazu gehörigen Prozessor zu betrachten.

Wir betrachten im Folgenden den Aufbau und die Funktionsweise des Prozessors und weiterer Hardwarekomponenten, die für die Ausführung von Programmen notwendig sind. Es soll hierbei nur so weit ins Detail gegangen werden, wie für das Verständnis der nachfolgenden Kapitel nötig ist. Einzelheiten können in der Standardliteratur, z.B. [PH07], nachgeschlagen werden.

2.3.1 Prozessoren

Ein Prozessor ist eine digitale Schaltung, die einen Befehlssatz (engl.: *Instruction Set Architecture*, ISA) als Verhalten implementiert und zur freien Programmierung zur Verfügung stellt. Dieser Satz an Instruktionen ermöglicht es, sequenzielles Verhalten zu beschreiben und den Prozessor ausführen zu lassen.

Der Prozessor arbeitet ein Programm ab, welches aus einzelnen Maschinen-Instruktionen besteht, die sequenziell nacheinander ausgeführt werden. Diese Instruktionen liegen im Instruktionsspeicher (engl. *Instruction Memory*), welcher in Abbildung 2.3 dargestellt wird. Um diese Instruktionen auszuführen, wird die jeweilige Instruktion mit Hilfe des *Programmzählers* (engl. *Program Counter*, *PC*), der auf die nächste auszuführende Instruktion des Programms zeigt, aus dem Speicher geladen und in einem speziellen Instruktionsregister gespeichert. Diese Instruktion kodiert nicht nur den Typ der Instruktion, sondern zusätzlich auch die Operanden, die für die jeweilige Operation benötigt werden.

Die Steuerung der Operation wird vom Steuerwerk (engl. *Control*) ausgeführt, das entsprechend der Instruktion den Datenpfad (engl. *Datapath*) konfiguriert. Die Operanden werden aus dem Registersatz geladen, der zusammen mit der Recheneinheit (engl. *Arithmetic Logical Unit, ALU*) und dem Speicher den Datenpfad bilden.

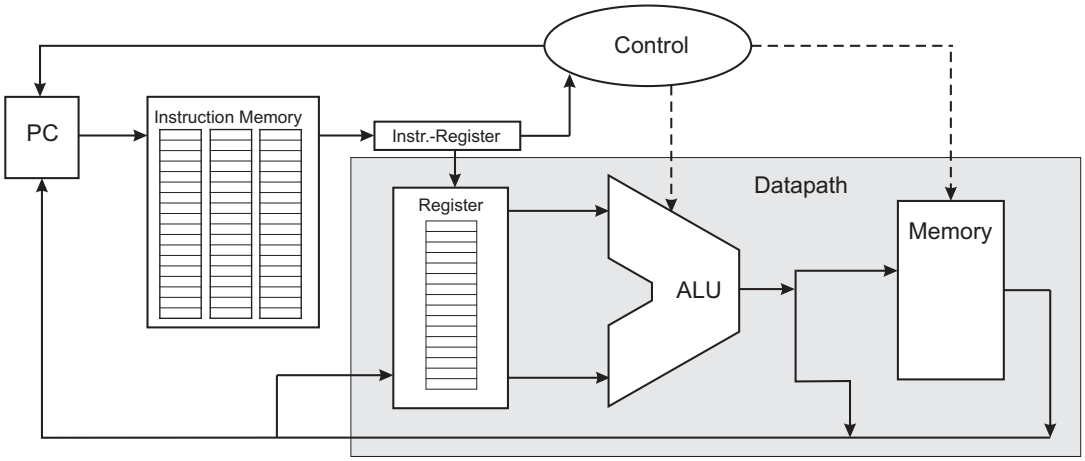

Abbildung 2.3: Prozessormodell [PH07]

Prozessoren lassen sich in unterschiedliche Klassen einteilen. Die wichtigste und bekannteste Einteilung von ist die Unterscheidung zwischen einer *Von-Neumann-Architektur* oder einer sogenannten *Harvard-Architektur*. Der wesentliche Unterschied besteht darin, dass die Harvard-Architektur über einen getrennten Programm- und Datenspeicher verfügt. Die Von-Neumann-Architektur dagegen benutzt einen gemeinsamen Speicher für das Programm und die Daten.

Die meisten Prozessoren gehören zur Klasse der Von-Neumann-Architekturen. Viele dieser Prozessoren besitzen aber zusätzlich Caches zum Beschleunigen der Speicherzugriffe. Teilweise sogar mehrere Ebenen (engl. *Level*). Level-1-Caches sitzen unmittelbar am Prozessor und Puffern meistens Daten und Programm-Instruktionen separat. So greift der Prozessor wie in Abbildung 2.3 separat auf Daten im Speicher und Instruktionen zu und funktioniert wie eine Harvard-Architektur trotz des gemeinsamen Speichers.

Weiterhin werden Prozessoren überwiegend nach ihrem Befehlssatz unterschieden, da dieser maßgeblich die Eigenschaften des Prozessors definiert. Als Oberklasse werden die Prozessoren entweder in *Reduced Instruction Set Computer* (*RISC*) oder *Complex Instruction Set Computer* (*CISC*) unterteilt. Die RISC-Architektur besitzt i.A. eine Load/Store-Architektur, die Zugriffe auf den Speicher mittels Lese- und Schreib-Operationen separat von den arithmetischen und logischen Instruktionen implementiert. Die meisten Instruktionen können ausschließlich mit den in den Registern gespeicherten Werten arbeiten.

Die CISC-Architekturen verfügen dagegen über komplexe Operationen, die sowohl

lesend als auch schreibend auf den Speicher zugreifen können und diese Daten auch direkt verarbeiten.

In eingebetteten Systemen haben sich überwiegend RISC Architekturen durchgesetzt [Tec13].

Aber auch bei den Desktop Systemen haben sich viele Ideen aus der RISC Architektur etabliert. So ist die x86-Architektur – ursprünglich eine CISC-Architektur – seit der Einführung des Pentium Pro eine hybride Architektur die sowohl CISC-Instruktionen, als auch RISC-Konzepte enthält. Die CISC-Instruktionen werden dabei während der Ausführung vom Prozessor in RISC-artige Instruktionen übersetzt [HP07].

Die Architekturen der Prozessoren lassen sich auch anhand der Art und Weise unterscheiden, wie auf die Peripherie zugegriffen wird.

Es gibt zwei Kategorien von Peripheriezugriffen: Die eine unterscheidet zwischen Speicher und Peripherie und besitzt für den jeweiligen Zugriff spezielle Instruktionen, die den Zugriff durchführen. In die zweite Kategorie fallen die Architekturen, die die gleichen Lese- und Schreiboperationen für den Zugriff sowohl auf den Speicher als auch auf die Peripherie verwenden. Dabei wird der Adressraum zwischen Speicher und Peripherie aufgeteilt und mittels der verwendeten Adresse entschieden, ob auf den Speicher oder auf ein Ein-/Ausgabe-Device zugegriffen wird. Das Verfahren wird *Memory-mapped I/O* genannt, da der Prozessor die Peripherie mittels desselben Adressraums adressiert, der klassischerweise dem Speicher vorbehalten ist.

Die Aufgaben, die mit Hilfe eines Prozessors gelöst werden können, werden immer komplexer. Entsprechend entwickeln sich die Prozessoren weiter, und es werden zunehmend Techniken zur Beschleunigung verbaut.

Zunächst waren die ersten Prozessoren multicycle-basiert, so dass eine Instruktion über mehrere Zeitschritte hinweg abgearbeitet wurde. Um das gesamte Programm zu beschleunigen, wurden Pipeline-Techniken in den Prozessor eingebaut, die die einzelne Instruktion in kleine Unteraufgaben aufbrechen, die immer wieder auftreten, so dass einzelne Instruktionen im Vergleich zu einem Multicycle-Prozessor auch langsamer sein können. Da aber je nach Pipeline-Tiefe entsprechend mehr Instruktionen gleichzeitig überlappend bearbeitet werden können, liegt die Gesamtperformance des Programms weitaus höher als bei einem vergleichbaren Mehrzyklenprozessor. Typische Einzelschritte in einer RISC-basierten Pipeline sind:

- Laden der nächsten Instruktion (engl.: *Instruction Fetch*, IF),
- Dekodieren der neuen Instruktion (engl.: *Instruction Decode*, ID),
- Ausführung der Instruktion (engl.: *Execute Instruction*, EX),
- Schreiben in den Speicher/IO (engl.: *Memory-Phase*, MEM)
- Geänderte Werte zurück in die Register schreiben (engl.: Write-Back, WB).

Üblicherweise wird die Instruction Set Architecture unabhängig von der Implementierung definiert. Sie beschreibt lediglich das funktionale Verhalten der einzelnen Instruktio-

nen, und ihre Semantik ist die einer streng sequenziellen Abarbeitung des Programms. Sequenzielle Multicycle-Implementierungen und Pipeline-Implementierungen haben daher unterschiedliches Zeitverhalten. Das Verhalten bezüglich der programmsichtbaren Register hingegen ist bei beiden Implementierungen gleich.

2.3.2 Interrupts und Kontextwechsel

Aus Sicht der ISA führt ein Prozessor sequenziell eine Instruktion nach der anderen aus solange bis das System wieder abgeschaltet wird. Werden in dieser Zeit Daten von der Peripherie bereitgestellt, bekommt das aktuell laufende Programm dies nur mit, wenn es aktiv an dem entsprechenden Port der Peripherie zugreift – also aktiv auf neue Daten wartet. Der Zugriff erfolgt via einer LOAD-Instruktion, die regelmäßig ausgeführt werden muss und lesend auf die zum Port zugehörige Adresse zugreift. Je nach Komplexität wird zusätzliche Programmlogik verwendet, um neue Daten zu erkennen. Dieses aktive Warten wird Polling genannt und blockiert Rechenleistung die für andere Programme genutzt werden könnte. Um dies potentiell zu vermeiden, werden von allen gängigen Prozessoren sogenannte Interrupts unterstützt. Ein Interrupt ermöglicht es, den regulären Ablauf des aktuellen Programms im Prozessor von außen zu unterbrechen, damit dringende Aufgaben für die Peripherie bevorzugt behandelt werden können. Programme brauchen dadurch nicht mehr aktiv zu warten und Polling wird dadurch obsolet. Im Kapitel 2.4.3 wird der Interrupt-Mechanismus im Zusammenspiel mit der Software ausführlich beschrieben. Zur Unterstützung von Interrupts wird die Hardware eines Prozessors um eine Komponente erweitert, die vor jeder Ausführung einer Instruktion überprüft, ob ein Interrupt-Request (IRQ) vorliegt. Diese werden von dem Prozessor mittels separater Signaleingänge, den sogenannten Interrupt-Lines, empfangen. Je nach Komplexität des Interrupt-Mechanismus wird nicht nur überprüft, ob ein Interrupt-Request aufgetreten ist, sondern auch mittels einer gegebenen Priorität entschieden, ob er tatsächlich ausgeführt wird. Die Priorität kann, je nach Prozessor, entweder statisch für die jeweilige Interrupt-Line vergeben oder bei komplexeren Prozessorarchitekturen beliebig konfigurierbar sein. Einen Interrupt-Request kann nur ein anderes Hardware-Device erzeugen, welches am Systembus des Prozessors angebunden ist.

Durch die Einführung des Interrupts ist es möglich das aktuell ausgeführte Programme zu unterbrechen. Um nach der Interrupt-Behandlung das unterbrochene Programm fortsetzen zu können, muss der aktuelle Kontext des Programms gespeichert werden. Die Hardware des Prozessors muss dieses Konzept unterstützen, da dies nicht ausschließlich in Software durchgeführt werden kann. Essentiell ist dabei die Sicherung des Programmzählers, der immer auf die aktuelle Stelle im Programm zeigt. Somit kann die Software nur die Position des PCs im aktuell ausgeführten Programm abspeichern, aber nie die Position des Programms, welches vorher ausgeführt wurde.

Prozessoren verfügen über zusätzliche Hardware, welche die Rücksprungadresse in ein Register sichert. Dabei ist die Rücksprungadresse in der Regel die Adresse, die in den PC geladen werden muss, um die nächste auszuführende Instruktion des unterbrochenen Programmes zu laden. Alles Weitere für den vollständigen Kontextwechsel, wie

das Speichern der Register, kann auch mittels Software durchgeführt werden und wird in Kapitel 2.4.3 und Kapitel 4.5 ausführlicher beschrieben.

Es gibt auch Architekturen, die diesen Kontextwechsel komplett in Hardware durchführen können. Diese besitzen dann nicht nur einen Satz von Registern, der bei jedem Kontextwechsel gesichert werden muss, sondern mehrere Sätze von Registern, wobei für eine Applikation nur ein Satz sichtbar ist. Um den Kontextwechsel durchzuführen wird nur von einem Register Satz auf den anderen gewechselt.

Konkrete Prozessor Architekturen

Auf dem Markt der Prozessoren gibt es diverse unterschiedliche Architekturen. Während der Markt der Desktop PCs überwiegend von der x86-basierten Intel-Architektur und wenigen Herstellern dominiert wird, ist im Bereich der eingebetteten Systeme die Angebotsvielfalt sowohl der Architekturen als auch der Hersteller wesentlich größer. Eine gute Übersicht gängiger eingebetteter Prozessoren kann in [Tec13] gefunden werden. Durch die starke Verbreitung von Smartphones ist mittlerweile auch die ARM-Architektur weit verbreitet und der Allgemeinheit bekannt. Zusätzlich hat sich die ARM-Architektur auch in anderen eingebetteten Systemen verbreitet. Im Rahmen der Arbeit wurde der Aquarius [Ait03] verwendet, der auf der SuperH2-Architektur der Firma Renesas (ursprünglich Hitachi) basiert.

2.3.3 Kommunikationsbusse

Ein Bus ist ein Leitungsbündel, welches mittels eines standardisierten Protokolls die Kommunikation zwischen verschiedenen Hardwarekomponenten ermöglicht. Das Protokoll definiert, in welchem Format Daten übertragen werden und wie der Empfänger einer Nachricht bestimmt wird.

Ein Prozessor, wie in Kapitel 2.3.1 beschrieben, kommuniziert in der Regel ausschließlich über seinen eigenen Systembus. Dort werden alle Komponenten, auf die der Prozessor zugreifen muss, direkt oder indirekt (d.h., über *Busbrücken*) angeschlossen. Dazu gehören vor allem der Arbeitsspeicher und die Peripheriegeräte. Neben dem Systembus gibt es noch diverse andere Bussysteme. Diese Bussysteme werden primär daran unterschieden, ob sie Daten seriell oder parallel übertragen. Parallele Busse, wie der Systembus, verfügen über viele Leitungen über die mehrere Bits gleichzeitig transportiert werden können. Sie verfügen oft über zusätzliche Leitungen, auf denen die Adresse des Empfängers angelegt wird und weitere Steuersignale, die parallel zu den Daten transportiert werden. Serielle Busse hingegen verfügen in der Regel nur über eine Leitung bzw. ein paar wenige Leitungen. Die Daten werden dabei nacheinander über eine Leitung übertragen. Die weiteren Leitungen dienen je nach Implementierung als Steuerleitungen. Ein weiteres wichtiges Unterscheidungsmerkmal der Busse ist, wie die Nachricht auf dem Bus synchronisiert wird. Synchrone Busse besitzen einen eigenen Takt, der für jeden Teilnehmer des Busses zur Verfügung steht. Der Takt wird entweder über eine direkte Taktleitung im Bus verteilt oder jeder Teilnehmer am Bus besitzt seinen eigenen Takt

der über den Bus synchronisiert werden kann, um eine reibungslose Kommunikation zu ermöglichen. Asynchrone Busse hingegen besitzen keinen Takt, sondern müssen zusätzliche Synchronisationsmethoden im Kommunikationsprotokoll anbieten. Bedingt durch ihre Anwendungsgebiete gibt es eine Unmenge von unterschiedlichen Bussen, die sich an den genannten Kriterien zwar kategorisieren lassen, sich dennoch in vielen Details unterscheiden. Der Vorteil der Busse ist, dass sie wohl definierte Schnittstellen zwischen unterschiedlichen Hardwarekomponenten zur Verfügung stellen. Es ermöglicht eine einfache und abstrakte Betrachtung der Schnittstelle und dient somit zum Beispiel in Kapitel 7.4 zur Erstellung der zu prüfenden Eigenschaften.

Prozessor-nahe Busse werden in [PH07] und [HP07] beschrieben. Andere Busse wie z.B. Feldbusse sind in unterschiedlichen Normen beschrieben. Zum Beispiel der in Kapitel 7.4 verwendete Automotive Bus wird in [LIN03] standardisiert.

2.3.4 Hardware Modellierung

Die real implementierte Hardware besteht aus Transistoren, Widerständen und weiteren Komponenten. Zur Betrachtung des Verhaltens der Hardware, ist diese Ebene sehr detailreich und komplex. Deswegen gibt es die Möglichkeit die Hardware auf unterschiedlichen Abstraktionsebenen zu modellieren. Dies erlaubt eine einfachere Beschreibung des funktionales Verhaltens der Hardware.

Zum Beschreiben des funktionalen Verhaltens bietet sich mindestens eine Ebene an, die das logische Verhalten auf Gatterebene beschreibt. Weit verbreitet ist die Register-Transfer-Ebene (engl.: *register transfer level*, RTL). Sie ist eine höhere Abstraktions-Ebene, die einfache Darstellungen komplexere Operationen unterstützt. Die RTL-Ebene ist die meist verwendete Abstraktion beim digitalen Hardwareentwurf und wird von den gängigen Tools für die Synthese und Verifikation unterstützt. Gängige Hardwarebeschreibungssprachen sind zurzeit VHDL [Ash01] oder Verilog [20009]. Es gibt auch noch Sprachen mit einer höheren Abstraktionsebene, wie zum Beispiel SystemC [20111].Modelle auf Abstraktionsebenen oberhalb von RTL dienen vornehmlich der Architekturexploration, der Validierung des Systementwurfs und der frühen Softwareentwicklung (auf sog. “virtuellen Prototypen”).

2.4 Software

Mit Software bezeichnet man Computerprogramme aller Art. Die Software, die von einem Prozessor ausgeführt werden kann, ist *Maschinencode*, also die binär kodierte Abfolge von Maschineninstruktionen eines Programmes. Da Maschinencode für Benutzer schlecht lesbar ist, und somit das Schreiben solcher Codes sehr fehleranfällig ist, wurde die Assemblersprache entwickelt, die überwiegend eine 1:1-Abbildung des Maschinencodes ist [Häb11]. Gängige Assembler unterstützen auch verschiedene Pseudoinstruktionen, die es dem Programmierer ermöglichen, Sprünge mittels Sprungmarken (engl. *labels*) zu realisieren oder auch Adressen bei Lese- und Schreiboperationen mittels Labels darzu-

stellen. Der Assembler, der das Assemblerprogramm in ein Maschinenprogramm übersetzt, muss diese Pseudoinstruktionen durch die korrekten Operationen ersetzen und die verwendeten Labels durch die korrekten Sprungweiten oder Adressen ersetzen.

Assembler ist durch die Hardwarenähe sehr prozessorspezifisch und bietet wenig Abstraktionen für wiederkehrende Aufgaben an. Deswegen sind Programmiersprachen entwickelt worden, die einen viel höheren Abstraktionsgrad haben, wie z.B. C oder die objektorientierten Sprachen wie z.B. C++ oder Java. Programme, die in einer Hochsprache geschrieben werden, müssen mittels eines Compilers in Maschinencode übersetzt werden. In der Regel unterstützen Compiler unterschiedliche Prozessorarchitekturen, so dass die Hochsprachen überwiegend plattformunabhängig sind.

Bei Java gibt es zusätzlich die Möglichkeit, den Code nicht für einen konkreten Prozessor zu kompilieren; er kann auch für eine virtuelle Maschine übersetzt werden. Diese virtuelle Maschine ist eine Software, die gegebene Instruktionen während der Ausführung in die entsprechenden Maschineninstruktionen des real verwendeten Prozessors übersetzt. So wird die Java-Software plattformunabhängig, wobei für jede Plattform eine entsprechende virtuelle Maschine implementiert werden muss.

Eine weitere Alternative sind Interpreter, die den gegebenen Code ähnlich wie virtuelle Maschinen bei Java interpretieren und dann entsprechend umsetzen. Diese Interpreter selbst sind plattformabhängig, aber der zu interpretierende Code dagegen nicht. Gängige Beispiele dafür sind Python-Skripte oder auch Webseiten in HTML.

Mit der Weiterentwicklung der Programmiersprachen und auch der Möglichkeiten von Interpretern und Applikationen ist die Bedeutung des Wortes Software ebenfalls mitgewachsen und umfasst nahezu alle Programme, die in irgendeiner Form auf einem Medium gespeichert sind, also auch Quellprogramme (engl. *Sourcen*), Skripte und dergleichen.

2.4.1 Hardwarenahe Software

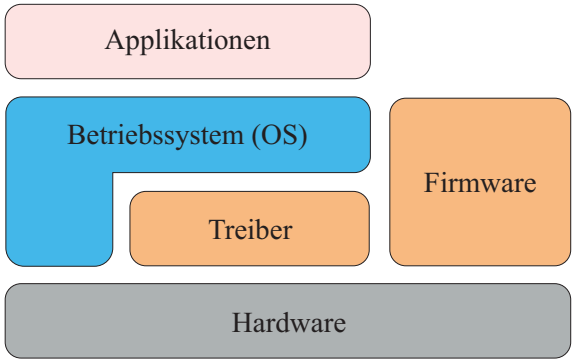

Abbildung 2.4: Softwareschichten

Unter dem Begriff “hardwarenahe Software” werden die Codefragmente und Programme zusammengefasst, die für die Kommunikation mit der Peripherie und anderen Hardwarekomponenten zuständig sind. Sie spielen in allen eingebetteten Systemen und

auch in Computern eine wesentliche Rolle, beispielsweise wären ohne die Interaktion mit der Umgebung die berechneten Ergebnisse nicht sichtbar.

Ein Arbeitsplatzrechner, an dem der Anwender einen Text schreibt, benötigt hardwarenahe Software für die Eingabe des Textes über die Tastatur, für die Verwendung der Maus und ebenfalls für die Ausgabe auf dem Bildschirm. Diese Funktionen werden von Treibern und Teilen des Betriebssystems zentral ermöglicht und gesteuert.

Dies ermöglicht es der Applikationssoftware, über definierte, in Software implementierte Schnittstellen, die das Betriebssystem zur Verfügung stellt, auf die Hardware zuzugreifen. Abbildung 2.4 zeigt, welche Schichten ein solches Softwaresystem besitzt. Dabei verfügt das Betriebssystem nicht nur über eine Schnittstelle für die Applikationen, sondern auch eine definierte Schnittstelle zur Seite der Treiber. Somit sind Treiber und auch Teilkomponenten des Betriebssystems hardwarenahe Software. Zusätzlich gibt es in vielen Computersystemen noch Firmware. Firmware ist Software die fest (engl. *firm*) in einem eingebetteten System integriert ist. Sie ist speziell an die Hardware angepasst, nicht ohne weiteres auf andere Hardware portierbar und für die korrekte Funktion des eingebetteten Systems notwendig. Dabei kann die Komplexität der Firmware von wenigen Zeilen Code – als fest programmiertes Programm eines Miniprozessors – bis hin zu einem vollständigen Betriebssystem mit diversen Programmen reichen.

Das BIOS (engl.: *basic input/output system*) oder UEFI (engl.: *Unified Extensible Firmware Interface*) im Personal Computer ist ein Beispiel für eine Firmware. Weitere Beispiele sind auch die Software eines digitalen Signalprozessors, der zur Signalverarbeitung in den UMTS-Empfänger eines Smartphones eingebaut ist oder in einem mp3-Decoder-Chip.

Software eingebetteter Systeme ist ebenfalls aufgebaut wie in Abb. 2.4. Ihre Applikationen sind wesentlich stärker auf die umgebende Hardware bezogen, auch wenn sie nicht direkt drauf zugreifen. Treiber und die hardwarenahen Softwarekomponenten des Betriebssystems spielen im eingebetteten System eine noch wichtigere Rolle und beschreiben einen großen Teil des Gesamtverhaltens. Ein Beispiel eines eingebetteten Systems aus dem Automotive-Bereich ist die Motorsteuerung im Auto. Sie steuert das komplette Verhalten des Motors und ist für den Fahrer nicht direkt zugreifbar. Er hat zwar die Möglichkeit, Gas zu geben, aber das tatsächliche Verhalten hängt zusätzlich auch von der Luftfeuchtigkeit, der Lufttemperatur, der aktuellen Umdrehung, der Wärme in den Zylindern und weiteren Faktoren ab. Die Motorsteuerung verfügt über viele verschiedene Sensoren und Aktoren, um den Zustand des Motors zu erkennen und entsprechend anzupassen. Hardwarenahe Software ist für den Anwender nicht sichtbar, kann sich aber dennoch bei einem Fehler auf die Aufgaben des Gesamtsystems auswirken und dabei potentiell nicht nur das System gefährden, sondern auch den Benutzer.

2.4.2 Software-Modellierung

Software kann, wie gerade beschrieben, mittels unterschiedlich hoher Programmiersprachen geschrieben werden. Die einzige Software, die tatsächlich auf dem Prozessor ausgeführt wird, ist der Maschinencode. Somit dienen für den Entwickler die meisten Pro-

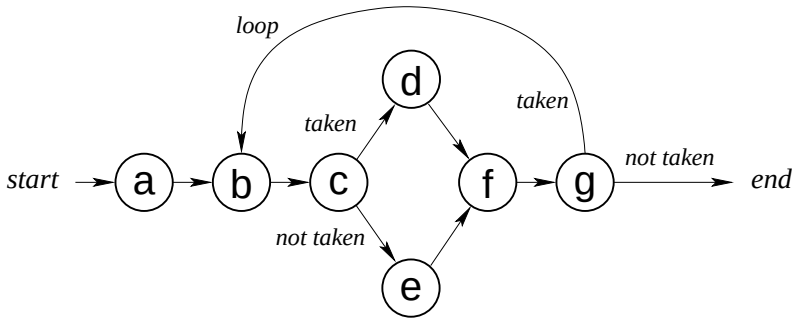

Abbildung 2.5: Beispiel CFG – Graphische Darstellung eines Programms

grammiersprachen als Abstraktionshilfen, so dass er sich auf die Aspekte des Programmes konzentrieren kann.

Die meisten Programmiersprachen unterstützen sequenzielle Beschreibungsformen, so dass der Entwickler seinen Algorithmus auf ein sequenzielles Verhalten abbilden muss. Das sequenzielle Verhalten ist zum menschlichen Denken und Verhalten sehr ähnlich und daher haben die imperativen Programmiersprachen eine hohe Verbreitung erlangt. Neben den Programmiersprachen können auch Graphen genutzt werden um sequenzielle Software zu beschreiben.

Ein solcher Graph besteht aus Knoten, die jeweils eine lineare Sequenz von Programmschritten darstellen (sog. *Basisblock*).Die (gerichteten) Kanten zwischen den Knoten stellen mögliche Verzweigungen zwischen den Basisblöcken dar. Ein Basisblock kann eine einzelne Instruktion oder eine beliebig lange Folge sequenzieller Instruktionen enthalten. Die Folge selbst darf nicht verzweigen. Sprünge von anderen Stellen des Programms sind nur an den Anfang der Sequenz, nicht aber in die Sequenz erlaubt. Dieser Graph wird Kontrollflussgraph (engl.: *Control flow graph, CFG*) genannt, da er den Ablauf des Programms wiedergibt und keine Datenabhängigkeiten oder andere Eigenschaften abgebildet werden.

Abbildung 2.5 zeigt ein Beispiel eines solchen Kontrollflussgraphen für ein Programm, welches neben Verzweigungen auch eine Schleife enthält.

Durch die Zunahme an komplexen Programmen haben sich Programmiersprachen etabliert, die neben dem sequenziellen Verhalten komplexe Datenstrukturen und ihre Beziehungen abbilden können. Dazu gehört vor allem die Klasse der objektorientierten Sprachen, die es erlauben eine Struktur zu beschreiben. Sie erlauben einfachere Hierarchien, Abhängigkeiten zwischen verschiedenen Klassen und deren Instanzen zu definieren und ermöglichen Vererbung und Interfaces [Str00]. Diese Modellierung erlaubt es dem Entwickler in der Programmiersprache eine abstrakte Sichtweise auf Objekte zu erzeugen, die je nach Anwendung unterschiedlich sein kann. Neben den Sprachen wurden auch graphische Konzepte zur Modellierung objektorientierten und strukturellem Verhaltens entwickelt. UML und SDL sind zwei etablierte Beispiele für solche Konzepte.

2.4.3 Interrupt-getriebene Software

Software, die mit der Umgebung interagiert, greift dabei über einen Systembus auf die Hardware zu. Immer wenn Daten ausgegeben werden, kann die Software diese Daten einfach auf dem Bus an das entsprechende Device senden. Werden dagegen bestimmte Daten von einer Hardware-Komponente erwartet, kann die Software zwar die Daten auslesen, es ist aber im Vorfeld nicht bekannt, wann die neuen Daten anliegen werden. Daher muss die Software so lange das Register vom entsprechenden Device überprüfen (Polling), bis der gewünschte Wert tatsächlich vorliegt. In dieser Zeit muss der zugehörige Prozess auf dem Prozessor ausgeführt werden, und es kann kein anderer Prozess ausgeführt werden. Das heißt, das Programm wartet aktiv auf den auszulesenden Wert.

Als Alternative gibt es passives Warten, währenddessen der Prozess pausiert, solange die erwarteten Daten noch nicht vorliegen. Sobald die gewünschten Daten vorliegen, wird der entsprechende Prozess wieder gestartet und weiter ausgeführt. Die Realisierung wird auch mittels der Hardwareunterstützung des Interrupts ermöglicht. Es kann nicht nur auf externe Daten gewartet werden, sondern ebenfalls eine Priorisierung der Daten erfolgen.

So kann ein externes Device, sobald es über neue Daten verfügt, den Prozessor mittels eines Interrupt-Requests auf sich aufmerksam machen. Je nach Priorität wird der Interrupt-Request akzeptiert, oder er muss warten, bis ein eventuell höherwertiger Interrupt abgearbeitet ist.

Nachdem der Interrupt-Request akzeptiert wurde, wird das aktuelle Programm unterbrochen, und eine, dem Interrupt zugehörige Interrupt-Service-Routine (ISR) wird gestartet. Diese ist, sofern nicht hardwareseitig vom Prozessor übernommen, für das Sichern des Architekturstatus des Prozesses zuständig. Da der Programmzähler bereits modifiziert werden muss, um den Interrupt-Handler zu starten, muss der alte PC in jedem Fall von der Hardware selbst gespeichert werden. Die von der CPU gespeicherte Rücksprungadresse gehört ebenfalls zum Architekturzustand. Nachdem die Daten des zuvor unterbrochenen Prozesses gespeichert sind, beginnt die eigentliche Interrupt-Behandlung. Die Routine lädt die Daten vom externen Device und beginnt ggf. mit einer Verarbeitung der Daten. Zum Abschluss werden die verarbeiteten Daten in den Speicher geschrieben, in dem die Prozesse auf die benötigten Daten zugreifen können.

Je nach der Komplexität des Systems kann die ISR dann dem Scheduler signalisieren, dass der wartende Prozess wieder aktiviert werden kann, da die Daten im Speicher bereitliegen.

Der Interrupt wird nicht ausschließlich zur Kommunikation verwendet. Auch das präemptive Multitasking wird mit Hilfe eines Interrupts von einem externen, konfigurierbaren Timer realisiert. Dieser löst in festen Intervallen Interrupts aus, die anstelle eines klassischen ISR den Scheduler starten.

Typische Interrupt-getriebene Programme bzw. Routinen sind Betriebssystem-Komponenten wie der Scheduler und Treiber, die externe Eingaben von der Hardware erwarten.

2.5 Nicht-formale Verifikationstechniken (NFV)

In den beiden folgenden Teilkapiteln werden die unterschiedlichen Verifikationstechniken zur Qualitätssicherung der entworfenen Systeme vorgestellt. Der Schwerpunkt in diesem Kapitel liegt auf den nicht-formalen Techniken, während im anschließenden Kapitel 2.6 die formalen Ansätze behandelt werden.

> Software-Qualitätssicherung und Hardware-Qualitätssicherung besitzen zum Teil Gemeinsamkeiten. Sie entwickeln sich aber seit mehreren Jahrzehnten relativ getrennt zueinander. [Lig02, S. 27].

Entsprechend dieser getrennten Entwicklung bei der Software- und der Hardwareverifikation unterscheidet sich die Betrachtungsweise der zu lösenden Aufgaben teilweise erheblich. Daher werden hier die vorgestellten nicht-formalen Verifikationsansätze (NFV) für die jeweilige Anwendungsdomäne betrachtet. Die in [Lig02, S. 27] erwähnten Gemeinsamkeiten liegen überwiegend in den zugrundeliegenden Verifikationstechniken, die als Backend in den beiden Feldern verwendet werden.

Zusätzlich lassen sich die Methoden nach Black-Box- und White-Box-Verfahren unterscheiden. Black-Box bedeutet, dass das zu testende System nur von außen beobachtet wird. Es werden Signale an die Eingänge angelegt und die Antwort an den Ausgängen abgewartet. Interne Zustände können so nicht überprüft werden. White-Box dagegen bedeutet, dass auch Signale und Werte innerhalb des Systems betrachtet werden können bzw. müssen. Der Vorteil des White-Box-Verfahrens ist im Allgemeinen, dass innere Zustände besser erkundet werden können. Daher kann die Aussagekraft der Ergebnisse besser den Teilkomponenten im Design zugeordnet werden. Typischerweise ist eine Verifikation nach dem White-Box-Prinzip aufwändiger, da der Verifikationsingenieur sich mit der konkreten Implementierung auseinandersetzen muss.

Da in dieser Arbeit ein Modell entwickelt werden soll, das eine gemeinsame Betrachtung von Hardware und Software zum Ziel hat, werden kombinierte Ansätze nochmals zusätzlich aufgegriffen. Diese Ansätze stammen entweder aus der Hardware- oder aus der Softwaredomäne und wurden erweitert, um die Aspekte aus beiden Domänen betrachten zu können.

Zu den nicht-formalen Techniken gehören vor allem dynamische Ansätze, zu denen auch das klassische Testen und Simulieren zählen. Diese Ansätze kommen sowohl in der Hardware- als auch in der Software-Verifikation vor.

2.5.1 NFV für Hardware

Als nicht-formale Techniken gelten vor allem simulative Ansätze oder auch Prototypen, die mittels Tests untersucht werden. [HT10, Seite 99ff.]

Simulation

Die Simulation gehört in der Industrie zu den am weitesten verbreiteten Verifikationstechniken, um Hardware zu verifizieren.

Zuerst wird ein Simulationsmodell erstellt, welches anschließend unter dem Anlegen von gewissen Eingangsbelegungen, sogenannter Stimuli, ausgeführt wird. Die dabei erzeugte Ausgabe wird mit der zu erwartenden Ausgabe verglichen, um zu entscheiden, ob der Test korrekt ist. Eine solche Ausgabe ist ein *Simulationstrace*, mit den Werten der betrachteten Signale über die Zeit. Die betrachtete Simulationszeit kann je nach verwendetem Simulator entweder – für getaktete Systeme – zyklengenau oder eventbasiert dargestellt werden. Eventbasierte Simulation wird im Falle von analogen oder asynchronen Komponenten verwendet. Um eine Simulation durchzuführen, muss das Original-Design in ein Simulationsmodell übersetzt werden. Dieses sogenannte "System under verification" (SUV) erhält seine Eingabewerte aus einer simulierten Umgebung, einer Testbench. Abbildung 2.6 zeigt den Aufbau einer Simulation. Für unterschiedliche Stimuli muss das unterschiedliche Verhalten in den Testbenches beschrieben oder jeweils für jeden Simulationstrace eine neue Testbench geschrieben werden.

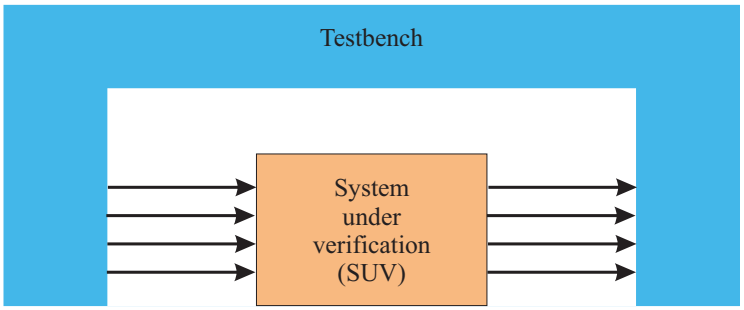

Abbildung 2.6: Aufbau einer Simulationsumgebung [HT10]

Ein einzelner Simulationstrace überprüft das Verhalten des Designs für eine bestimmte Eingabe. Diese Information nachzuvollziehen ist recht einfach, da nur ein bestimmter Fall überprüft wird. Der grundsätzliche Nachteil simulationsbasierter Verifikation besteht darin, dass nur die Szenarien verifiziert werden, die durch die Testbenches beschrieben werden. Simulation kann also möglichlerweise Szenarien "übersehen", in denen sich Entwurfsfehler auswirken. Typische Simulationstraces sind *Compliance-Tests* wie zum Beispiel *Conformance-Tests* für Busprotokolle. In solchen Tests sind eine Reihe von Fällen beschrieben, die eine Implementierung erfüllen muss, um als kompatibel zu dem gegebenen Protokoll oder Standard zu gelten. Neben dem korrekten Verhalten wird in diesen Tests auch gezielt erwartetes Fehlverhalten der Umgebung getestet. Unerwartete Fälle werden im Allgemeinen dennoch übersehen. Zusätzlich zur Simulation erwarteten Verhaltens, können auch *Random-Pattern-Tests* verwendet werden, bei denen willkürliche Eingangsbelegungen gewählt werden. Hier besteht die Herausforderung darin den Zustandsraum möglichst abzudecken. Eine erschöpfende Simulation ist bei großen Designs mit vielen internen Speichern meist unmöglich in akzeptabler Zeit. Das liegt auch an der Schwierigkeit, das Design in jeden gewünschten Zustand zu bringen, um das Verhalten an dieser Stelle zu testen.

Je nach Komplexität des zu verifizierenden Designs ist die Simulation schneller oder langsamer; wenige Sekunden Echtzeit können möglicherweise nur in wenigen Minuten

oder mehreren Tagen simuliert werden.

Es gibt verschiedene Ansätze, ein Maß dafür zu finden, "wieviel" der Funktionalität eines Designs durch einen Test abgedeckt wird. Als Referenz dient entweder die Spezifikation oder die implementierte Hardware in Form des HDL-Codes. Im Falle der Spezifikation sollte für jede Aussage in der Spezifikation ein oder mehrere Simulationstimuli in der Testbench geschrieben und am DUV simuliert werden. Die Ergebnisse werden mit den Erwartungen der Spezifikation verglichen. Werden alle Elemente der Spezifikation abgedeckt, so wird die Verifikation als vollständig betrachtet.

Da es je nach Komplexität oder wegen unpräziser Aussagen in der Spezifikation nicht möglich ist, jede Aussage in Form von einigen Simulationstraces zu verifizieren, kann es unmöglich sein ein System vollständig zu verifizieren. In der Regel beschreibt eine Spezifikation das Verhalten nicht vollständig und lässt dem Entwickler Freiheiten bei der Implementierung. Diese Spezifikationslücken werden im Allgemeinen bei einem Vollständigkeitsansatz auf Basis der Spezifikation nicht erkannt. [Bor09]

Wird anstelle der Spezifikation der HDL-Code zur Vollständigkeitsbetrachtung herangezogen, so muss zumindest für jeden Zweig bzw. jede Zeile des Codes eine Eigenschaft geschrieben werden, die diese abdeckt. Dabei kann eine einzelne Simulation viele einzelne Codezeilen bzw. Codeblöcke abdecken. Es ist allerdings nicht zwingend, dass alle möglichen Wertebelegungen abgeprüft wurden. Mit diesem Ansatz können Lücken in der Spezifikation erkannt werden, da beim Erstellen der Testbenches auffällt, dass Verhalten beschrieben werden muss, welches von der Spezifikation nicht abgedeckt ist.

Rapid Prototyping

Ein Ansatz, der im Allgemeinen in Echtzeit oder zumindest nahe an der Echtzeit ausgeführt werden kann, ist ein real implementierter Prototyp. In solchen Fällen wird nicht direkt die finale Hardware implementiert, sondern ein Prototyp, der einfach zu realisieren ist. Ein FPGA-basierter Ansatz ermöglicht es, die später zu verwendende Hardware in kurzer Zeit von der HDL-Sprache (wie z.B. VHDL) in eine real nutzbare Hardware zu übersetzen. Das hat den Vorteil, dass der so beschriebene FPGA – sofern die finale Umgebung schon vorhanden ist – direkt in seine Zielumgebung eingesetzt und im Zusammenspiel getestet werden kann. Diese FPGA-basierten Implementierungen arbeiten nahezu in Echtzeit und können daher wesentlich mehr Einzeltests in kurzer Zeit durchlaufen. Auch können, im Vergleich zur Simulation, längere Laufzeiten des Systems getestet werden.

Wenn die Umgebungs-Hardware noch nicht vorhanden ist, gibt es sogenannte *Hardware-in-the-Loop-Systeme* (*HIL-Systeme*), die die implementierte Hardware in einer durch komplexe Programme simulierten Umgebung testen. Die simulierte Umgebung erzeugt dabei die Signale, die die zu testende Hardware erwartet.

Sowohl beim Rapid Prototyping als bei HIL-Systemen wird die implementierte Hardware als Black-Box angenommen und nur anhand der Ausgaben bewertet.

2.5.2 NFV für Software

Die Verifikation von Software ist ein intensiv erforschtes Gebiet der Informatik und wird schon seit mehreren Jahrzehnten weiterentwickelt. Dabei haben sich parallel zur Weiterentwicklung von Software auch die Anforderungen an die Verifikation verändert. Ursprünglich sind nur sequenzielle Programme auf einer Single-Core-Architektur ausgeführt worden. Seitdem ist die Softwarewelt jedoch wesentlich komplexer geworden. Die Einführung von Multitasking, Interrupt-Mechanismen und Multicore-Architekturen ermöglichen völlig neue Freiheitsgrade in der Softwareentwicklung, wodurch die Wartbarkeit und die Verifikation von Software komplizierter wird. Die echte parallele Programmierung erfordert ebenfalls eine zunehmende Kommunikation zwischen den einzelnen Softwarekomponenten und Synchronisation der Daten und Programme. Beide – Kommunikation und Synchronisation – können einen Einfluss auf das Verhalten der Software haben. Entsprechend steigen dazu die Anforderungen an die Verifikation.

Eine gute Übersicht über die nicht-formalen Verifikationstechniken für Software ist in [Lig02] gegeben; sie werden überwiegend als Testverfahren eingeführt. Dabei wird zwischen statischen und dynamischen Tests unterschieden. Bei statischen Tests wird kein Code ausgeführt, sondern nur mit statischen Verfahren analysiert. Bei den dynamischen Tests wird der Code mit gewissen Wertebelegungen ausgeführt. Wie bei der Hardwareverifikation ist bei der Softwareverifikation eins der bekanntesten dynamischen Verfahren die Simulation.

Dynamisches Testen

Um ein zu verifizierendes Programm zu simulieren, muss das Programm mit Eingaben stimuliert und die Ausgabe entsprechend überprüft werden. Das geschieht entweder durch den Vergleich mit einem Referenzmodell, das die korrekten Ausgaben enthält, oder mit Hilfe von Monitoren, die bestimmte Eigenschaften der Software und der Daten kontinuierlich überprüfen und gegebenenfalls einen Fehler anzeigen. Die Simulation des Programms erfolgt entgegen der Hardwareverifikation nicht taktgesteuert, sondern simuliert einen Pfad sequenziell – Instruktion für Instruktion. Die Parallelität, wie sie in der Hardwareverifikation berücksichtigt werden muss, ist bei den meisten sequenziellen Programmen nicht gegeben. Der Pfad durch die Software wird durch die Eingabe implizit vorgegeben.

Eine andere Form des Testens ist das *Rapid Prototyping*. In diesem Fall soll eine Software für ein noch nicht real existierendes System getestet werden. Mittels Software wird ein System implementiert, das das reale Verhalten der Hardware modelliert, so dass die Software ausgeführt werden kann. Ein solches System kann zum Beispiel ein “Instruction Set Simulator” sein, der den Maschinencode ausführen soll. Es gibt auch die Möglichkeit, die Ausführung von Programmen in höheren Programmiersprachen durch Interpretation direkt zu simulieren und dabei zu testen.

Beide Verfahren sind in ihrer Anzahl an Simulationen bzw. Ausführungen limitiert und ermöglichen in der Regel keine erschöpfende Simulation und somit vollständige Verifikation. Ohne eine erschöpfende Simulation können Fehler gefunden werden, aber die

Fehlerfreiheit kann nicht nachgewiesen werden.

Statisches Testen

Statt den Code zu simulieren, kann er auch mit Hilfe von statischen Mitteln analysiert werden. Bei solchen Verfahren liegt der Fokus weniger auf den Datenberechnungen als auf generellen Implementierungskonzepten. In der Regel wird geprüft, ob die Variablen korrekt initialisiert wurden, ob geschriebene Variablen auch wiederverwendet werden, ob Schleifenabbruchbedingungen erfüllt werden können, oder ob bestimmte Verzweigungen tatsächlich eintreten können. Diese Verfahren können direkt auf den Code angewendet werden, ohne ein zusätzliches Modell zu erzeugen. Zur Analyse derselben Fehler können auch formale Techniken wie zum Beispiel abstrakte Interpretation (vgl. Kapitel 2.6.2) verwendet werden. Der Unterschied liegt jedoch in der Aussagekraft der Verfahren. Während beim statischen Testen potenzielle Fehler gefunden werden können, können die formalen Techniken darüber hinaus auch die Abwesenheit jeglicher Fehler beweisen.

2.6 Formale Verifikationstechniken

Im Gegensatz zu den nicht-formalen Verifikationstechniken sind die formalen Methoden (FV) im Allgemeinen vollständig [HT10, Seite 99]. D.h., wenn der Beweis einer Eigenschaft gelingt, gilt das für alle möglichen Zustände des Modells und nicht nur für einen bestimmten gewählten Pfad bzw. für eine bestimmte Ausführung. Außerdem ist es mit der formalen Verifikation möglich, die Abwesenheit eines Fehlers zu beweisen. Das hat den Vorteil, dass die Aussagekraft eines solchen Verfahrens wesentlich höher ist, verglichen mit herkömmlichen Simulationen. Dafür sind diese Methoden wesentlich aufwendiger. Das bedeutet vor allem, dass die meisten Verfahren bei einer linearen Zunahme der zu prüfenden Design-Komplexität ein exponentielles Wachstum in Kennzahlen wie Speicherverbrauch oder Zeitbedarf aufweisen. Ein weiterer Nachteil der meisten formalen Methoden ist, dass eine zeitliche Aussage, wie lange der Beweis dauern wird bzw. wieweit das Beweisverfahren mit der Berechnung schon fortgeschritten ist, in der Regel unmöglich ist. In den beiden folgenden Kapiteln wird zwischen Hardware- und Softwareverifikation unterschieden. Dabei sind die verwendeten Beweistechniken überwiegend dieselben. Doch gerade bei der Erstellung des Berechnungsmodells für die Verifikation unterscheiden sich die einzelnen Ansätze erheblich.

2.6.1 FV für Hardware

Bei den formalen Techniken haben sich in der Anwendung überwiegend die modellbasierten Ansätze durchgesetzt. In diesen Verfahren wird ein formales Modell generiert, das eine Repräsentation des später tatsächlich zu implementierenden Designs ist. Auf diesem Modell werden dann die zu verifizierenden Eigenschaften überprüft.

Modellprüfung

Modellprüfung (engl.: *Model Checking*, MC) ist ein modellbasierter Ansatz [CGP99]. Das zu betrachtete System und die zu prüfenden Bedingungen werden in ein mathematisches Modell überführt. Das Modell enthält je nach zu prüfender Bedingung mehr oder weniger Details über das System. Es muss mindestens über ein Maß an Genauigkeit verfügen, das es erlaubt die zu prüfende Bedingung im Modell zu beobachten. Sollen zum Beispiel analoge Spannungsschwankungen betrachtet werden, kann das System in Form von Differentialgleichungen beschrieben werden, damit kontinuierliche Werte darstellbar sind. Beim klassischen Model Checking wird hingegen nur die Boolesche Interpretation einer Spannung (low and high) betrachtet. In diesem Fall ist eine wertdiskrete und ggf. zeitdiskrete Beschreibung des Modells ausreichend. Passend zum Modell müssen die zu prüfenden Eigenschaften in einer entsprechenden Sprache definiert werden. Zur Beschreibung der Eigenschaft können sowohl Sprachen mit der Mächtigkeit der *Linear-time temporal logic* (LTL) [HR04] oder auch der *Computation Tree Logic* (CTL) [McM93, HR04] verwendet werden. Dabei ermöglicht CTL in den Aussagen Verzweigungen, während LTL nur einen linearen Strang beschreibt. Diese, in LTL geschriebene, zu beweisende und lineare Eigenschaft gilt dennoch allgemeingültig für alle Pfade des Modells. Neben dem Modell und der Sprache wird auch ein Verfahren benötigt das überprüft, ob die zu verifizierende Eigenschaft auf dem Modell gültig ist.

Die gebräuchliche Kombination, die in [McM93, HR04] beschrieben wird, ist die Verwendung von temporalen Strukturen, auch *Kripke-Modell* [Kri63] genannt, mit der Verwendung von CTL zur Beschreibung der zu prüfenden temporalen Aussagen. Ein Kripke-Modell lässt sich leicht aus einer sequenziellen Schaltung ableiten. Mittels der CTL-Fixpointberechnung können die Zustände im Kripke-Modell berechnet werden, in denen die zu beweisenden CTL-Formeln halten, und die somit gültig sind. Dieses Verfahren lässt sich sehr gut automatisieren.

Die Eigenschaften, die mit CTL beschrieben werden können, lassen sich in zwei Kategorien einteilen. Zum einen die sogenannten *Safety Properties*, die Eigenschaften beschreiben, die immer gelten müssen; zum anderen die sogenannten *Liveness Properties*, die ein Verhalten beschreiben, welches eintreten kann. Die Mächtigkeit der Sprache erlaubt die Überprüfung von einfachen bis hin zu sehr komplexen Eigenschaften.

Auf der anderen Seite kann Model Checking durch eine Zustandsexplosion schnell in ein Komplexitätsproblem laufen. Das liegt an der Notwendigkeit den kompletten Zustandsgraph für das System zu modellieren, bevor das Verfahren angewendet werden kann. Die Anzahl der Zustände wächst exponentiell mit der Anzahl an Speicherkomponenten im System.

Mit diesem Verfahren können dann entweder Beweise nur auf kleinen Systemen durchgeführt werden, deren Modell noch handhabbar ist, oder es wird eine höhere Abstraktion des Modells gegenüber dem Originalsystem verwendet, damit das Modell klein genug bleibt. Um die Komplexität besser zu beherrschen, gibt es auch den Ansatz [CLM89] das System in kleinere Teile zu zerlegen und dort lokale Beweise zu führen, die global gültig sind.

Eine Idee, um die Komplexität zu reduzieren, ist die *symbolische Modellprüfung* (engl.: *Symbolic Model Checking*) [CGJ+03]. Um das Problem der Zustandsexplosion zu verringern, werden die Zustände des Systems nicht mehr explizit aufgezählt, sondern mittels Boolescher Formeln beschrieben. Dazu müssen die entsprechenden Booleschen Formeln mit den zugehörigen Zustandsmengen in Form von binären Vektoren und den Übergangsrelationen erzeugt werden. Wie beim klassischen Model Checking werden nun mit Hilfe einer Bildberechnung die gültigen Zustände errechnet. Dabei werden auch diese Zustände nicht als einzelne erreichbare Zustände berechnet, sondern in Mengen von erreichbaren Zuständen. Die Zustandsmengen können mittels reduzierten, geordneten binären Entscheidungsbäumen (engl.: *Reduced Ordered Binary Decision Diagram, ROBDD* [Bry86]) beschrieben werden. Im schlechtesten Fall können diese BDDs ebenfalls exponentiell wachsen.

Bounded Model Checking

Bounded Model Checking (BMC) betrachtet, im Gegensatz zu klassischem Model Checking, nur endliche Ausführungspfade im System, wodurch die Komplexität des Beweises reduziert wird. Dabei werden jedoch nur die Zustände berücksichtigt, die in der beschränkten Ausführungstiefe erreichbar sind. BMC ist nur vollständig, wenn die untersuchte Ausführungstiefe mindestens so groß ist wie die sequenzielle Tiefe des betrachteten Schaltwerkes. Das Schaltwerk kann als Modell beschrieben werden, wie es in Kapitel 2.2 vorgestellt wird. Beim Bounded Model Checking werden diese Automaten über eine endliche Anzahl von Zeitschritten abgerollt, um anschließend auf dem abgerollten Modell die entsprechenden Eigenschaften zu beweisen. Abbildung 2.7 zeigt ein Beispiel für einen abgerollten Automaten (vgl. Abb. 2.2 c)). Das Abrollen für k Zeitschritte erfolgt durch das Konkatenieren von k Instanzen der kombinatorischen Schaltungen δ und λ des Automaten. Dabei steht jede Instanz der Schaltung für einen einzelnen Zeitschritt. Die Rückkopplungskomponente des Automatenmodells wird an den Registern aufgeschnitten. Die Schaltung besitzt nun für jeden abgerollten Zeitpunkt ein entsprechendes Ein- und Ausgangssignal, und die Zustände werden zu internen Signalen der abgerollten Schaltung.

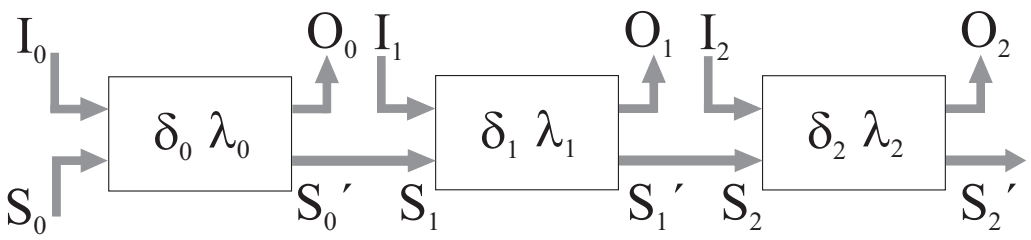

Abbildung 2.7: BMC: Abrollen der Schaltung über die Zeit

Das resultierende Modell ist eine Schaltung, die aus rein kombinatorischer Logik besteht und eine Form hat wie in Abbildung 2.7 dargestellt. In der Abbildung ist der neu

errechnete Zustand der ersten Instanz S'_0 mit S_1 verbunden. So wird die Ausgabe der Übergangsfunktion δ die Eingabe für den neuen Anfangszustand der nächsten Instanz. Allgemein ausgedrückt wird im Zeitschritt t_i das Signal S'_i mit dem Signal S_{i+1} verbunden. An den Zustandsvariablen S_0 der ersten Instanz wird der Anfangszustand des Automaten angelegt.

Dieses abgerollte Modell kann verwendet werden, um Eigenschaften auf dem Design zu prüfen. Dafür wird eine temporale Logik verwendet, die sich auf die einzelnen Signale des abgerollten Modells beziehen kann. Da BMC ein White-Box-Verfahren ist, können neben den Ein-, Ausgängen und den Zustandssignalen auch interne Signale der Logik in den Eigenschaften verwendet werden. Die Eigenschaften werden ebenfalls als eine Boolesche Funktion ausgedrückt.

Eine automatisierte Verifikation, die prüft, ob die beschriebene Eigenschaft auf dem ausgerollten Modell gültig ist, ist durch Abbildung auf ein Erfüllbarkeitsproblem (*SAT, engl. satisfiability*) möglich. In [BCC+99] wird gezeigt, wie eine solche Verifikation mit Hilfe eines sog. SAT-Solvers gelöst werden kann. SAT solving bezeichnet ein Verfahren, das eine erfüllbare Belegung für eine gegebene boolesche Formel ermittelt. Es wird vorausgesetzt, dass die Formeln in *konjunktiver Normalform* (KNF) vorliegen. Zum Start des SAT-Solvers sind alle Klauseln unbelegt. Durch Festlegung einzelner Variablen auf den Wert 0 oder 1 (also *true* oder *false*) werden die Klauseln entweder erfüllbar, unerfüllbar oder sind noch nicht bestimmt, da sie noch von weiteren Variablen abhängen. Ist eine Klausel unerfüllbar, widerspricht das dem Ziel, eine gültige Belegung zu finden. Geschieht das, wird mit Hilfe eines Backtracking-Verfahrens die letzte Entscheidung zurückgenommen und eine andere Belegung gewählt. Wird keine Klausel unerfüllbar und ist die Formel noch nicht belegt, wird die nächste Variable auf einen Wert festgelegt. Dieses Verfahren läuft solange, bis eine Belegung gefunden wird, die die Formel erfüllt, oder eben keine erfüllende Belegung gefunden wird.

Um das BMC-Problem auf SAT abzubilden, wird das zu beweisende Problem in die entsprechende KNF übersetzt. Dazu wird die zu prüfende Eigenschaft p, das k-Schritte abgerollte kombinatorische Modell M_k und eine boolesche Repräsentation X_{s0} der Startzustände S_0 jeweils als KNF-Formel benötigt. Das zu lösende Erfüllbarkeitsproblem hat dann die Form: $X_{s0} \wedge M_k \wedge \neg(p)$.

Da für eine allgemein gültige Aussage immer die sequenzielle Tiefe des Systems erreicht werden muss bzw. für einen Beweis in einem bestimmten Zustand erst einmal der Automat solange vom gültigen Startzustand abgerollt werden muss, bis dieser betrachtete Zustand erreicht ist, kann das zugrunde liegende Modell immer noch sehr groß werden. Zusätzlich muss auch überprüft werden, ob die gewählte Tiefe k ausreicht.

Um dieses Problem zu umgehen bzw. die notwendigen Schritte beim Absrollen zu minimieren, gibt es ein Verfahren namens *Interval Property Checking* (*IPC*) [NTW+08], das dem BMC sehr ähnlich ist. Der Unterschied liegt darin, dass anstelle eines bekannten Anfangszustandes aus S_0 jeder beliebige Zustand angenommen werden kann. Der Vorteil liegt darin, dass dieses Modell nur so viele Zeitschritte abgerollt werden muss, wie die zu beweisende Eigenschaft benötigt, um komplett dargestellt zu werden.

Dadurch kann IPC größere Schaltungen handhaben. Der Nachteil ist, dass ein durch

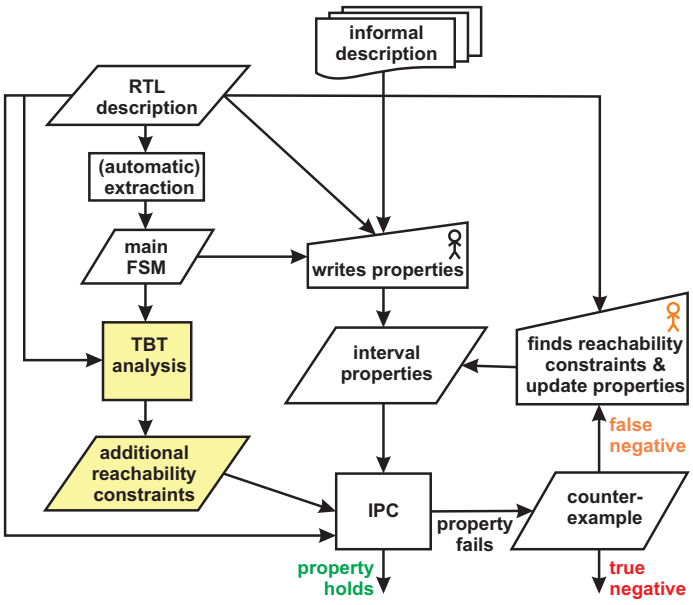

Abbildung 2.8: Ablauf von Interval Property Checking [NTW+08]

den SAT-Solver gefundenes Gegenbeispiel eventuell auf einem ungültigen Anfangszustand basiert, ein sogenannter *False Negative*. Dieser ungültige Anfangszustand muss vom Verifikationsingenieur erkannt werden. Um diesen unerwünschten Fall auszuschließen wird einer zusätzlichen Annahme, die diesen Zustand zu Beginn ausschließt, hinzugefügt. Danach kann die Eigenschaft erneut geprüft werden. So ergibt sich ein iterativer Ablauf, wie in Abbildung 2.8 gezeigt. Dabei kann die Anzahl an False Negatives durch den in [NTW+08] vorgestellten Algorithmus "Transition-By-Transition (TBT) FSM traversal" stark reduziert werden.

Um herauszufinden, ob bei den geprüften Eigenschaften – zum Beispiel durch zusätzliche hinzugefügte Annahmen – keine Verifikationslücken entstehen und so relevante Fehler übersehen werden können, gibt es die Möglichkeit mit den Ansätzen [BBM+07, Cla07] eventuelle Lücken zu erkennen.

2.6.2 FV für Software

Bei der formalen Verifikation von Software sind die Ansätze überwiegend an der sequenziellen Struktur von Software orientiert. Deswegen verwenden viele Methoden und Tools einen pfadorientierten Ansatz. Die dahinter liegenden Beweistechniken sind überwiegend die gleichen, die auch zur formalen Verifikation von Hardware eingesetzt werden. Sie lassen sich in folgende Kategorien aufteilen:

- Abstrakte Interpretation (engl.: *Abstract Interpretation*)
- Modellprüfung (engl.: *Model Checking*)
- Symbolische Ausführung (engl.: *Symbolic Execution*)

- Bounded Model Checking

Dabei kann bei einzelnen Verfahren zusätzlich noch zwischen expliziter oder symbolischer Repräsentation der Zustände oder der Daten in den Modellen unterschieden werden.

Einen guten Überblick über formale Ansätze für die Softwareverifikation geben [JM09] und [DKW08].

Abstrakte Interpretation

In diesem Ansatz aus [PR77] wird im Allgemeinen kein Verifikationsmodell erzeugt, mit dem der anschließende Beweis durchgeführt wird. Stattdessen werden hier die arithmetischen Formeln von der vorhandenen Software direkt berechnet, was gleichzeitig auch der Beweisführung entspricht. Zusätzlich müssen das Speichermodell und die entsprechenden Speicherzugriffe vom Programm modelliert werden. Als Basisinformationen für das Speichermodell dienen prozessorspezifische Informationen wie zum Beispiel Datentypengrößen. So kann das Verfahren dafür verwendet werden, um bestimmte Randbedingungen des Programms an beliebigen Stellen im Programm zu ermitteln und zu garantieren. Damit können Laufzeitfehler wie "Array out of Bound", Division durch Null oder Variablenüberläufe erkannt werden [HT10]. Diese Technik findet Anwendung in den kommerziellen Tools Astrée [Abs] und Polyspace Codeprover [The]. Es ist eine der wenigen formalen Verifikationstechniken für Software, die kommerziell eingesetzt wird [BCC+15, KSS+19]. Abstrakte Interpretation kann viele Laufzeitfehler in der Software automatisiert prüfen, ohne dass es notwendig ist Eigenschaften von Hand zu schreiben. Eigene Eigenschaften können wie auch beim BMC formuliert werden, doch sind das in der Regel einfache Eigenschaften, die sich nur auf lokale Stellen im Programm beziehen.

Modellprüfung

Die zugrundeliegende Verifikationstechnik der Modellprüfung für Software ist die gleiche wie die der Hardwareverifikation (s. 2.6.1). Entsprechend besteht das Modell eines Programms aus Zuständen und Transitionen. Der Zustand eines Programms beinhaltet die Programmposition, die die jeweils aktuelle Instruktion identifiziert, sowie alle Variablen, die im Arbeitsspeicher liegen, und alle weiteren Daten, die dem Programm zugeordnet sind. Je nach Genauigkeit des Modells können das zusätzliche Daten vom Stack oder auch der Architekturzustand sein, der aus Registern und Konfigurationsregistern besteht. Diese Zustände sind mit Transitionen verbunden, wobei jede Transition einer bestimmten Instruktion des Programms zugeordnet werden kann, die abhängig vom Zustand und gemäß der Art der Instruktion in den nächsten Zustand überführt. Die Reihenfolge der Transitionen ist durch das Programm vorgegeben. Viele Model Checker arbeiten mit einer hohen Programmiersprache wie C, C++ oder Java.

Eigenschaften können beim Software Model Checking, wie beim Hardware Model Checking, mittels CTL- oder LTL- Formeln beschrieben werden. Um Eigenschaften auf

dem Modell erfolgreich zu beweisen, also zu zeigen, dass der zu überprüfende Fehlerzustand nie auftritt, muss das Modell alle Zustände des Programms beinhalten.

Für die Repräsentation von Zuständen gibt es verschiedene Ansätze: Sie können entweder explizit aufgezählt werden [Hol97] oder mit Hilfe von Pfaden – also ohne explizite Darstellung aller Zustände dargestellt werden [God05, HP00]. Diese pfadorientierten Ansätze gehören zu den ausführungsbasierten Modellprüfungsverfahren. Das Modell wird nur entlang des aktuell betrachteten Pfades erzeugt und nicht für alle möglichen Pfade. Das hat den Vorteil, dass nicht das vollständige Modell im Speicher vorgehalten werden muss, sondern Auszüge des Modells ausreichen.

Ist ein Zustand gefunden, der einen Fehler ermöglicht, ist die Aussage – aus der zu prüfenden Eigenschaft – widerlegt, und der Model Checker kann stoppen. Mit dem ermittelten Pfad wird ein Gegenbeispiel erzeugt. Im schlechtesten Fall müssen allerdings alle Pfade berechnet werden, was – je nach Implementierung – in einen hohen Aufwand münden kann. Dann ist das betrachtete Modell potentiell ähnlich komplex oder sogar komplexer als bei den Ansätzen, die das vollständige Modell vor der Verifikation generieren.

Eine weitere Möglichkeit ist, die Zustände symbolisch zu ermitteln. Dadurch kann bei einer expliziten Darstellung von Zuständen mit Hilfe eines symbolischen Zustands eine größere Menge expliziter Zustände modelliert werden. Bei den ausführungsbasierten Ansätzen gibt es auch gemischte Ansätze, in denen Teile der Zustandsvariablen explizit berechnet werden und andere symbolisch bestimmt sind. Dabei werden Zustände, die essentiell für den Programmfluss sind, explizit dargestellt. Bei einem Assemblerprogramm ist es zum Beispiel sinnvoll, die Zustände des "Program Counter" explizit aufzuzählen, da dieser unmittelbar die Position im Programm repräsentiert. Variablen, die keinen großen Einfluss auf den Programmablauf haben, können dagegen symbolisch als Zustandsmenge beschrieben werden.

Dies führt zu einer Reduktion der Anzahl an explizit aufgezählten Zuständen. Dieses Verfahren ist zum Beispiel in JavaPathFinder [HP00] und auch in den meisten anderen Model Checkern implementiert. Eine Übersicht über Model Checker und ihre Unterstützung symbolischer Werte ist in [DKW08, S. 12] gegeben. Um zu zeigen, ob das zu verifizierende Programm korrekt ist, muss überprüft werden, ob die gegebenen Eigenschaften erfüllt werden.

Ein Model Checker, der mit Assemblercode arbeitet, ist [MC]Square [Sch10]. Dort wird ein CFG anhand des Assemblerprogramms erzeugt und dieser für die Verifikation herangezogen. Da Assemblercode hardwareabhängig ist, wird dabei für jeden Mikrocontroller, den das Tool unterstützt, ein eigenes Modell vorgehalten, welches den Mikrocontroller abstrakt beschreibt. In [Sch10] wird ein simulativer Ansatz zur Berechnung der möglichen Nachfolgezustände verwendet. Der Ansatz unterstützt teilweise auch symbolische Zustände.

Symbolic Execution

Ein weiterer – den ausführungsbasierten Ansätzen ähnlicher – Ansatz ist Symbolic Execution [CS13] Dieser basiert darauf, dass es zu jeder Programmiersprache eine Ausführungssemantik gibt. Anhand dieser kann ein Kontrollfluss extrahiert werden, der wie bei einem CFG die möglichen Nachfolgezustände des Programms vorgibt. Einer der bekanntesten Vertreter von Symbolic Execution ist das Softwarewerkzeug Klee [CDE08].

Die Position im Programm ist durch die Extraktion des CFG explizit gegeben. Die einzelnen Instruktionen werden nacheinander "ausgeführt" und dabei die symbolischen Nachfolgewerte des Zustandes berechnet. Dieser dient dann als Eingabe für die folgende Instruktion. Im Falle einer bedingten Verzweigung wird der Zustand an beide Zweige weitergegeben, und zusätzlich auch eine Bedingungsvariable mitgegeben, unter denen der jeweilige Zweig aktiv ist. So wird das Programm anhand des extrahierten CFG durchlaufen. Es entstehen nur Nachfolgezustände, die durch den CFG des Programms gegeben sind. Das resultierende Modell ist ein Ausführungsbaum (engl.: *Execution tree*), der alle möglichen Ausführungspfade beinhaltet, und für jeden Pfad einen Zustand besitzt, der gerade für die betrachtete Instruktion des Pfades aktuell ist. Da die Anzahl der Pfade sehr groß werden kann, wurden in anderen Implementierungen [PV09, AEO+08] Optimierungen durchgeführt, um das Modell möglichst klein zu halten. Das wird erreicht, indem beim Verzweigen überprüft wird, ob die Pfadbedingungen überhaupt möglich sind. Wenn das nicht der Fall ist, wird der Pfad weg optimiert. Zusätzlich können isomorphe Teilgraphen, also identische Teilgraphen, die mehrere Pfade beinhalten, wieder zusammengefasst werden (s. [AEO+08]).

Symbolic Execution kann auch kompositional, wie in [PV09], verwendet werden. Dabei werden Funktionsaufrufe innerhalb des Programms als definierter Schnitt angesehen. Statt die Instruktionen der Funktion in die Ausführung mit aufzunehmen wird die aufgerufene Funktion separat vom restlichen Programm betrachtet. Wird beim Abarbeiten der einzelnen Instruktionen ein Funktionsaufruf erreicht, hält die aktuelle symbolische Ausführung an, und eine für die Funktion eigene symbolische Ausführung wird gestartet. Dazu wird der Zustand des Programms an der Stelle des Funktionsaufrufs als Eingabeparameter verwendet. Nach dem Beenden der Funktionsbearbeitung wird das Ergebnis der symbolischen Ausführung als "Zusammenfassung" für die weitere symbolische Ausführung im aufrufenden Code verwendet. Der Vorteil liegt in der Reduktion der Gesamtzahl an gleichzeitig zu betrachtenden Pfaden für die getrennte symbolische Ausführung, im Vergleich zur Anzahl an Pfaden in einem gemeinsamen Gesamtmodell. Die Optimierung nutzt vor allem die Beschaffenheit der Funktionsaufrufe. Die Pfade, die innerhalb der aufgerufenen Funktion entstehen, werden am Ende der Funktion auf eine oder einige wenige Rücksprunginstruktionen zusammengeführt.

Bounded Model Checking

Da die Modellgröße bei klassischem Model Checking potentiell recht schnell zunehmen kann, gibt es wie bei der Hardware auch Softwareansätze, die das Problem mit Bounded Model Checking und SAT lösen. Dabei betrachten viele Lösungen Programme in einer

$$
\begin{array}{lcl}
x = x + y & & x_1 = x_0 + y_0 \\
if(x! = 1) & & if(x_1! = 1) \\
\quad x = 2; & & \quad x_2 = 2; \\
else & \Rightarrow & else \\
\quad x++; & & \quad x_3 = x_1 + 1; \\
 & & x_4 = (x_1! = 1)?x_2 : x_3; \\
assert(x <= 3); & & assert(x_4 <= 3);
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
C := x_1 = x_0 + y_0 \\
\quad \wedge x_2 = 2 \\
\quad \wedge x_3 = x_1 + 1 \\
\quad \wedge x_4 = (x_1 \neq 1)?x_2 : x_3 \\
\\
P := x_4 \leq 3
\end{array}
$$

Abbildung 2.9: Beispiel für die Erzeugung des zu prüfenden Modells bei CBMC [CKL04]

hohen Programmiersprache wie C, wie zum Beispiel [BH08, IYG+08] und [CKY03]. In den BMC-basierten Verfahren für Software wird der C-Code instrumentiert und dabei um die zu prüfenden Eigenschaften erweitert. Dabei wird in [CKY03] und in [BH08] der Befehl *assert*() eingefügt. Innerhalb der Klammern steht die zu prüfende Eigenschaft, die als Ergebnis einen Booleschen Wert liefert.

Das Programm inklusive der im Programm befindlichen Eigenschaften muss in logische Formeln umgewandelt werden, damit die Eigenschaften mittels eines SAT-Solvers bewiesen oder widerlegt werden können. In [CKY03] wird ein Verfahren in zwei Schritten vorgeschlagen.

Zuerst wird in einem C-Programm Instruktion für Instruktion eine Formel erstellt, die angibt, welche Variablen wie verändert werden. Wird einer Variablen erneut ein Wert zugewiesen, wird sie neu indiziert, und somit eine neue Variable generiert. So entsteht eine Datenkette, die das komplette Verhalten des Programms wiedergibt. Eine zeitliche Information ist nur noch durch den Index bzw. durch den Fan-In der zugehörigen Logik zu erkennen.

In Abbildung 2.9 wird in einem Beispiel gezeigt, wie die einzelnen Zugriffe indiziert werden. Die Variablen x_0 und y_0 stehen für die Anfangswerte, die in dem Programmfragment keinen festen Wert besitzen. Sie entsprechen einem primären Eingang (primary input) beim BMC für Hardware.

Zusätzlich wird im Fall einer Verzweigung auch die Verzweigungsbedingung mit betrachtet. Diese Information wird benötigt, wenn eine Variable auf verschiedenen Pfaden geschrieben wird und diese Pfade später wiedervereinigt werden. So kann anhand der Verzweigungsbedingung entschieden werden, von welchem Pfad der Variablenwert übernommen wird. Innerhalb der Verzweigung sind keine Bedingungsprädikate notwendig, da durch die Indizierung der Variablen in jedem Pfad eine eigene Variable für den Wert x existiert.

Nach dem Indizieren der Variablen wird jede Variable nur noch ein einziges Mal geschrieben, so dass dieses Programm nun einzelne logische Formeln ohne Zeitbezug besitzt die im zweiten Schritt jeweils entsprechend in eine Eigenschaft übersetzt werden können. Auf der rechten Seite der Abb. 2.9 ist die zu prüfende Eigenschaft beschrieben. Dabei werden die Formeln, die vom Programm gegeben sind, in den Teil (C : = *(*Commit)) der Eigenschaft übernommen. Die bei der Instrumentierung hinzugefügten *assert*()-Instruktionen werden in den zu prüfenden Bedingungsabschnitt (P : = *(*Prove))

übernommen. Diese logische Aussage kann mittels eines SAT-Solvers geprüft werden, der entscheidet, ob die Bedingung erfüllt ist, oder ob ein Gegenbeispiel gefunden werden kann. Da dieses Modell rein kombinatorisch ist, muss es nicht zusätzlich über die Zeit abgerollt werden, wie es bei der Hardware der Fall ist.

Im Fall von Schleifen im Programm muss der Anwender im Vorfeld angeben, wie oft diese Schleife ausgerollt werden soll. Diese Grenze sollte der Verifikationsingenieur so wählen, dass sie nicht kleiner ist als die in tatsächlichen Programmläufen zu erwartende maximale Anzahl an Schleifendurchläufen.

Wird die Grenze zu klein gewählt, kann es vorkommen, dass nicht alle Programmausführungen im Modell berücksichtigt werden. Im Tool CBMC ist als Weiterentwicklung inzwischen ein Mechanismus zum automatischen Abrollen von Schleifen implementiert worden.

Ein anderer Ansatz die Schleifen einfacher berechnen zu können wird in [BH08] beschrieben. Der Ansatz wurde in dem Tool Calysto implementiert, welches von CBMC und von SATURN [XA07], einem SAT-basierten Tool der Universität Stanford, inspiriert wurde. Calysto benötigt weder die manuelle Angabe der Schleifenlängen bzw. Invarianten, stattdessen setzt es einen automatisierten heuristischen Ansatz zur Berechnung von Schleifenpfaden ein. Im Unterschied zu CBMC werden die Schleifen dabei nur einmal ausgerollt. Das kann aber zu False Positives führen. Sind innerhalb der Schleife noch bedingte Ausführungspfade (If-Then-Else Code Elemente), ist es möglich, dass nicht alle Fälle abgeprüft werden, wenn beim ersten Ausrollen nicht alle Bedingungen erreichbar sind. So können Designfehler übersehen werden.

In [IYG+08] wird das Tool F-Soft mit seiner BMC-basierten Implementierung vorgestellt. Der Ansatz unterscheidet sich von CBMC und Calysto insofern, als das Modell noch über eine bestimmte Anzahl von Schritten ausgerollt werden muss. Um das zu ermöglichen, wird ein spezieller CFG wie in Abbildung 2.10 gebaut, der sequenzielle Blöcke der einzelnen Programmteile zusammenfasst und diese über einen Onehot-Coded-Zähler aktiviert werden können.

Die SAT-Engine kann dieses System dann systematisch ausrollen. Dabei sind nicht beliebige Blöcke als Nachfolger andere Blöcke möglich. So muss der SAT-Solver die möglichen Nachfolger-Blöcke berechnen, wobei nur wenige Nachfolger möglich sind. Die Sequenzen in den Blöcken können unterschiedlich lang sein. Das Beispiel in Abbildung 2.10 zeigt eine Funktion $bar()$ mit der Unterfunktion $foo()$. Die Sequenzen, die über keine Verzweigungen verfügen, werden zusammengefasst. Dies sind im Beispiel die beiden Instruktionen $int\ x = 3;$ und $y = x - 3;$. Die Pfeile ergeben sich aus den Übergängen zwischen den einzelnen Sequenzen. Zusätzlich werden noch Zwischenmodule für Funktionsaufrufe generiert.

CBMC, zum Vergleich, übergibt eine Formel, bei der der SAT-Solver nicht noch zusätzlich zum Beweis die möglichen Sequenzen berechnen muss.

Neben SAT-Solvern können auch sogenannte "*Sat Modulo Theorie*"-Solver (SMT-Solver) verwendet werden. Gegenüber reinen SAT-Solvern haben sie den Vorteil, dass sie die arithmetischen Operationen nicht in bitweise Operationen zerlegen müssen, sondern auf Wortebene arbeiten. So unterstützt CBMC neben SAT-Solvern auch SMT-Solver

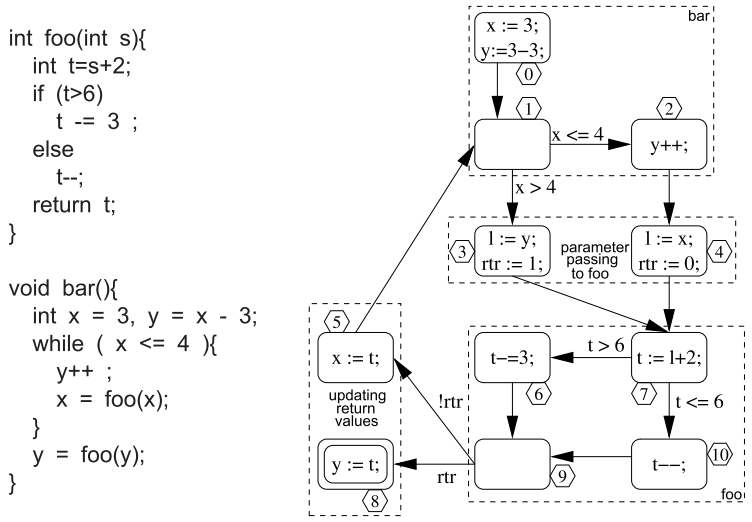

Abbildung 2.10: F-Soft: Beispiel Programm und zugehöriger CFG [IYG+08]

wie z. B. Z3 von Microsoft. Andere BMC Tools basieren ausschließlich auf SMT wie SAGE [GLM12] und SLAM [BR].

2.7 HW/SW Co-Verifikation

In den beiden vorherigen Kapiteln 2.5 und 2.6 wurde Hardware- und Softwareverifikation jeweils getrennt betrachtet. Das Ziel dieser Arbeit ist, den Einfluss der hardwarenahen Software auf die Hardware zu betrachten und diese zu verifizieren. Dieses Kapitel zeigt vergleichbare Ansätze. Es gibt jedoch wenige Ansätze für eine echte Hardware/Software-Co-Verifikation.

Ein Ansatz, der auf klassischem Hardware-BMC basiert, wird in [NWSK11] vorgestellt. Das Programm wird in ein ROM geschrieben und dieses zusammen mit dem Speicher, der Peripherie und der CPU als Hardwaremodell, wie in Abbildung 2.11 gezeigt, mittels BMC verifiziert. Beim Ausrollen wird in BMC die vollständige Hardware inklusive ROM für jeden Zeitschritt neu instanziiert, wodurch schnell ein komplexes Modell entsteht. So können Eigenschaften nur über kurze Zeitbereiche verifiziert werden. Komplexere Eigenschaften wie beispielsweise die funktionale Korrektheit der Übertragung eines kompletten Datentelegramms wie es in [NWSK11] für das LIN-Protokoll vorgestellt wird, können aus Komplexitätsgründen nicht bewiesen werden. Deswegen wird in [NWSK11] eine Abstraktion vorgeschlagen, bei der die große zu prüfende Eigenschaft in kleinere Stücke zerlegt wird. Kleiner bedeutet in diesem Fall, dass eine zeitliche Zerlegung der Eigenschaften durchgeführt wird, um die Anzahl an Zeitschritten, die das Modell abgerollt wird, zu minimieren. Diese Stücke lassen sich mittels Interval Property Checking zügig beweisen. Die Zerlegung der Eigenschaften wird in Abbildung 2.12 nochmals verdeutlicht. Um dennoch das Verifikationsziel zu erreichen, werden die einzelnen Eigenschaften als Abstraktion der konkreten RTL-Implementierung verwendet. Das wird

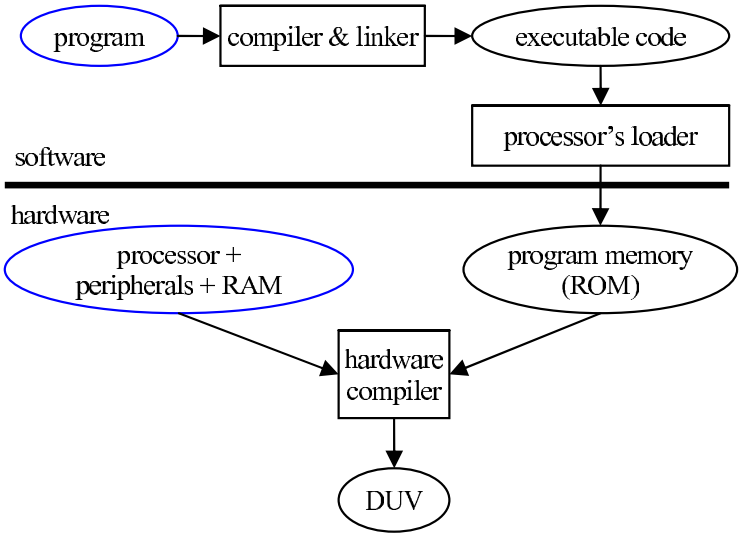

Abbildung 2.11: HW/SW-Modell Generierung nach [NWSK11]

erreicht, indem die Eigenschaften wieder konkateniert werden und somit das resultierende abstrakte Modell bilden, auf dem die vollständigen Verifikationsziele geprüft werden können. Können die globalen Eigenschaften auf dem abstrakten Modell erfolgreich geprüft werden, so ist gezeigt, dass die Abstraktion der RTL-Implementierung das zu prüfende Verhalten und somit auch die tatsächliche RTL-Implementierung die geforderten globalen Eigenschaften erfüllt. Dieses Verfahren erfordert einen hohen manuellen Aufwand und lässt sich wegen der zu formulierenden Eigenschaften nicht automatisieren.

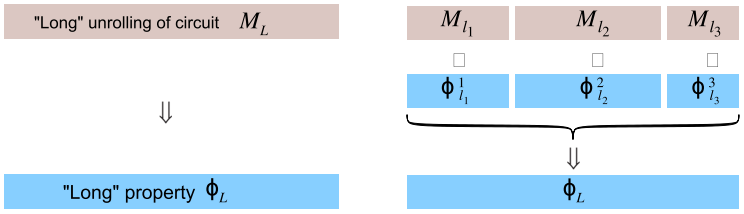

Abbildung 2.12: Zerteilung der Eigenschaften in kleinere Teileigenschaften [NWSK11]

Einen ähnlichen Ansatz verfolgen auch [EES04] und [GKD06]. Bei [GKD06] wird das Hardwaremodell der CPU auf einer hohen SystemC-Ebene implementiert, in der das abgebildete Programm in Form eines Programmspeichers integriert wird. Die Aufgabe des BMC ist es, das System entsprechend der benötigten Anzahl an Schritten auszurollen, um anschließend den Beweis zu führen. Dafür wird ein SAT-basierter Ansatz verwendet. Auf eine weitergehende Abstraktion zur Reduktion der Komplexität wird hier hingegen verzichtet.

In [HTV+13] wird ähnlich zu [GKD06] das Hardwaremodell auf einer höheren Ebene wiedergegeben. Statt eine komplette formale Verifikation auf dem gemeinsamen Modell zu erzeugen, werden mittels symbolischer Ausführung (vergl. 2.6.2) Multi-Pfad-

Ausführungen erzeugt, die im Anschluss mittels BMC auf potenzielle Fehler formal verifiziert werden. Der Vorteil des Verfahrens ist, dass es gut skaliert und so Fehler auf größeren Modellen finden, aber nicht die Abwesenheit von Fehlern beweisen kann.

Ein weiterer Ansatz für eine HW-/SW-Co-Verifikation wird anhand eines konkreten Beispiels mit einer speziellen Variante von CBMC in [MPPK17] vorgestellt. Die spezielle Variante von CBMC, HW-CBMC [CKY03], ist um die Fähigkeit erweitert, auch HDL-Code in Form von Verilog zu lesen und mit ins interne Modell zu übernehmen. Dies erlaubt sowohl Software wie Hardware in eine gemeinsame Verifikation zu übernehmen und ermöglicht so theoretisch eine Durchführung von echter Co-Verifikation. Dabei laufen Hardware und Software nicht komplett parallel zueinander, sondern die Hardware muss über Extrabefehle, die in den Code integriert werden müssen, getriggert werden.

2.8 FV hardwarenaher Software mit Interrupts

Eingebettete Systeme zeichnen sich durch intensive Interaktion zwischen Hard- und Software aus. Hierbei werden Hardwareinterrupts, wie sie in Kapitel 2.4.3 beschrieben wurden, genutzt um Kommunikation ausgehend von der Hardware mit der Software zu initiieren. Um die Korrektheit der Kommunikation mittels formaler Verifikation überprüfen zu können, ist es notwendig, die Interrupts im Verifikationsmodell ebenfalls zu berücksichtigen. Ein Ansatz für die Berücksichtigung von Interrupts bei Low-Level-Software (Assembler) wird in [SNBB11] vorgestellt. Dort wird die Abhängigkeit zwischen den Interrupt-Routinen und dem Hauptprogramm ermittelt und ein entsprechender Graph der Software erzeugt, bei dem zwischen je zwei Instruktionen ein Interrupt möglich ist. Diese eingefügten Interrupt-Service-Routinen werden vereinfacht als atomar angesehen und somit im Graph als nur eine einzelne Instruktion dargestellt.

Um die Anzahl der ISR-Aufrufe zu reduzieren, wird mit Hilfe des in [SNBB11] beschriebenen Abstraktionsverfahrens IHER der Graph annotiert und entschieden, ob ein Aufruf der Interrupt-Service-Routine das Programm an den jeweiligen Stellen beeinflussen kann. Zum Beispiel wird das Programm von der ISR beeinflusst, wenn ein Register modifiziert wird, welches vom Hauptprogramm genutzt wird. Andersherum kann ebenfalls das Hauptprogramm die Interrupt-Handler beeinflussen. So kann das Programm den Interrupt abschalten oder Register schreiben, die der Interrupt ebenfalls liest. Abhängig von diesen Stellen, bei denen die Programme sich gegenseitig beeinflussen, wird mit einem automatischen Verfahren entschieden, ob es notwendig ist, den Interrupt-Handler-Aufruf im Graphen zu ermöglichen. Ist das nicht der Fall, wird dieser entfernt. Die Durchführung im CFG des Assemblerprogramms erfordert die gesonderte Behandlung von Schleifen, da diese nicht ausgerollt werden. Zur Verifikation der Abstraktion als auch der eigentlichen späteren Eigenschaften wird [MC]Square [Sch10] verwendet. Dies ist ein expliziter Model Checker, der teilweise auch symbolisches Model Checking unterstützt.

Weitere Ansätze auf C-Ebene wie [KLM+15, VHHK15] verfolgen ähnliche Ideen zur Modellierung. In [KLM+15] werden verschiedene Varianten und Verifikationstools evaluiert und experimentell gezeigt, dass das Ausnutzen von Zusatzinformationen über

die Interrupts, wie z.B. der Priorität, hilfreich ist. Durch das Erstellen einer Halbordnung, die dieses Wissen repräsentiert, ist eine präzisere Überapproximation der möglichen Interrupt-Aufrufe möglich. Dieses Wissen ermöglicht eine Reduktion der unnötigen Interrupt-Aufrufe, die im realen System wegen der Priorität nicht passieren können, aus dem Modell zu entfernen und reduziert somit die Gesamtkomplexität des Modells. Dies hat einen positiven Einfluss auf die benötigte Rechenzeit und Speicherbedarf für die Verifikation. Die größere Herausforderung für die Verifikation von Interrupts ist ein Betriebssystem oder andere Systemsoftware wie Treiber, da dort in der Regel viele Interrupts vorkommen. In [VHHK15] werden Eigenschaften von C-Programmen die die Interrupts aus dem OS nutzen verifiziert. Dazu wird die Software instrumentiert. Ähnlich zu [SNBB11] wird an jeder möglichen Stelle ein Interrupt-Aufruf integriert, der nichtdeterministisch aufgerufen werden kann oder übersprungen wird. Dabei unterstützt das Verfahren aus [VHHK15] ebenfalls die Möglichkeit die, durch die Priorisierung von Interrupts erzeugte, Halbordnung zu nutzen, um so unnötige – nicht im realen System auftretende – Interrupt-Aufrufe zu vermeiden. Sowohl [VHHK15], als auch die von [KLM+15] favorisierte Lösung verwenden CBMC als Backend.

Kapitel 3

Basisidee für das Modell

In diesem Kapitel wird die grundsätzliche Idee der in dieser Arbeit vorgestellten Methodik beschrieben. Es enthält alle Informationen, um zu verstehen, wie diese Methodik grundlegend funktioniert, und verzichtet auf Details, die zur Vollständigkeit und zur Korrektheit des Verfahrens notwendig sind. Kapitel 4 (Modellgenerierung) stellt alle Details vor, die für die vollständige Methodik zur Generierung des Modells verwendet werden, und zeigt, warum das resultierende Modell dem Originalprogrammfluss entspricht. Zusätzlich werden in Kapitel 4 Optimierungen beschrieben, die bei der Generierung verwendet werden, und die zur Erstellung eines kompakten Gesamtmodells notwendig sind.

In Kapitel 2 wurden verschiedene Methoden und Werkzeuge vorgestellt, die zur formalen Verifikation von Hardware oder Software verwendet werden. Dabei werden die meisten zugrundeliegenden Beweistechnologien sowohl in der Hardware- als auch in der Softwareverifikation verwendet. Die Zahl der Ansätze, die eine echte Hardware-/Software-Co-Verifikation ermöglichen, ist jedoch sehr gering. Die in Kapitel 2.7 vorgestellten Verfahren dieser kombinierten Ansätze erreichen in Abhängigkeit der Problemgröße schnell einen Komplexitätsgrad, der eine effiziente formale Verifikation undurchführbar macht.

Die Herausforderung bei der HW-/SW-Co-Verifikation liegt an der Kombination von Hardware und Software. Hardwarekomponenten arbeiten zumeist parallel und das HW-Modell bei der HW-Verifikation muss oft nur wenige Zeitschritte ausgerollt werden, um die gewünschten Eigenschaften zu prüfen. Dadurch ist die Größe des Hardwaremodells für die Komplexität der Verifikation relevant. Bei der Software hingegen können auch kleine Programme durch ihre hohe sequentielle Tiefe die Komplexität der SW-Verifikation erheblich steigern. Die Tiefe ergibt sich aus der Anzahl an Instruktionen die auf einem Pfad maximal ausgeführt werden können. Die Tiefe hängt stark von der Struktur des Codes und vor allem von den im Code eingebetteten Schleifen ab. Diese Tiefe trägt je nach verwendeter Verifikationstechnik stark zur Komplexität des Verifikationsproblems der Software bei.

Bei großen Hardwaredesigns – gepaart mit einer hohen Ausrolltiefe für die verwendete Software – ist ein Beweis einer HW-/SW-Co-Verifikation mit klassischen Mitteln oft nicht möglich, wie zum Beispiel in [NWSK11] gezeigt wird.

Der hier vorgestellte Ansatz beruht auf der Idee, dass verschiedene Arten der Veri-

fikationsprobleme mit unterschiedlichen Verfahren besser oder schlechter gelöst werden können. Eine geschickte Kombination von verschiedenen Verfahren kann somit die Komplexität der Verifikation reduzieren.

Die Betrachtung von hardwarenaher Software bei formaler Verifikation kann – bedingt durch den Aufbau des Programms und je nach Komplexität der Berechnungen (wie schon vorher beschrieben) – ein komplexes Modell erfordern. Das Programm verfügt dabei über zwei Informationen: Zum einem besitzt es durch seine Struktur, die sich auch als CFG abbilden lässt, einen Programmfluss. Gleichzeitig geben die Instruktionen vor, welche Datenberechnungen durchgeführt werden. So besitzt der CFG auch implizit Informationen über den Datenfluss, ohne die genaue Implementierung der einzelnen Instruktionen zu kennen.

Die gültigen Pfade durch den CFG des Programms entscheiden, welche Instruktion des Programms als nächste ausgeführt wird. Auch wenn bedingte Sprünge im Programm von externen oder berechneten Daten abhängen, so sind die möglichen Sprungziele in der Regel festgelegte Stellen im betrachteten CFG. So wird der Programmfluss grundsätzlich durch die Struktur der Software beschrieben.

Die Datenberechnung hängt stark vom Programmfluss ab, da dieser über die Reihenfolge der ausgeführten Instruktionen entscheidet und die finale Programmsemantik bildet. Die Datenberechnung innerhalb einer Instruktion oder sogar in einer Sequenz von Instruktionen ohne Verzweigungen ist jedoch immer identisch. Die Berechnung der Daten wird von der Prozessorhardware durchgeführt und durch die Software gesteuert.

Der Programmfluss ("control") und die eigentliche Berechnung der Daten ("computation") sind in der Betrachtungsweise sehr unterschiedlich.

Die in dieser Arbeit vorgestellte Idee beruht darauf, dass *control* und *computation* in getrennten Phasen betrachtet und bearbeitet werden.

In der ersten Phase wird der Programmfluss evaluiert und dabei ein *Execution Graph* (*EXG*) erstellt, der als Ausführungsgraph alle möglichen Pfade durch das Programm explizit repräsentiert. Bei der Erstellung des Graphen werden zusätzlich automatisierte Techniken wie das Zusammenführen von isomorphen Untergraphen (*Merging*) und das Abschneiden von toten Pfaden eingesetzt, um den Graphen kompakt zu halten.

Der Execution Graph ist der expliziten Darstellung des Programmflusses in Symbolic Execution sehr ähnlich. Der Unterschied zwischen den beiden Methoden ist, dass in Symbolic Execution nicht nur der Programmfluss, sondern auch die Programmsemantik – und somit ebenfalls die Berechnung der Daten – explizit evaluiert wird, um symbolische Formeln zu erstellen, die in der hiesigen Darstellung des Execution Graph außen vor gelassen werden.

In der zweiten Phase wird der erzeugte Execution Graph um die Programmsemantik zur Berechnung der Daten erweitert. Dabei wird die Programmsemantik, die überwiegend durch die funktionalen Einheiten des verwendeten Prozessors erzeugt wird, mittels einer impliziten Darstellung für den SAT-Solver effizient dargestellt, ähnlich der klassischen Hardwareverifikation, wie es z.B. bei BMC der Fall ist.

Der hardwareabhängige Softwareverifikations-Ansatz kombiniert die explizite pfadorientierte Darstellung für die Software mit der impliziten Darstellung der Programmse-

mantik, repräsentiert als ein Hardwaremodell.

Das resultierende Gesamtmodell ist die sogenannte *Programmnetzliste* (PN). Sie beinhaltet alle relevanten Informationen für die Verifikation der funktionalen Korrektheit des Programms. Zusätzlich sind auch alle notwendigen Kommunikationsdetails im Modell vorhanden, die zur Interaktion zwischen der Software und der Hardware benötigt werden.

In den beiden folgenden Abschnitten werden die beiden Phasen genauer dargestellt.

3.1 Phase 1: Generierung des Programmflusses

In dieser Phase wird ein Control Flow Graph (CFG), der die Struktur und die Instruktionen des Assembler-Programms beinhaltet, in einen entsprechenden Execution Graph umgewandelt. Der CFG selbst wird vorher mittels eines Parsers erstellt, der die Instruktionen aus dem Maschinencode extrahiert. Potentiell sind nicht alle relevanten Kanten in dem erstellten CFG enthalten. So werden Kanten, die durch Sprünge im Code entstehen, vom Parser nicht erfasst, da sie nicht ohne eine Pfadanalyse berechnet werden können. Diese Lücken im Programmfluss des CFG können mittels der in Kapitel 4 beschriebenen Techniken geschlossen werden. Hier wird zum Verständnis der Grundidee vereinfachend davon ausgegangen, dass der CFG alle Ausführungsmöglichkeiten des Programms beinhaltet.

Abbildung 3.1 zeigt ein Beispiel eines CFG. Er beinhaltet eine Schleife mit einer bedingten Verzweigung im Knoten c, durch den die Schleife terminieren kann. Die Execution-Graph-Generierung beginnt beim Startknoten, der für die erste Instruktion des Programms steht (Knoten a in Abb. 3.1). Dieser Knoten wird direkt in den Execution Graph (EXG) übernommen und ist somit bearbeitet. Anschließend werden seine Nachfolger gesucht. Sofern der Knoten nur einen Nachfolger hat, wird dieser direkt an den entsprechenden Knoten im EXG angehängt (Knoten: b), da das Programm mit dieser Instruktion fortfahren wird. Das geschieht solange, bis ein Knoten mit mehr als einem Nachfolger gefunden wird. Ein Knoten mit mehreren Nachfolgern kann für einen bedingten Sprung oder einen generellen Sprung mit mehreren möglichen Zielen stehen. Knoten c ist der erste Knoten mit mehreren Nachfolgern, der gefunden wird. Er verfügt über genau 2 Nachfolger: der eine Nachfolger für den Fall, dass die Bedingung erfüllt ist, der andere für den Fall, dass sie nicht erfüllt ist.

Da der Nachfolger jetzt nicht mehr eindeutig ist, muss geprüft werden, welche Bedingungen – in Abhängigkeit von den vorher ausgeführten Instruktionen des Programms – überhaupt auftreten können. Gibt es für eine Bedingung eine gültige Belegung der Variablenwerte, dann gilt der zu der Bedingung zugehörige Nachfolgerknoten als "*lebendig*". In Schritt 1 ist d schon als lebendig untersucht worden. Nach Schritt 2 ist ebenfalls Knoten e als lebendig markiert und entsprechend der Pfad weiter ausgerollt worden. Im Falle, dass ein Nachfolger nicht lebendig ist, wie zum Beispiel der Nachfolger von c in Schritt 3, wird der Nachfolger nicht in den EXG eingefügt und dieser Pfad entsprechend auch nicht weiter untersucht, sondern an dieser Stelle "gekappt" (engl.: *pruned*).

Die Überprüfung geschieht, indem der bis dahin abgerollte Execution Graph entspre-

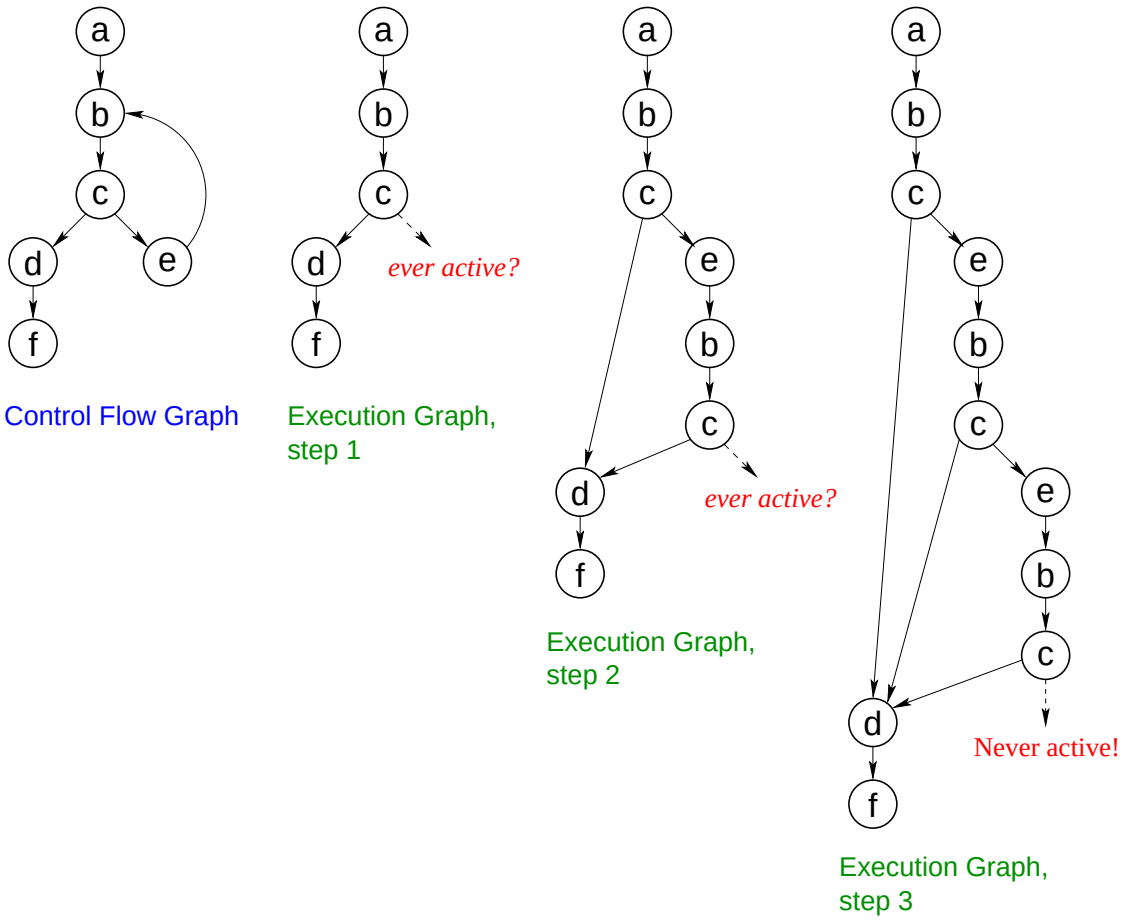

Abbildung 3.1: Abrollen eines CFG zu einem Execution Graph

chend der in Phase 2 beschriebenen Methodik in eine Programmnetzliste umgewandelt wird, auf der mit Hilfe eines SAT-Solvers Eigenschaften verifiziert werden. Diese einfachen Eigenschaften überprüfen, ob die Nachfolger-Knoten wenigstens in einem Programmpfad aktiv sein können. Die Eigenschaften können mit geringen Aufwand automatisiert geprüft werden.

In Schritt 2 der Abbildung 3.1 wird nach dem Knoten c kein neuer Knoten d eingesetzt, sondern stattdessen wird der Nachfolgerknoten mit einem im EXG bereits vorhandenen Knoten zusammengeführt. Dies geschieht, um den Execution Graph möglichst kompakt zu halten, da sonst ein Ausführungsbaum generiert würde, bei dem sich durch jede Verzweigung die Nachfolgerknoten verdoppeln würden. Damit diese Zusammenführung (engl.: *merging*) möglich ist, muss der zu vereinigende Knoten genau dieselbe Instruktion (an der gleichen Programmadresse) darstellen und zusätzlich darf er kein mittelbarer Vorgänger des Knotens sein, der gerade betrachtet wird. In Schritt 2 sind diese Bedingungen erfüllt, und entsprechend werden die Knoten zusammengeführt.

Das Programm wird mit diesem Verfahren solange weiter ausgerollt, bis entweder alle potenziellen Nachfolger nicht lebendig sind oder alle Pfade das Programmende erreicht

haben, wie es in Schritt 3 der Abb. 3.1 der Fall ist.

Der finale Execution Graph ist ein azyklischer gerichteter Graph, der alle möglichen Ausführungspfade des Programms besitzt und durch die Merging- und Pruning-Mechanismen dennoch kompakt ist. Dabei steht jeder Knoten für eine eindeutige Instruktion des Programms.

3.2 Phase 2: Generierung des Gesamtmodells für die Verifikation

Für die Verifikation von Software genügt die alleinige Beschreibung der möglichen Ausführungspfade nicht. Zu überprüfen ist das Verhalten des gesamten Programms inklusive der Datenoperationen, die das Programm durchführt. Dafür muss zum Execution Graph die Semantik des Programms hinzugefügt werden.

Als Repräsentation des Programmablaufs wird der erzeugte Execution Graph verwendet. Jeder Knoten entspricht einer Instruktion des Maschinenprogramms, die zusätzlich durch die gegebenen Parameter, wie zum Beispiel von der Instruktion verwendete Register, konfiguriert ist. Für jeden Knoten im EXG wird eine Instanz der zum Knoten gehörigen Instruktion erzeugt, und alle Instanzen werden entsprechend dem Graphen verbunden. Das resultierende Modell beschreibt das Verhalten des Prozessors, dargestellt durch die Instruktionen, kombiniert mit den möglichen Ausführungen des Programms gegeben durch den EXG. Wie die Instanzen der Instruktionen aufgebaut sind und wie genau diese verbunden werden, wird im folgenden Kapitel 3.2.1 gezeigt. Die ausführliche Beschreibung der Zellen und die Herleitung des Modells beschreibt Kapitel 4.

Das in den zwei Phasen erzeugte Gesamtmodell stellt das Berechnungsmodell für die eigentliche Verifikationsaufgabe dar. Die Verifikation selbst kann mit einem klassischen Eigenschaftsprüfer (engl. Property Checker) oder jedem anderen Hardware-Verifikationstool durchgeführt werden.

3.2.1 Modellierung der Programmsemantik

Jeder Knoten im EXG repräsentiert eine Instruktion aus der Instruction Set Architecture (ISA). Ein Prozessor unterstützt eine bestimmte Instruction Set Architecture und implementiert ihr gesamtes Verhalten. So besteht der Prozessor aus vielen Logik-Operationen, die für die Realisierung aller Instruktionen notwendig sind, die aber nicht für jede einzelne Instruktion benötigt werden. Wenn beispielsweise der Prozessor über einen dedizierten Hardwaremultiplizierer verfügt, wird dieser bei einer Addition nicht benötigt. Würde nun für jede Instruktion die komplette Logik des Prozessors ins Modell für die Verifikation übernommen, würde das Modell unnötig komplex werden.

Anstatt das komplette Verhalten der CPU für jede Instruktion zu übernehmen, ist es sinnvoll, das spezifische Verhalten des Prozessors für die jeweilige Instruktion einzeln zu modellieren. Dabei entsteht ein Satz von Instruktionsmodellen, die sogenannten *Instruktionszellen*. Jede Zelle enthält nur die notwendigen Teile der Prozessor-Logik, um

die konkrete Instruktion zu modellieren. Dass die Instruktionszelle das genaue Verhalten der CPU für eine bestimmte Instruktion implementiert, lässt sich mit klassischen Verifikationsmethoden wie dem Hardware-Property-Checking beweisen. Der Aufbau einer Instruktionszelle wird stark durch die Funktionsweise des Prozessors geprägt und muss für jede ISA individuell erstellt werden.

Um die Lesbarkeit der Beispiele zu verbessern, wird im Folgenden die Betrachtung auf Prozessoren beschränkt, die eine Load-Store-Architektur besitzen, in der Lese- und Schreibzugriffe auf den Speicher nur von speziellen Lese- und Schreiboperationen durchgeführt werden. Die Modellierung der Zellen funktioniert aber auch für komplexere Instruktionen, wie sie zum Beispiel bei CISC-Architekturen vorkommen.

Wenn eine einzelne Instruktion des Maschinenprogramms betrachtet wird, modifiziert diese Instruktion den aktuellen Zustand des Programms. Dieser Programmzustand setzt sich aus den Registern innerhalb der CPU und den verwendeten Variablen im Arbeitsspeicher zusammen. Die Register der CPU bilden den Architekturzustand. Dazu gehören neben den Datenregistern ebenfalls alle Register, die aus Sicht des Programms bzw. des Programmierers sichtbar sind oder direkt beeinflusst werden können.

Da der Execution Graph alle Ausführungspfade in sich vereint, das Programm aber in einem konkreten Fall nur einen Pfad ausführt, wird noch ein zusätzliches Signal zum Programmzustand hinzugefügt, welches als aktiv markiert wird, wenn der Programmzustand und der dazugehörige Knoten auf dem aktiven Pfad liegen. Das Signal heißt Aktives-Bit (engl.: *active bit*).

Die Instruktionen lassen sich in zwei Gruppen unterteilen: Die Instruktionen, die ausschließlich auf den Daten arbeiten und die Instruktionen, die den Programmfluss modifizieren können. Zur ersten Gruppe gehören i.A. die arithmetischen, logischen Operationen und modifizierende Lese- und Schreibzugriffe. Die pfadmodifizierenden Instruktionen sind Verzweigungsinstruktionen, wie zum Beispiel ein bedingter Sprung.

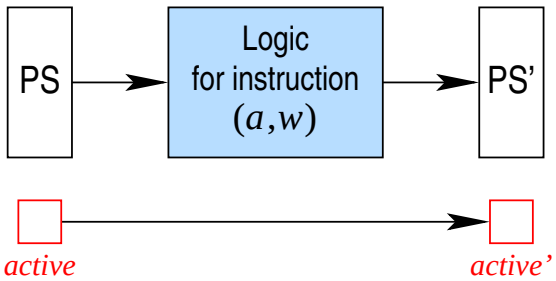

Abbildung 3.2: Instruktionszelle: Nicht-verzweigende Instruktion

Eine Instruktionszelle für eine nicht-verzweigende Instruktion, wie z.B. die Addition als eine arithmetische Operation, sieht aus wie in Abbildung 3.2. Sie hat als Eingabe den aktuellen Programmzustand und als Ausgabe den neuen Programmzustand nach der Ausführung der Instruktion. In der Zelle selbst befindet sich die Modellierung der eigentlichen Operation einschließlich der Modifikationen des Programmzustand. Da die Instruktion keine Verzweigungen ermöglicht und somit einen einzigen Nachfolger besitzt,

gibt es nur einen Ausgang mit dem neuen Zustand. Das zusätzlich in den Programmzustand aufgenommene Aktive-Bit muss daher nur vom Eingang zum Ausgang propagiert werden. Damit wird der Ausgangszustand aktiv, wenn der Eingangszustand zu Beginn der Instruktion auch auf dem aktiven Pfad lag. Eine zusätzliche Logik ist bei einer nicht-verzweigenden Instruktion nicht notwendig.

Listing 3.1: Instruktionszelle: Addition (ADD)

```
1  use work.proc_pack.all;
   use work.numeric_bit.all;
   -- Instruction ADD
   entity inst_add is
5  generic(Rm: integer;
     Rn: integer;
     ex_cycles: integer := cycles_simple_inst);
   port(signal s: in arch_state_type;
        signal z: out arch_state_type);
10 end inst_add;

   architecture arch of inst_add is
   begin
   process(s)
15 begin
     z <= s;
     z.regfile(Rn) <= bit32(unsigned(s.regfile(Rm))+unsigned(s.regfile(
        Rn)));
     z.cycles <= s.cycles + ex_cycles;
   end process;
20 end arch;
```

Der interne Aufbau der einzelnen Instruktion ist prozessorspezifisch. Das Listing 3.1 ist ein Beispiel einer in VHDL-implementierten ADD-Instruktionszelle eines abstrakten Prozessors. Sie definiert die Ein- und Ausgangszustände als Portsignale S und Z und die Parameter der Instruktion in Form von VHDL *generics*. Der Hauptteil beschreibt, welche Operationen durchgeführt werden. Entsprechend können alle weiteren arithmetischen Instruktionen implementiert werden. Auch alle logischen Operationen werden so umgesetzt. Die Lade- und Speicheroperationen sind maßgeblich für die Modifikation des Programmzustandes und enthalten wenige interne Operationen, dennoch ist deren Entwurf sehr ähnlich.

Kontrolloperationen, die den Programmfluss modifizieren können, verfügen über mehrere mögliche Nachfolger. Ein bedingter Sprung zum Beispiel verfügt über zwei Nachfolger. Die entsprechende Instruktionszelle verfügt somit ebenfalls über zwei Nachfolger und implementiert zwei entsprechende Ausgänge für den Programmzustand, wie in Abbildung 3.3 dargestellt. Zusätzlich muss entschieden werden, welche der beiden Nachfolger aktiv sein soll. Das geschieht wieder über die zugehörigen Active-Bits. Die Entscheidungslogik ist durch die Spezifikation der Instruktion vorgegeben und auch in der eigentlichen Instruktionszelle der Sprung-Instruktion implementiert. Diese Logik kann mittels eines Steuersignals einen Demultiplexer ansteuern, der das Active-Signal korrekt

an die entsprechenden Ausgänge propagiert, damit nur ein Pfad gleichzeitig aktiv werden kann.

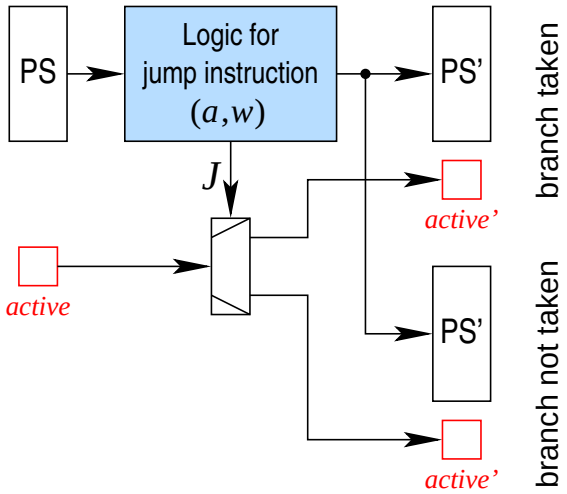

Abbildung 3.3: Instruktionszelle: Bedingter Sprung

Diese Menge von Instruktionszellen, welche für alle verwendeten Instruktionen einmalig generiert werden muss, kann für beliebige Programme verwendet werden, die auf der gleichen Prozessorarchitektur ausgeführt werden.

Die Gesamtnetzliste ist durch den gegebenen Execution Graph und die Instruktionszellen unkompliziert zu erstellen. Der Graph dient dabei als Vorlage für die finale Struktur. Jeder Knoten im Graph wird nun durch die entsprechende Instruktionszelle ersetzt. Die Kanten des Graphen geben die Verbindungen der Programmzustände zwischen den Instruktionszellen wieder. So kann der Graph eins zu eins in die Programmnetzliste übersetzt werden, wie in Abbildung 3.4 gezeigt. Wenn zwei oder mehr Verzweigungen durch *Merging* wieder zusammengefügt werden, muss beachtet werden, dass mehr als ein treibendes Signal existiert. Um zu entscheiden, welches Signal das richtige ist, wird wieder das Active-Bit der ankommenden Signale betrachtet. Ein erweiterter Multiplexer (Abb. 3.4) propagiert das ankommende Signal, welches das Active-Bit gesetzt hat, an den Eingang der nachfolgenden Instruktionszelle weiter. Da immer nur ein Pfad im Programm gleichzeitig aktiv sein kann, gibt es an diesem Multiplexer auch maximal ein Eingangssignal mit einem aktiven Active-Bit. Dieser Multiplexer wird Merge-Zelle genannt.

Das resultierende Gesamtmodell, die Programmnetzliste, ist eine Netzliste, die nicht mehr ausgerollt werden muss, wie es zum Beispiel bei BMC der Fall ist. Das Berechnungsmodell für das Gesamtverhalten ist, wie eine ausgerollte BMC-Netzliste, ein rein kombinatorisches Modell, auf dem mittels Standardwerkzeugen der Hardwareverifikation Beweise geführt werden können. Das Zeitverhalten der Software ist dabei in einer abstrakten Darstellung bei den einzelnen Zuständen zwischen den Instruktionszellen zu sehen. Da der Execution Graph alle Ausführungspfade der Software enthält, beinhaltet auch die Programmnetzliste alle möglichen Ausführungen des Programms. Die Programmnetzliste ist durch die Vereinigung von Pfaden (Merging) und die Erkennung von toten Pfaden

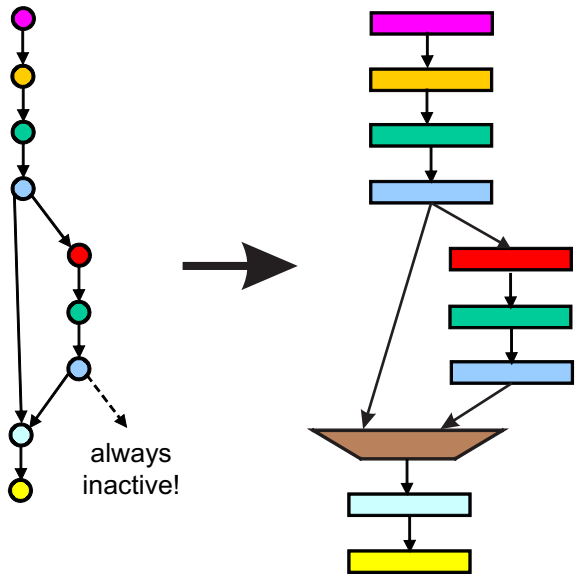

Abbildung 3.4: Gesamtnetzliste Generierung: EXG in PN übersetzt.

(Pruning) ebenfalls eine kompakte Repräsentation des Programms.

Diese Programmnetzliste kann als Grundlage für die Verifikation verwendet werden, um das eigentliche Verhalten des Programms zu überprüfen. Sie ist eine Beschreibung, wie sich der Maschinencode auf der CPU verhält. Eine Stärke des Modells ist diese Nähe zur Hardware, wodurch der Aspekt der Interaktion der CPU mit seiner Umgebung sehr gut betrachtet werden kann. Dieser Ansatz ist als Grundlage für die Modellierung der Kommunikation zwischen der Hardware und der Software besonders gut geeignet.

3.3 Interrupt-Modellierung

Die Programmnetzliste soll auch hardwarenahe Software beschreiben. Da der Interrupt-Mechanismus ein wichtiger Synchronisations-Mechanismus für die Kommunikation zwischen Hardware und Software auf einem Prozessor ist, muss ein Modell, welches Kommunikation zwischen HW und SW unterstützt, ebenfalls Interrupts modellieren können. Eine Schwierigkeit ist, dass ein Interrupt Request ein unerwartetes externes Ereignis ist, welches nicht genau vorhergesagt werden kann und gesondert behandelt werden muss. Zusätzlich zur zeitlichen Unwägbarkeit wird das Gesamtverhalten der Software durch das gemeinsame Verhalten des Hauptprogramms und der Interrupt-Service-Routinen erzeugt.

Der Ansatz für die Modellierung von Interrupts beruht auf der Möglichkeit, einzelne Programmnetzlisten zu einer größeren kombinierten Programmnetzliste zusammenzusetzen. Dabei können sie an unterschiedlichen Stellen verbunden werden.

Der Ansatz ist, die Programmnetzlisten für die Interrupt Service Routinen (ISR) und das Hauptprogramm, welches mit den ISRs kommuniziert, separat – nach dem oben vorgestellten Verfahren – zu erzeugen und dann in Abhängigkeit der gegeben Kommunikati-

on zusammenzusetzen.

Ein naiver Ansatz wäre, überall dort, wo ein Interrupt das Hauptprogramm unterbrechen kann, eine Instanz dieser ISR einzufügen. Das wäre in der Regel zwischen je zwei Instruktionen im Hauptprogramm, was eine starke Vergrößerung der Programmnetzliste zur Folge hätte.

Die Abhängigkeiten zwischen dem Hauptprogramm und der ISR sind aber nicht willkürlich gegeben, sondern schon bei der Erstellung der Software geplant worden. Da das Programm an jeder beliebigen Stelle in der Ausführung durch den Interrupt unterbrochen werden kann, sollte der Interrupt nicht beliebig Register manipulieren, welche das Hauptprogramm benutzt, da dies zu Fehlverhalten führen könnte. Deshalb sollte bei jeder Unterbrechung durch einen Interrupt der Architekturzustand des Hauptprogramms erhalten bleiben. Diese Unterbrechung wirkt sich nur aufs absolute Zeitverhalten aus, aber nicht auf die Ausführungsreihenfolge des Programms selbst, und ist somit quasi transparent für das Hauptprogramm. Da die beiden Programmteile dennoch miteinander kommunizieren sollen, wird eine Schnittstelle definiert wie zum Beispiel ein fester Shared-Memory-Bereich, der zur Kommunikation verwendet wird.

Die gemeinsamen Speicherbereiche können durch reine Betrachtung der Lese- und Schreibzugriffe der jeweiligen Programmnetzlisten gut berechnet werden, da die Zugriffe auf dieselbe Speicheradresse erfolgen. Anhand dieser gemeinsamen Speicherbereiche können alle Zeitpunkte in den Netzlisten bestimmt werden, in denen ein Zugriff auf diesen Shared-Memory stattfindet. Zu diesen Zeitpunkten kann der Interrupt Einfluss auf das Hauptprogramm ausüben und umgekehrt.

Die genaue Funktionsweise und das exakte Zusammensetzen der Gesamtnetzliste wird in Kapitel 4.2 ausführlich beschrieben.

Ob ein Interrupt wirklich transparent ist, wie wir es für unsere Modellgenerierung annehmen, kann anhand der für den Interrupt generierten Programmnetzliste mittels einer automatisierten Eigenschaftsprüfung berechnet werden. Ist ein Interrupt nicht transparent, weist dies zumindest bei externen Interrupts auf eine mögliche Fehlerquelle hin, die der Entwickler gesondert überprüfen muss. Es kann vorkommen, dass dieses Verhalten vom Entwickler gewünscht ist, wie z.B. bei synchronen Interrupts – sogenannten Exceptions. Doch diese dienen weniger zur Behandlung von I/O-Kommunikation, auf die in dieser Arbeit der Fokus liegt. Der Transparenz-Beweis wird ebenfalls in Kapitel 4.5 im Detail erläutert.

3.4 Vergleich mit vorhandenen Ansätzen

Der in dieser Arbeit vorgestellte Ansatz ist ein dem Bounded Model Checking verwandtes Verfahren, in dem der Berechnungsgraph des Programms explizit und programmspezifisch ausgerollt wird, um das Modell zur Verifikation zu generieren. Der Datenfluss kann anhand des Modells vom SAT-Solver einfach berechnet werden.

In [NWSK11] und [MPPK17] wird das RTL-Modell des Computersystems inklusive des Programmspeicherinhaltes für die Modellgenerierung verwendet. Somit ist ei-

ne zusätzliche Verifikation, ob das Modell korrekt erstellt wurde, nicht notwendig. In der vorliegenden Arbeit und in [GKD06] hingegen wird ein Modell der Hardware erzeugt, das jeweils auf Korrektheit gegenüber dem Originaldesign oder Implementierung überprüft werden muss. Verglichen mit den Hardware/Sofware-Co-Verifikationsansätzen von [NWSK11], [MPPK17] und [GKD06] wird in dieser Arbeit bei der Erzeugung der Programmnetzliste zusätzlich eine Abstraktion der Hardware vorgenommen.

Prozessor und das Programm im Programmspeicher werden durch die Instruktionszellen und der Programmnetzliste (PN) ersetzt. Das hat den Vorteil, dass nur ausschließlich die Instruktionszelle mit der minimal notwendigen Hardware betrachtet werden muss, die in dem entsprechenden Zeitschritt von der Hardware ausgeführt wird. Das ist bei der PN durch das explizite Ausrollen über die Programmpfade möglich, da in jedem Schritt genau bekannt ist, welche Instruktion mit welchen Parametern ausgeführt wird. Im Gegensatz dazu muss beim Ausrollen der kompletten Hardware, wie z.B. in [NWSK11] und [MPPK17], – ob auf RT- oder SystemC-Ebene – die nächste Instruktion immer vom SAT-Solver berechnet werden. Somit ist beim ausgerollten Modell in den meisten Zeitschritten nicht klar, welche Instruktion ausgeführt wird. Daher muss der komplette Prozessor in jedem Zeitschritt instanziiert werden, wodurch das Modell, auf dem der Beweis durchgeführt werden soll, potentiell von einer größeren Komplexität ist als nötig.

Ein Vorteil des RTL-basierten Ansatzes von [NWSK11] ist, dass dieses Verfahren taktzyklengenau ist. Es kann die vorhandene RTL-Implementierung nutzen und die kompilierte Software mit geringem Aufwand in das Modell integrieren. Zusätzliche Hardware ist ebenfalls leicht integrierbar, wenn der HDL-Code zur Verfügung steht. Das gleiche Vorgehen ist bei dem SystemC-basierten Ansatz [GKD06] möglich, jedoch ist bis heute die überwiegende Mehrheit der digitalen Schaltungen – Prozessor und Peripherie – in VHDL, in Verilog oder in einer anderen Sprache auf RT-Ebene entworfen. Somit müssen die meisten Hardwaremodule als Modell in SystemC neu implementiert und auf Fehlerfreiheit überprüft werden. Bei [HTV+13], wird ähnlich wie bei [GKD06] ein abstraktes Hardwaremodell verwendet, welches jedoch in C implementiert ist.

Die Programmnetzliste kann je nach Art der verwendeten Instruktionszellen auch taktzyklengenau sein. Ein zusätzlicher Aufwand ist notwendig, um die unterschiedlichen Pfade zeitlich korrekt an die Peripherie zu koppeln. In [VSB+14] wird das Verbinden von der Programmnetzliste mit der Peripherie ausführlich beschrieben. Bei der Verwendung von zeitabstrakten Zellen, die hier verwendet werden, ist eine exakte zeitliche Kopplung nicht direkt möglich. Da die Kommunikation zwischen Prozessor und Peripherie aber mittels wohl definierter Schnittstellen erfolgt, ist eine Kopplung mit zusätzlicher Logik dennoch möglich. Als Schnittstelle fungiert der Bus und zum triggern der Events können die Interrupts der Peripherie verwendet werden. Somit unterstützt die PN grundlegend sowohl den Vorteil der RTL-Nähe wie bei Nguyen ([NWSK11]), als auch die Möglichkeit eher zeitabstrakt zu arbeiten. Die Verbindung von Hardware und Software in [MPPK17] unterscheidet sich von [NWSK11] und der PN. Es wird zwar weder Hardware noch C-Software abstrahiert, aber die Taktung der Hardware wird von der Software vorgegeben. Der Code muss um Trigger-Elemente für die Hardware erweitert werden, so dass die Hardware immer synchronisiert zur Software entsprechende Takte weiter läuft.

Der wesentliche Unterschied zwischen dem hier vorgestellten Verfahren und vielen aus dem Bereich der Softwareverifikation ist, dass die PN auf Maschinencode-Level und spezifischen Prozessoren arbeitet und nicht auf dem C-Code oder anderen höheren Programmiersprachen. So können die Auswirkungen bei der PN direkt auf Hardware-Level Bit-akkurat betrachtet und aussagekräftige Beweise in Bezug auf die von der Software angesprochene Hardware geführt werden.

Wenn der Level der Sprache für die zu verifizierende Software außer Acht gelassen wird und die Ansätze direkt verglichen werden, hat die Programmnetzliste weitere Vorteile.

Die Programmnetzliste ist eine ausgerollte Netzliste, die ohne weiteren Aufwand direkt in einen SAT-Solver übertragen werden kann. Im Vergleich dazu wird bei F-Soft [IYG+08] das Modell wie beim klassischen HW-BMC ausgerollt, so dass in jedem betrachteten Zeitschritt eine Instanz von jedem Block an Instruktionen entsteht und der SAT-Solver entscheiden muss, welcher Block aktiv wird. Das Modell von CBMC [CKY03] und die Programmnetzliste hingegen sind schon ein ausgerolltes Modell, so dass der SAT-Solver keine zusätzlichen Berechnungen für die richtigen Sequenzen benötigt werden. Dennoch müssen in allen drei Verfahren die Nachfolger an bedingten Verzweigungen berechnet werden. Beim CBMC und der PN während der Modellgenerierung, bei F-Soft erst später bei der eigentlichen Verifikation. Die Aussagen über CBMC gelten ebenfalls für die Ansätze, die CBMC als Verifikationstool einsetzen, wie zum Beispiel [MPPK17] oder auch bei einem Teil bei [HTV+13].

Bei Calysto und der PN wird mittels der Verzweigungsbedingung entschieden, welche Verzweigung aktiv wird. Bei CBMC wird die Verzweigungsbedingung erst betrachtet, wenn eine Variable berechnet wird, die Werte aus verschiedenen vereinten Ausführungspfaden besitzen kann. Dies ist ein Vorteil von CBMC, da nicht alle Verzweigungsbedingungen berechnet werden müssen, sondern nur diejenigen, die sich direkt auf Variablen auswirken.

Der Vorteil von CBMC wird dadurch erreicht, dass es keine explizite Zeitinformation mehr im Modell gibt. Durch die Reihenfolge der Indexierung ist die Ausführungsreihenfolge implizit noch vorhanden, doch diese kann nicht ohne Weiteres zur Verifikation verwendet werden. Hinzu kommt, dass Schleifen beschränkt ausgerollt werden können, was später noch erläutert wird.

Bei hardwarenaher Software ist es allerdings für die Kommunikation oft entscheidend, die Informationen in der richtigen Reihenfolge zu übermitteln. In CBMC lässt sich das nur durch Instrumentieren des Codes und den manuellen Einbau eines Zählers erreichen, der während der Ausführung des Programms die Anzahl der Zugriffe aufaddiert. In der Programmnetzliste ist es auch notwendig zusätzliche Logik einzusetzen, die allerdings automatisiert erzeugt werden kann.

Ein weiterer Nachteil von CBMC und auch Calysto [BH08] gegenüber der Programmnetzliste ist, dass die zu beweisenden Eigenschaften direkt in den Code eingefügt werden. Das vereinfacht zwar das Schreiben von Eigenschaften, schränkt aber auch die Mächtigkeit der sogenannten Assertions ein. Da sie im Original C-Code eingefügt werden, können sie sich ausschließlich auf Variablen aus dem jeweiligem Gültigkeitsbereich (*scope*)

beziehen. Somit ist ein Bezug zwischen Werten einer Variablen an verschiedenen Zeitpunkten nur dann möglich, wenn der Code weiter instrumentiert wird und zusätzliche Variablen erzeugt werden, was aber einer Veränderung des Programms entspricht.

Zwei substantielle Vorteile der Programmnetzliste lassen sich hier herausstellen: Erstens sind komplexe Eigenschaften ebenfalls zwischen verschiedenen Zeiten von ein und derselben Variablen möglich – oder noch allgemeiner: Die Eigenschaft kann sich auf jede Stelle im Programm beziehen und diese Stellen auch in Beziehung zueinander setzen. Der zweite Vorteil ist, dass der Code nicht instrumentiert werden muss. Somit ist eine formale Verifikation des Original Codes möglich. Es kann so ausgeschlossen werden, dass die Instrumentierung den Code so beeinflusst hat, dass neue Fehler entstehen oder Fehler überdeckt werden. Bei CBMC werden durch das Verfahren alle Funktionen inlined, d.h. wird eine Funktion an zwei Stellen aufgerufen, wird sie auch zweimal in den Code ausgerollt, was zu einem größeren Modell führen kann. In der PN wird diese Funktion dagegen nur dann mehrfach ausgerollt, wenn sie vom selben Pfad mehrfach aufgerufen wird. Das kann die PN vereinfachen, erfordert dafür aber einen Mehraufwand für die Synchronisation, der aber durch das Active-Bit und die zeitliche Ordnung, die wir betrachten wollen, schon in der PN vorhanden ist.

Sowohl bei Calysto, CBMC als auch die Erzeugung der PN versuchen die Tools einen korrekten Umgang mit Schleifen durchzuführen. Während CBMC versucht, die Anzahl an Iterationen durch eine Schleife beim Erstellen des Modells zu berechnen, wird bei der Erzeugung der Programmnetzliste die Verfikationsengine genutzt, um während des Abrollens zu entscheiden, ob es zusätzliche Iterationen gibt. Das Verfahren von CBMC ist schneller, allerdings werden potentiell nicht alle Abbruchbedingungen korrekt ermittelt. Bei der PN dagegen werden die Schleifen entsprechend den tatsächlichen Bedingungen ausgerollt und der Schleifeninhalt wird auch immer vollständig berücksichtigt. Hängen die Bedingungen zur Beendigung einer Schleife von externen Variablen ab, können sowohl CBMC und die PN keine Obergrenze erkennen und der Verifikateur muss dies gesondert behandeln. Das Vorgehen wird in Kapitel 4.2.6 erörtert.

Das Tool Calysto hingegen versucht mittels einer speziellen Methodik die Schleife nur einmal zu durchlaufen und dabei alle Möglichkeiten zu berechnen. Das kann je nach Programm auch dazu führen, dass der Inhalt der Schleife nicht vollständig überprüft wird (vergl. 2.6.2). Diese Einschränkungen hat die Programmnetzliste – wie vorher beschrieben – nicht.

Symbolic Execution durchläuft den Code auf eine Weise – ausgehend von dem CFG des Programms – die mit den Pfaden im Execution Graph vergleichbar ist. Dabei werden auch Optimierungen bezüglich ausführbarer Verzweigungen und des Zusammenführens von unterschiedlichen Pfaden durchgeführt. Ähnlich zur PN ist der kompositionale Ansatz von [PV09]. Somit sind die erzeugten Graphen von ihren Eigenschaften ähnlich.

Der wesentliche Unterschied zwischen Symbolic Execution und der PN ist vor allem, dass die symbolischen Formeln, die die Zustände beschreiben, explizit formuliert werden müssen. Das führt zu einem höheren Aufwand bei der Erstellung des Ausführungsgraphen. Um Speicher zu sparen wird oft nicht der gesamte Graph gespeichert, sondern nur für jeden separaten Pfad (oder auch mehrere Pfade, wenn sie sich noch nicht verzweigt ha-

ben) ein einzelner Zustand mitgeführt, der an einer Verzweigung dupliziert wird. Bei der Programmnetzliste wird die Darstellung der Zustandsmengen implizit durch die Aneinanderreihung der Instruktionszellen erzeugt. Das erleichtert den Aufwand zur Generierung des Execution Graphen.

Bei Symbolic Execution werden die Zustandsmengen symbolisch beschrieben. Die möglichen Wertebelegungen werden dabei explizit in Wertemengen oder Formeln aufgeschrieben, die den korrekten Werteinhalt bzw. eine Überapproximation der möglichen Werte wiedergeben. Diese Überapproximation kann zu einem Präzisionsverlust führen und somit eventuell auch Eigenschaften bei der Verifikation als Falsch markieren, obwohl sie korrekt sind.

Die aufwendige explizite symbolische Beschreibung der Zustände in Symbolic Execution ist ein begrenzender Faktor dieser Methodik.

Dies wird in [PV09] zum Anlass genommen um das Modell in kleinere Stücke aufzubrechen, in dem Unterprogramme separat betrachtet werden. Nur die finalen symbolischen Formeln der Unterprogramme werden im ursprünglichen wieder aufgenommen um weiter zurechnen. Diese Modularisierung des Gesamtmodells ist auch mit der Programmnetzliste möglich.

Ein weiterer Ansatz die Komplexität zu reduzieren wird in [AEO+08] verwendet: Sie versuchen Pfade ähnlich wie bei der Erzeugung der Programmnetzliste wieder zusammen zu führen. Dabei werden in [AEO+08] nur isomorphe Untergraphen wieder vereint und nicht beliebige Knoten im Graphen, die dieselben Instruktionen abbilden. Somit ist die Programmnetzliste in der Regel kompakter oder im schlechtesten Fall genauso kompakt wie das Modell von [AEO+08]. Dafür muss der SAT-Solver bei der PN herausfinden, welcher Pfad möglich ist, doch dank des eingeführten Aktive-Bits ist das mit geringem Rechenaufwand möglich. Durch das Zusammenführen von Pfaden in der PN werden auch keine komplexen Formeln generiert, sondern durch die implizite Darstellung des Datenflusses wird die Erreichbarkeit von Pfaden "on-the-fly" von dem SAT-Solver berechnet.

In [SNBB11, KLM+15, VHHK15] werden jeweils ein Verfahren vorgestellt, um Interrupts zu handhaben. Das Verfahren von [SNBB11] basiert darauf, im nicht ausgerollten Kontrollfluss-Graphen die Stellen zu ermitteln, an denen ein Interrupt notwendig ist. Dabei verfolgen sie den umgekehrten Ansatz, verglichen zu dem hier vorgestellten Verfahren: Zunächst wird – ähnlich wie bei der in Kapitel 3.3 erwähnten naiven Sichtweise – davon ausgegangen, dass der Interrupt-Request zwischen jeder Instruktion passieren kann, und an jede dieser Stellen einen ISR ins Modell eingefügt. Danach werden diese wieder nach und nach entfernt, sofern das möglich ist. Das Arbeiten auf dem CFG hat zur Konsequenz, dass Schleifen dabei gesondert behandelt werden müssen und keine Optimierungen über die Grenzen der Schleifen erfolgen können. Bei der PN hingegen wird auf dem ausgerollten Execution Graph gearbeitet. Die Schleifen bilden keine Einschränkungen mehr, wodurch weniger Ausnahmen behandelt werden müssen. Es werden in der PN nicht an jeder Stelle Interrupt Service Routinen eingefügt, sondern nur an den tatsächlich notwendigen Stellen, die anhand der statischen PNs einfach errechnet werden können.

[KLM+15] und [VHHK15] arbeiten im Unterschied zu [SNBB11] und zur Pro-

grammnetzliste auf C-Level. Beide haben gemeinsam, dass sie Interrupts auf Stellen reduzieren, an denen die Interrupts mit dem Programmen interagieren. Dieses Ziel ist identisch zu [SVF+13], das als Grundlage für diese Arbeit dient (vgl. Kapitel 4.5.2). Der Weg, wie die Informationen gewonnen werden, unterscheidet sich. Während in [VHHK15] auf dem Original Code eine Reduktion von Interrupt-Aufrufen errechnet wird, wird bei [SVF+13] und [KLM+15] jeweils auf dem ausgerollten Programm die notwendigen Interrupt Interaktionen berechnet.

Weiterhin wird bei [VHHK15] auch der Abstand von periodischen Interrupts berücksichtigt. Das geschieht durch die Implementierung eines Zählers für den Interrupt und einer Funktion die ihn inkrementiert. Wenn der Zähler einen Schwellwert überschreitet, kann der Interrupt auftreten, ansonsten wird er unterbunden. Funktion und Zähler werden mit in die Verifikation eingebunden. Der Startwert des Zählers wird dabei auf einen beliebigen Wert kleiner gleich dem Schwellwert gesetzt.

In [SVF+13] wird zusätzlich ein potenzieller Abstand zweier Interrupts berücksichtigt. Diese Zusatzannahme gilt jedoch auch für nicht periodische Interrupts. Das Wissen über die Auftrittshäufigkeit basiert auf Informationen die von Außerhalb gewonnen werden muss. Dadurch ist der Ansatz in dieser Arbeit universeller, aber unter Umständen weniger präzise als der Ansatz in [VHHK15]. Zusätzlich unterscheidet sich die Form des für das Modell betrachteten Interrupts. Während bei [SNBB11] davon ausgegangen wird, dass der Interrupt die Register des Architekturzustands das Hauptprogramms modifiziert, beschränkt sich der in dieser Arbeit vorgestellte Ansatz, [KLM+15] und [VHHK15] auf sogenannte transparente Interrupts (vergl. 4.5.1). Die Annahme von transparenten Interrupts ist eine Einschränkung, die von klassischen Interrupt Service Routinen unterstützt wird, sofern es keine Exception-Handler sind, die gezielt Fehler im Hauptprogramm korrigieren sollen. Sollte ein Interrupt-Handler dennoch direkt in den Registern des Hauptprogramms Werte modifizieren, setzt das Zusammenspiel ein spezialisiertes System voraus, auf dem nicht mehrere Tasks laufen können, da der Handler nicht wissen kann, welcher Task im Falle eines Interrupts unterbrochen wurde.

Kapitel 4

Modellgenerierung

Der Schwerpunkt dieses Kapitels liegt auf der Herleitung und Generierung der Programmnetzliste.

In Kapitel 4.1 wird die Herleitung des Modells ausführlich beschrieben und gezeigt, warum die Programmnetzliste eine korrekte und kompakte Darstellung des Programms ist.

Die schon in Kapitel 3 eingeführte und allgemein behandelte Modellgenerierung wird in Kapitel 4.2 noch einmal aufgegriffen und detailliert beschrieben.

Dabei liegt in diesem Teilkapitel der Schwerpunkt auf den Details, die bei der Erstellung der Programmnetzliste benötigt werden und zu beachten sind. Sie wurden in Kapitel 3 ausgespart, um die Grundidee zu erklären. Diese Details sind aber für die Modellgenerierung notwendig und werden auch zur Optimierung der Implementierung genutzt.

In Kapitel 4.3 werden die Instruktionszellen und deren Generierung behandelt.

Kapitel 4.4 zeigt, wie sich der Speicher modellieren lässt, damit die Kommunikation mit der Peripherie und der Shared-Memory im Modell berücksichtigt werden können.

Kapitel 4.5 beschreibt schließlich die Modellierung von interrupt-basierter Software und wie das Gesamtmodell aus der Programmnetzliste zusammengesetzt wird. Dabei wird auf die Probleme der Interrupt-Modellierung eingegangen und erörtert, wie das Gesamtmodell diese löst.

4.1 Herleitung des Modells

Die Idee ist ein kombiniertes Modell, welches das gemeinsame Verhalten der hardwarenahen Software und des verwendeten Prozessors beschreibt, auf dem die Software ausgeführt wird.

Die Herleitung des Modells basiert auf dem Ansatz des endlichen Abrollens von Transitionen eines endlichen Automaten, der das kombinierte Hardware-Software-Verhalten modelliert. Dieser Automat setzt sich aus einem Prozessor und dem zugehörigen Programm zusammen. Der Prozessor ist abstrakt auf ISA-Ebene beschrieben. Er arbeitet das Programm streng sequenziell ab.

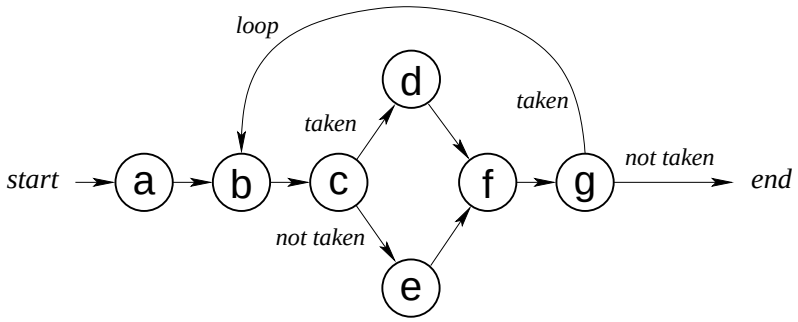

Abbildung 4.1: Beispiel CFG aus Kapitel 2.4.2

Nachfolgend soll gezeigt werden, wie die Elemente der Programmnetzliste als Ergebnis von verschiedenen Operationen und Optimierungen angesehen werden können, ohne dass diese das Verhalten des ausgeführten Programms verändern. Zu den Operationen und Optimierungen gehören unter anderem Logik-Optimierung, Logik-Duplizierung, Verschieben von Multiplexern, Vereinfachungen mittels Konstanten-Propagierung und Knoten-Zusammenführung. Nach der Herleitung wird ersichtlich, warum das konstruierte Modell sowohl korrekt als auch kompakt ist.

Als Ausgangspunkt wird ein Maschinenprogramm verwendet, welches mittels eines Kontrollflussgraphen (CFG) dargestellt wird.

Zur Generierung des Modells besteht jeder Basisblock des CFG zunächst aus nur einer einzelnen Maschinen-Instruktion. Nach der Generierung der Programmnetzliste können die einzelnen Instruktionen wieder zu größeren Basisblöcken zusammengefasst werden.

Abbildung 4.1 zeigt ein Beispiel eines solchen CFG eines Maschinenprogramms. Die Buchstaben a bis g stehen dabei jeweils nur für eine einzelne Maschinen-Instruktion.

Eine solche Maschinen-Instruktion setzt sich im Wesentlichen aus einer Adresse und einem Instruktionswort zusammen. Das Instruktionswort kodiert nicht nur den Typ der Instruktion, sondern enthält auch die notwendigen Parameter zur Konfiguration dieser speziellen Instruktion.

Der CFG kann auch als abstrakter *endlicher Automat* (*engl. Finite-State Machine, kurz FSM*) verstanden werden, der das Verhalten des kombinierten Hardware-Software-Systems steuert.

Als Basis unserer Herleitung dient das kombinierte Hardware-Software-Modell, wie es in Abbildung 4.2 gezeigt wird. Es beinhaltet alle Elemente in einer abstrakten Sichtweise, die bei Prozessoren und Programmen auch vorhanden sind. Das zu betrachtende Maschinenprogramm ist dabei im Programmspeicher gespeichert. Die Instruktionslogik bildet die Instruktionen des Prozessors ab und beschreibt damit unter anderem auch das Verhalten der Arithmetisch-Logischen-Einheit eines Prozessors. Als Vereinfachung wird angenommen, dass die Position des Programms im Speicher fest und von vornherein bekannt ist.

Erweiterungen der Methodik zur Code-Relokalisierung [HP07], wie beim Linken und Laden eines Programms üblich, sind gleichbleibend und werden nicht weiter betrachtet.

Das abstrakte Hardware-Software-Modell setzt sich aus zwei endlichen Automaten

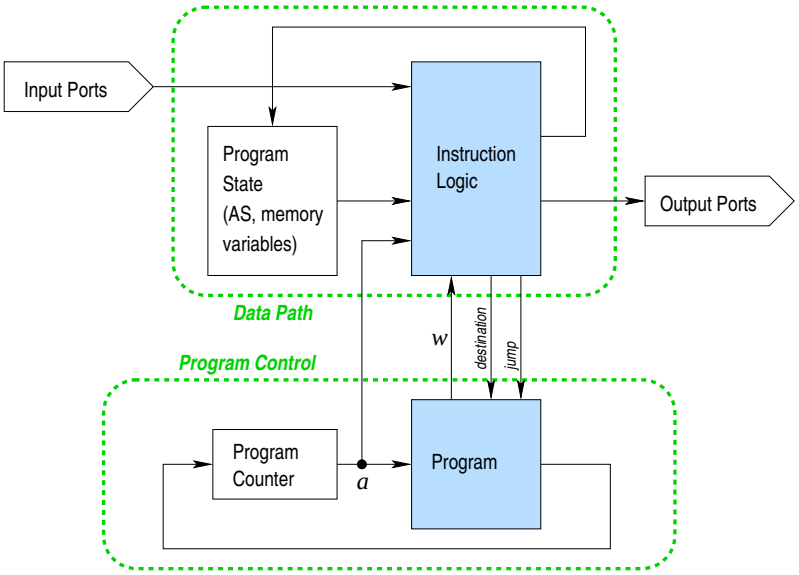

Abbildung 4.2: Abstraktes kombiniertes HW/SW Modell

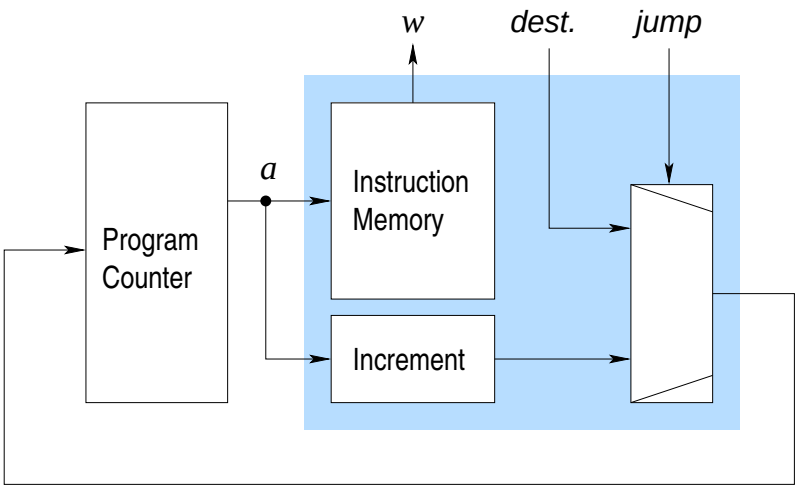

Abbildung 4.3: Programm Pfad FSM

zusammen. Der eine wird “Program Control” genannt und besteht aus dem eigentlichen Maschinenprogramm und dem Programmzähler (*program counter*, PC) des Prozessors, der definiert, welche Instruktion als nächste abgearbeitet wird. Als Eingänge verfügt diese Zustandsmaschine über ein Sprungsignal (jump) und Sprungzielsignal (destination). Das erste Signal zeigt an, ob ein Sprung stattfindet. Das zweite Signal gibt die Zieladresse, den neu berechneten PC, an. Findet kein Sprung statt, bewegt sich die FSM weiter sequentiell durch das Programm, indem der Wert des PC inkrementiert wird. Abbildung 4.3 zeigt eine detaillierte FSM der “Program-Control”-Einheit mit allen Komponenten und Signalen.

Der zweite Teil des Modells aus der Abbildung 4.2 ist der Datenpfad, der ebenfalls als ein endlicher Automat aufgefasst werden kann. Er enthält den Programmzustand (engl. *Program State* (PS)), die Instruktions-Logik (engl. *Instruction Logic*), und die Eingangs- und Ausgangsports.

Der Programmzustand beinhaltet alle Informationen, die über die einzelnen Zeitschritte hinweg gespeichert werden. Dabei unterteilt sich der Programmzustand in zwei Teile. Der erste Teil ist der sogenannte Architekturzustand (AS), der alle aus Sicht des Programmierers und des Programms sichtbaren Register des Prozessors enthält. Das sind neben den allgemeinen Registern aus dem Registersatz auch sichtbare Konfigurationsregister und Statusregister des Prozessors. Der zweite Teil enthält alle Variablen der Software, die als Speicherstellen im Arbeitsspeicher platziert sind, und auch alle weiteren für das Programm notwendigen Speicherstellen im Arbeitsspeicher, wie zum Beispiel den zum Programm zugehörigen Stack.

Die Eingangsports sind Speicherstellen, die von der Umgebung geschrieben und vom Programm gelesen werden können. Entsprechend dazu bilden Ausgangsports die Schreiboperationen des Programms auf Speicherstellen ab, die von der Peripherie gelesen werden können. Der Austausch von Informationen mit der Umgebung kann sowohl über spezielle Speicherbereiche in der Hardware als auch über Shared-Memory realisiert werden.

In dieser Arbeit wird davon ausgegangen, dass das betrachtete System Memory Mapped I/O unterstützt (vergl. Kap 2.3.1). Somit ist eine getrennte Unterscheidung an dieser Stelle nicht notwendig, da die Zugriffe auf beide Bereiche identisch sind.

Der Block “Instruction Logic” aus Abbildung 4.2 wird in Abbildung 4.4 detaillierter dargestellt und enthält eine Beschreibung der programmierbaren Hardware in einer Abstraktion, die für die übergreifende Verifikationsaufgabe geeignet ist.

Als Beschreibung kann zum Beispiel eine Register Transfer Language (RTL)-Beschreibung, eine System-Level-Beschreibung eines virtuellen Prototyps oder auch eine Instruction Set Architecture (ISA)-Level-Beschreibung eines Prozessors verwendet werden. Auch andere Beschreibungsebenen sind möglich, sofern bei der Abstraktion die einzelnen Instruktionen, die das Modell unterstützen muss, erhalten bleiben.

Im Folgenden wird gezeigt, dass die vorgestellte Methodik immer die Hardware-Beschreibung instanziiert, die zu einer spezifischen Instruktion des verwendeten Prozessors gehört. Deswegen wird in einer Pre-Processing-Phase je eine Instruktionszelle für jeden Befehl des Prozessors erzeugt, indem die Hardware auf die notwendige Funktionalität der entsprechenden Instruktion beschränkt und so vereinfacht wird. Dieser Prozess

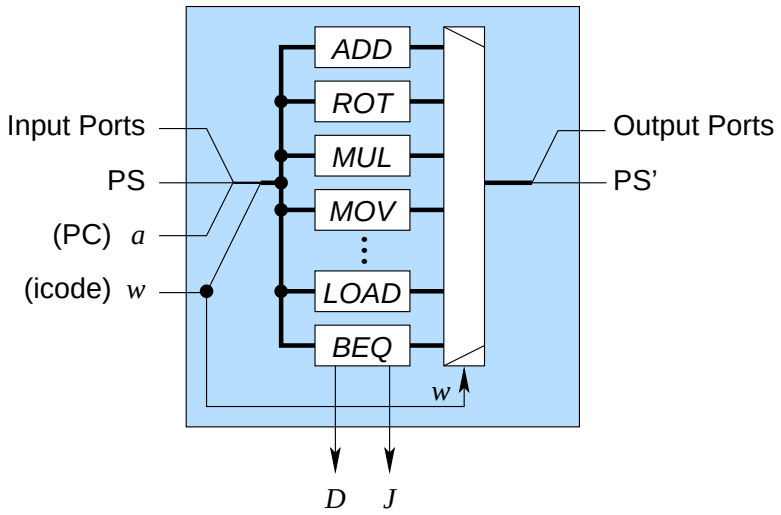

Abbildung 4.4: Datenpfad-Modell

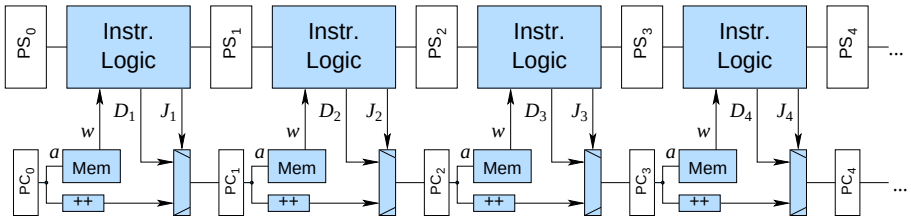

Abbildung 4.5: Ausrollen des abstrakten Modells, *IO wird nicht gezeigt*

muss für jeden Prozessor nur ein einziges Mal durchgeführt werden. Die dabei generierte Beschreibung in Form der Instruktionszellen kann für viele und beliebige zu verifizierende Programme verwendet werden.

Das Signal w aus Abbildung 4.2 stellt das Instruktionswort an der zugehörigen Programmadresse a dar. Es wird dazu verwendet, die entsprechende Instruktionszelle zu der durch den PC gegeben Programmstelle zu wählen. Das Ergebnis, der Nachfolgezustand, der von der gewählten Zelle generiert wurde, wird über Multiplexer wieder zurück auf den Programmzustand und/oder auf die Ausgangsports des Modells geschrieben.

Die für den Kontrollfluss zuständigen Instruktionen, wie zum Beispiel die bedingte Sprunginstruktion BEQ in Abbildung 4.4, unterscheiden sich von den anderen Instruktionen. Sie treiben zusätzlich die beiden Signale “jump” und “destination”, welche beide zurück zum Programm-Kontroll-Block führen und somit den Programmfluss beeinflussen können.

Im abstrakten Modell werden die beiden endlichen Automaten, der Kontrollpfad und der Datenpfad, mit demselben abstrakten Taktsignal synchronisiert. Dabei entspricht die Taktlänge des abstrakten Zeitgebers der Zeit, die benötigt wird, um eine Instruktion auszuführen. Das hier vorgestellte Modell für hardwareabhängige Software wird mittels eines speziellen Abrollens des abstrakten Modells erzeugt – basierend auf dem CFG des gegebenen Programms. In Standard Bounded Model Checking (BMC) 2.6.2 würde das

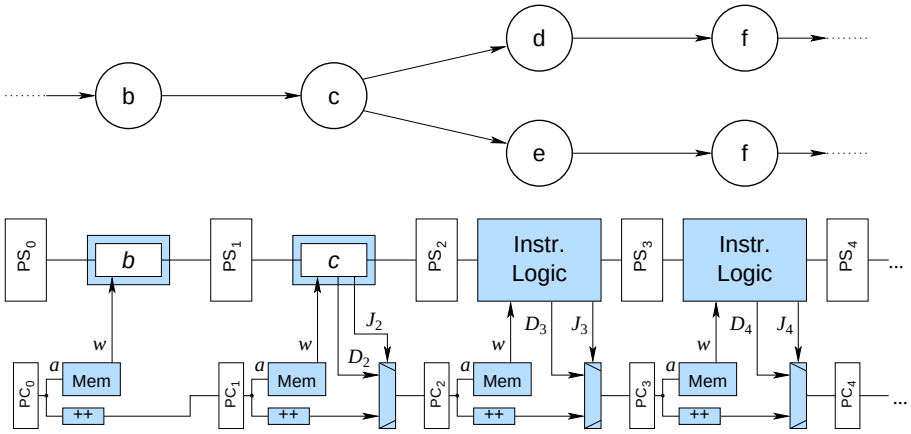

Abbildung 4.6: Ausrollen des abstrakten Models mit bedingtem Sprung

abstrakte Modell für eine bestimmte Anzahl an Taktschritten ausgerollt, beginnend von einer gegebenen Instruktion des Maschinenprogramms. Dabei entspricht ein Zeitschritt einer Instanz der Transitionslogik der angenommenen FSM. Abbildung 4.5 zeigt das ausgerollte abstrakte Modell für vier Zeitschritte. Bei dieser Art des Ausrollens ist der Programmzähler (PC) bis zur ersten Sprunginstruktion konstant. Wenn der PC konstant ist, dann ist auch das verwendete Instruktionswort konstant. Somit kann durch Konstantenpropagation die Instruktionslogik soweit vereinfacht werden, dass nur die entsprechende Instruktionszelle zum zugehörigen Instruktionswort als Logik verbleibt.

Im ausgerollten Modell werden die Signale jump (kurz: J_x), die als Steuersignale der Multiplexer (MUX) aus Abb.4.3 dienen, mit dem Wert 0 belegt, wenn die zum Signal gehörige Instruktion keine Modifikation am Programmfluss vornehmen kann. Bei einem unbedingten Sprung wird eine logische '1' angelegt und im Falle eines bedingten Sprungs wird eine einfache Logikfunktion auf der Basis des Architekturzustandes der zugehörigen Instruktion als Treiber für das Signal verwendet.

Sobald der erste bedingte Sprung in einem Zeitschritt des ausgerollten Modells vorkommt und den nächsten PC-Wert bestimmt, ist der PC nicht länger eine konstante Funktion. Entsprechend kann in den folgenden Zeitschritten keine Konstantenpropagation mehr erfolgreich durchgeführt werden. Das hat zur Folge, dass in diesen Zeitschritten im schlechtesten Fall die komplette Instruktionslogik des abstrakten Prozessormodells aus Abbildung 4.2 instanziiert werden muss. Abbildung 4.6 zeigt ein Beispiel für ein Ausrollen der programmierbaren Hardware der Instruktionen b bis f aus dem CFG aus Abbildung 4.1. Die erste Instruktion (b) ist eine nicht verzweigende Instruktion. Die darauffolgende Instruktion (c) ist ein bedingter Sprung. Offensichtlich sind die Werte des Programmzählers PC_2 und der Programmzustand PS_2 nicht mehr eindeutig, und somit ist auch keine Logikvereinfachung mehr für die dritte und die darauffolgende Instruktion in den nachfolgenden Zeitschritten möglich. Der dritte Zeitschritt zum Beispiel beinhaltet zwei verschiedene Instruktionen aus dem Programm: d und e. Mit jedem weiteren bedingten Sprung, der im Programmablauf vorkommt, nimmt die Anzahl der möglichen Instruktionen in einem Zeitschritt zu.

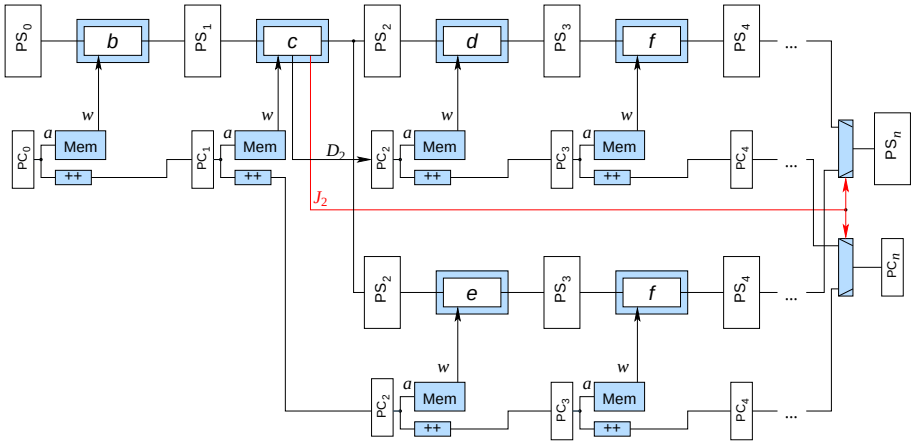

Abbildung 4.7: Duplizierte Logik nach Verschieben der Multiplexer

Dabei gehört der jeweilige Multiplexer immer zur einer entsprechenden Instruktion. Sowohl bei einer nicht verzweigenden Instruktion als auch bei direkten Sprüngen kann der Multiplexer mittels Konstantenpropagierung beseitigt werden, da das Steuersignal des Multiplexers einen konstanten Wert besitzt. Das funktioniert nicht mehr, sobald die Instruktion nach einem bedingten Sprung liegt, da dann zwei oder mehr Pfade modelliert werden. Dadurch können in dem Beispiel nur die Knoten (b) und (c) vereinfacht werden.

Die verbleibenden Multiplexer, die durch die Konstantenpropagierung nicht vereinfacht werden können, gehören zu bedingten Sprunginstruktionen. Ein solcher Multiplexer kann über die an seinen Ausgängen angeschlossenen Instruktionsblöcke hinweg verschoben werden, indem an jeden der Eingänge des Multiplexers statt des Multiplexers selbst eine Kopie der an den Ausgang des Multiplexer angeschlossenen Instruktionsblöcke angefügt werden.

Das dupliziert die Logik, aber dadurch ist in jeder der Kopien der Wert des PCs wieder eindeutig und somit konstant. Darauf kann die Konstantenpropagierung angewandt werden, die die Instruktionslogik wieder erheblich vereinfacht.

Abbildung 4.7 zeigt das Ergebnis, nachdem der Multiplexer vom ersten bedingten Sprung an das Ende des ausgerollten Modells verschoben wurde. Der obere Zweig mit den Instruktionen d und f zeigt den Fall, wenn das Steuersignal des Multiplexers J_2 auf den logischen Wert 1 gesetzt wurde und somit die Sprungadresse D_2 im Zeitschritt $t = 2$ verwendet wurde. Der untere Zweig steht für den Fall, dass der bedingte Sprung nicht stattfindet und J_2 auf 0 gesetzt wurde. Somit wurde der PC in diesem Zweig einfach nur inkrementiert.

Die Verschiebung des Multiplexers der Sprunginstruktion C dupliziert die Logik hinter der Sprunginstruktion. Das resultierende ausgerollte Modell entspricht einer Baumstruktur, die alle Pfade des Programms aufspannt. Das Programm endet mit Erreichen der letzten Instruktion des ursprünglichen Graphen aus 4.6 , die als Blattknoten in der Baumstruktur auftreten. Das Modell ist größer als das ursprüngliche Modell aus Abb. 4.6, aber jetzt ist jede Instruktion wieder eindeutig, und das Gesamtmodell kann mittels Konstan-

tenpropagierung optimiert werden. In diesem Beispiel wird in den letzten zwei Instruktionen auf beiden Pfaden nur noch die Logik für die einzelne konstante Instruktion benötigt. Im Fall des Sprungs ist es die Logik für die Instruktionen d und f, im anderen Fall die Logik für e und f. In Abbildung 4.6 sind an der entsprechenden Stelle die komplette Logik aller Instruktionen aus dem Datenpfad des abstrakten kombinierten HW/SW-Modells aus Abbildung 4.4 repräsentiert. Der verschobene Multiplexer vereint alle Pfade wieder auf den ursprünglichen Endzustand des Programms.

Entsprechend kann das Verfahren, wie dieser resultierende Logikbaum erzeugt wird, auch allgemein auf alle beliebigen Programme angewendet werden. Ein solcher Logikbaum ist eine rein kombinatorische Schaltung, der alle möglichen Ausführungspfade des entsprechend ausgerollten Maschinenprogramm darstellt.

Jeder Zeitschritt in der abgerollten Struktur repräsentiert die Ausführung einer Instruktion von einem Programmzustand zum nächsten. Die ausgeführte Instruktion ist eine Maschineninstruktion $C = (a, w)$ mit w als Instruktionswort und a der Speicheradresse im Speicher.

Die Größe des Ausführungsgraphen wächst im schlechtesten Fall (engl. *worst-case*) exponentiell mit der Anzahl der bedingten Sprünge im Programm. Deswegen ist es interessant Wege zu finden, um Pfade im Baum wieder zusammenzufassen und einen kompakteren aber weiterhin gerichteten und azyklischen Graphen zu erhalten. Wie das geschieht, wird später in Kapitel 4.1.1 diskutiert. Änderungen dieser Art werden im Allgemeinen auch mit dem englischen Begriff *Merging* bezeichnet und helfen, den Graphen wesentlich kompakter zu gestalten. Die Gesamtlogik kann somit kompakter sein als die Logik des Ausführungsbaums, der beim Verschieben der Multiplexer erzeugt wird.

Die Implementierung des ausgerollten Programms beschreibt den Programmkontrollfluss als hardwareabhängiges Modell. Teilstücke der Ausführungspfade zwischen Verzweigungs- und Zusammenführungspunkten im Graph sind explizit beschrieben und nicht implizit wie bei Ansätzen, die Prozessor+Programm ähnlich wie bei BMC abrollen. Wie in Kapitel 2.6.1 beschrieben, werden in solchen Ansätzen alle Ausführungspfade und die zugehörigen Übergangsrelationen in einer einzigen Kette an Zeitschritten repräsentiert. Die explizite Darstellung der Pfade in der abgerollten Struktur benötigt mehr Instanzen von Logikblöcken, doch dafür ist, wie vorher beschrieben, die Komplexität der einzelnen Logikblöcke wesentlich kompakter, und die Berechnung welcher Pfad betrachtet wird vereinfacht sich. Die sich ergebende Struktur entspricht der Programmnetzliste aus Kapitel 3 und ist somit das vorgeschlagene Berechnungsmodell für die formale Verifikation von hardwarenaher Software. Sie enthält alle möglichen Ausführungspfade des zu betrachtenden Programms.

Die Programmnetzliste setzt sich final aus drei Bestandteilen zusammen: Den Instruktionszellen für bedingte Sprünge / Verzweigungen, den Instruktionszellen für nicht verzweigende Instruktionen und den sogenannten Merge-Zellen aus dem nachfolgenden Kapitel 4.1.1.

Abbildung 3.3 zeigt eine Instruktionszelle für mögliche Verzweigungen, wie zum Beispiel einen bedingten Sprung. Der Logikblock erzeugt das Sprungsignal J. Statt das J-Signal bis ans Ende des ausgerollten Modells weiterzuleiten, wie es in Abbildung 4.7

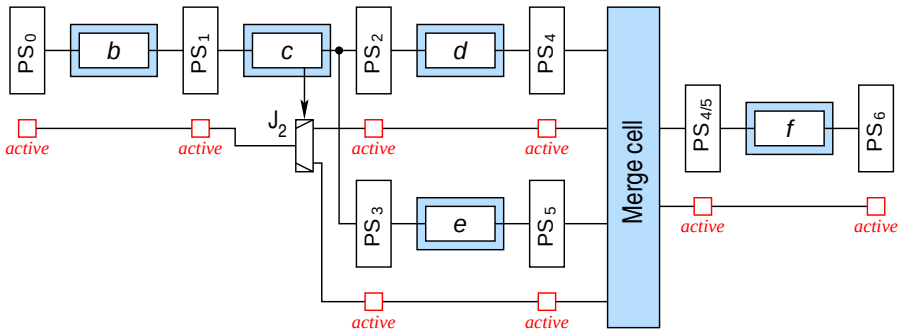

Abbildung 4.8: Programmnetzliste nach dem Zusammenführen.

zu sehen ist, wird ein $active$-Signal entlang jedes Strangs von den einzelnen Instruktionen in den Zeitschritten getrieben. Abbildung 4.8 zeigt die Implementierung des $active$-Signals für das Beispiel aus Abbildung 4.7. Dieses Signal markiert den aktuell betrachteten Pfad von allen in der Programmnetzliste vorhandenen Pfaden. In Abbildung 4.8 wird das $active$-Signal vom Demultiplexer, gesteuert durch J, an die Nachfolger propagiert. Je nachdem, wie J gesetzt ist, wird das ankommende $active$-Signal nur an den Zweig propagiert, der ausgewählt wurde, und die verbleibenden Zweige als nicht aktiv gesetzt. Für den Fall, dass der Sprung genommen wird (taken), wird das $active$-Signal an den Taken-Fall propagiert und der Fall *Not-Taken* auf nicht aktiv gesetzt, und umgekehrt. Es wird jedoch nie ein neues $active$-Signal erzeugt.

Das vereinfacht das Generieren der Programmnetzliste mit Hilfe von modularen Komponenten, und es erlaubt auch zusätzlich die einfache Erkennung von toten Pfaden.

4.1.1 Kompaktieren durch Zusammenführen

In diesem Unterkapitel wird das *Merging*, d.h. das Zusammenführen isomorpher Untergraphen des Execution-Graph, ausführlicher beschrieben und aufgezeigt unter welchen Bedingungen es möglich ist.

Der Control Flow Graph aus Abbildung 4.1 soll noch einmal in eine Programmnetzliste ausgerollt werden. Dabei gilt die zusätzliche Annahme, dass die Schleife zwischen b und g eine vordefinierte Anzahl an Iterationen besitzt. Ebenfalls wird angenommen, dass die Verzweigungsinstruktion im Knoten c von einem Wert abhängt, der von einer externen Eingabe gelesen wird. Wenn dieses Modell ausgerollt wird, hängt die Größe der resultierenden Programmnetzliste von der Anzahl an Schleifeniterationen ab.

Wird der Multiplexer von Knoten c nicht an das Ende des ausgerollten Modells geschoben (s. Abb. 4.7), sondern nur über den Knoten d und e zum Programmzustand vor den beiden Knoten mit der Instruktion f, dann können die beiden Knoten wieder zu einem Knoten zusammengefasst werden. Die Größe der resultierenden Programmnetzliste ist kleiner als die vergleichbare nicht kompaktierte Netzliste. Bei der finalen Netzliste wird pro Schleifeniteration mindestens ein Knoten eingespart. Abbildung 4.8 zeigt die so entstandene Programmnetzliste für den ausgerollten Teil des CFGs von Knoten b bis f.

Wenn die Multiplexer vorwärts – von ihrer Entstehungsposition in Richtung der nachfolgenden Instruktionen – durch das ausgerollte Modell bewegt werden, kann das Verschieben schon dann gestoppt werden, wenn eine Stelle im Programm erreicht wird, die auf beiden Ausführungspfaden (*Taken*- und *Not-Taken*-Zweig) vorkommt.

Dadurch kann die Anzahl der Zeitschritte/Basisblöcke, die durch die Duplizierung der Logik sonst benötigt würden, signifikant verringert werden. Diese Optimierung wird in dem folgenden kombinierten Ausroll- und Merge-Algorithmus für die Generierung der Programmnetzliste verwendet:

- Rolle das Maschinenprogramm von Instruktion zu Instruktion einzeln aus
- Generiere bei einem bedingten Sprung den zusätzlichen Zweig und rolle alle Zweige aus.
- Wenn beim Ausrollen eine Instruktionsadresse (PC-Wert) vorkommt, der schon mal besucht wurde, dann ist die Instruktionszelle ein Merge-Kandidat, sofern folgende weitere Bedingungen erfüllt sind:
  - Wenn beim Zusammenführen kein Zyklus in der PN generiert wird, dann füge eine Merge-Zelle vor der schon existierenden Instruktionszelle ein und vereinige die Zweige.
  - Wenn beim Zusammenführen ein Zyklus entstehen würde, dann wird nicht zusammengeführt, sondern eine neue Instanz der entsprechenden Instruktionszelle generiert.

Wie zu erkennen ist, werden in diesem Algorithmus keine Multiplexer relokalisiert. Die Funktionalität der verschobenen Multiplexer wird hier durch Merge-Zellen implementiert. Diese Zellen leiten diejenigen Werte des Programmzustandes weiter, der von den Programmzuständen am Eingang der Merge-Zellen mit Hilfe des $active$-Signals als aktiv markiert ist. Diese Konstruktion ist äquivalent zu der ursprünglichen Implementierung mit den verschobenen Multiplexern, hat aber den Vorteil einer deutlich vereinfachten Programmnetzlistenkonstruktion.

4.2 Modellgenerierung

In diesem Kapitel wird der Ansatz aus Kapitel 3 zunächst kurz wiederholt und im Anschluss werden zum einen Details besprochen, die für eine korrekte Generierung notwendig sind und zum anderen welche unterschiedlichen Strategien dabei verwendet werden können. Als Ausgangspunkt für die Generierung der Programmnetzliste wird die Software in Form von Maschinencode verwendet.

Der Code wird mithilfe eines Parsers ausgelesen und es wird ein Control Flow Graph (CFG) erstellt. Aus dem CFG wird mit Hilfe eines kombinierten Ausroll- und Merge-Algorithmus ein Execution-Graph (EXG) generiert. Dabei werden zur Optimierung "*tote*

Pfade" erkannt und von einer Darstellung im EXG ausgeschlossen. Tote Pfade sind jene Pfade, die in keiner möglichen Ausführung des Programms jemals aktiv werden können. Ist der finale EXG erzeugt, wird jeder Knoten im Graphen durch eine entsprechende Instruktionszelle ersetzt und so die Programmnetzliste generiert. Dabei muss für jede unterstützte ISA ein eigener Satz von Instruktionszellen generiert werden.

4.2.1 Extraktion des Kontrollflussgraphen

Soll das Verhalten der Software auf dem Prozessor in einer gegebenen Umgebung untersucht werden, ist es nicht ausreichend, die Software auf einer hohen Programmiersprache wie zum Beispiel C zu verifizieren. Auch Assemblercode kann noch Pseudoinstruktionen beinhalten, die erst in die von der Instruction Set Architecture unterstützten Maschineninstruktionen umgesetzt werden müssen. Zusätzlich müssen Labels in korrekte Adressen übersetzt werden. Maschinencode ist der Code, der tatsächlich auf dem Prozessor ausgeführt wird und somit der einzige, der das Verhalten des Prozessors exakt und vollständig definiert. Deswegen wird in dieser Arbeit der Machinencode betrachtet. Der Maschinencode ist dabei eine binäre Datei, die für die jeweilige unterstützte Plattform generiert wird. Zum Erstellen des CFGs müssen benötigte Informationen mit Hilfe eines Parsers extrahiert werden, der an die jeweilige Architektur angepasst ist.

Eine binäre Maschinencodedatei setzt sich im Wesentlichen aus einem Textsegment und einem Datensegment zusammen. Das Textsegment enthält die Instruktionen und das Datensegment die mitgegebenen Daten. Im Textsegment können neben Instruktionen auch zusätzlich Daten vorkommen. Die Besonderheit dabei ist, dass ein Datum, wenn es zufälligerweise einen Wert besitzt, der gültigem Maschinencode entspricht und als Instruktion erkannt werden kann, zusätzliche Knoten im Graphen erzeugen kann. Der Aufwand, die einzelnen Informationen zu extrahieren, und Daten von Instruktionen auseinander zu halten, wird in Kapitel 6 ausführlicher beschrieben.

Wie gerade beschrieben, kann ein erzeugter Kontrollflussgraph auch generierte Knoten besitzen, die real gar nicht zum Programmfluss dazugehören. In der Regel besitzen diese Knoten keine eingehenden Kanten und können bei der Erzeugung des Execution-Graphen nicht von den gegebenen Startknoten aus erreicht werden. Deswegen werden sie auch im späteren EXG nicht mehr auftreten. So haben die Daten, als Instruktion interpretiert, keinen Einfluss auf die Gesamtgröße oder das Verhalten des finalen Modells, sondern können ausschließlich in ihrer ursprünglichen Bedeutung als Datum das Modell beeinflussen.

Nach der Extraktion und Erzeugung der Knoten für die Instruktionen werden die Kanten des Graphen erzeugt, wobei von der ersten Instruktion bzw. von mehreren gegebenen Startinstruktionen aus gestartet wird. Die erste Instruktion des Programms ist dabei in der binären Datei immer eindeutig gegeben.

Anschließend werden alle Instruktionen der Reihe nach überprüft, ob sie eine verzweigende oder eine nicht verzweigende Instruktion sind. Im letzten Fall ist der Nachfolger der Instruktion immer die direkt nachfolgende Instruktion, da die Instruktionen zusammen ein sequentielles Verhalten abbilden.

Im Falle eines Sprunges wird versucht, den Nachfolger zu ermitteln. Ist der nicht direkt in der Instruktion kodiert, so kann zum Beispiel mittels einer statischen Analyse, wie der Konstantenpropagierung, versucht werden das Sprungziel zu ermitteln. Ist das erfolgreich, kann die entsprechende Kante hinzugefügt werden. Gelingt die Ermittlung nicht, entsteht ein unvollständiger CFG. Beim Fehlen von Kanten im CFG können diese später bei der Generierung des Execution-Graphen berechnet werden, wie in Kapitel 4.2.5 beschrieben.

Der CFG kann auch zusätzliche unvollständige Daten beinhalten wie zum Beispiel Adressen für Lese- und Schreiboperationen. Eine Lösung dieses Problems wird in Kapitel 4.2.6 und [BVS+14b] vorgestellt.

4.2.2 Ausrollen des Kontrollflussgraphen

Nachdem der Kontrollflussgraph erstellt wurde, wird aus diesem der Execution-Graph generiert. Dabei wird – beginnend vom Startknoten des CFGs – der Graph Knoten für Knoten durchlaufen. Listing 4.1 beschreibt den Ablauf zur Generierung des Execution-Graphen, in dem der CFG knotenweise abgerollt wird. Der Startknoten ist eine Instruktion, die definitiv im Ausführungsgraphen vorkommt, da dieser Knoten auf jedem möglichen Ausführungspfad durch das Programm liegt. Dieser Knoten ist somit zwingend lebendig. Da dieser Knoten zu Beginn der einzige zu bearbeitende Knoten ist, wird direkt im Anschluss auch sein Nachfolger überprüft. Gibt es nur einen eindeutigen Nachfolger, kann dieser direkt angelegt werden; wenn der aktuelle Knoten auf einem möglichen und somit gültigen Pfad liegt, liegt der einzige Nachfolger auch auf demselben gültigen Pfad. Sollte es mehr als einen Nachfolger geben, wird in den If-Zweig des Algorithmus aus dem Listing 4.1 gesprungen. In diesem Fall werden alle Nachfolger überprüft, ob sie lebendig sind. Lebendig bedeutet, dass sie mindestens auf einem gültigen Ausführungspfad des Programms liegen. Die lebendigen Knoten werden der zu bearbeitenden Liste hinzugefügt. Die als nicht lebendig verbleibenden Knoten liegen demnach auf keinem gültigen Pfad und müssen nicht weiter auf mögliche Nachfolgerknoten untersucht werden. Der Algorithmus wird solange durchgeführt, bis keine unbearbeiteten Knoten mehr verbleiben, was bedeutet, dass das Programm fertig abgearbeitet wurde.

Das Verfahren generiert keine eindeutigen Graphen. Je nach Abarbeitungsstrategie, also z.B. in welcher Reihenfolge die Knoten abgearbeitet werden, können von ein und demselben Programm unterschiedliche Graphen generiert werden. Die Graphen können sich sowohl durch Anzahl der Knoten und ebenfalls durch den Aufbau unterscheiden. Dennoch bilden die unterschiedlichen Graphen die gleichen Pfade ab. Eine Untersuchung über unterschiedliche Strategien wurde in [Kom11] durchgeführt.

Listing 4.1: Algorithmus zur Generierung des Execution Graph

```
1 Generiere_EXG(CFG, StartKnoten)
  {
    Liste von unprozessierte Knoten (U_Knoten) = leer;
    EXG = leer;
5   U_Knoten = Liste von Startknoten;
```

```
    do
      {
       Nehme Knoten K aus U_Knoten;
       Fuege K in EXG ein;
10     if (K.Nachfolger >1 )
         { // Mehrere Nachfolger
           Teste K.Nachfolger(i) ob lebendig
           if (i lebendig)
             { // Nur lebendige Nachfolger einfuegen
15             U_Knoten = U_Knoten + i;
             }
         }
       else
         {  // Nur ein Nachfolger
20         U_Knoten = U_Knoten + K.Nachfolger
         }
      }
    while (U_Knoten != leer)
    return EXG; // Finaler EXG
25 }
```

4.2.3 Kompaktieren durch Vermeidung toter Verzweigungen

Ein Kontrollflussgraph mit einem Zyklus beinhaltet unter Umständen Ausführungspfade mit infiniter Länge. Folglich muss die Programmnetzliste, die diesem CFG entspricht, ebenfalls infinit lang sein. Die in dieser Arbeit betrachteten Beispiele in den Experimenten sind Low-Level-Software, und sie bestehen aus Programmen und Treibern, die in Echtzeitsystemen und Interrupt-getriebenen Systemen Verwendung finden. Sowohl Interrupts als auch Programme benötigen eine finite Ausführungszeit, da sonst die Echtzeitanforderungen nicht erfüllt werden können.

Generell muss das vorgestellte Verfahren sowohl mit finiten als auch mit infiniten Zyklen umgehen können. Im Folgenden wird von Applikationen mit finiten Zyklen ausgegangen. Der Umgang mit infiniten Zyklen wird in Teilkapitel 4.2.6 gesondert erläutert.

Ein Beispiel für einen Zyklus ist eine FOR-Schleife mit einer endlichen Anzahl an Durchläufen. Wenn die Schleife aus dem Control Flow Graph in Abbildung 4.1 als FOR-Schleife der Länge N implementiert ist, wird die Verzweigung *Taken* beim Knoten g zurück zum Anfang der Schleife $N - 1$ mal durchlaufen. Beim N-ten Durchlauf wird dann die Kante *Not-Taken* anstelle des Rücksprungs zum Knoten b genommen. Da bei einer FOR-Schleife, bedingt durch die Zählvariable, immer nur eine der beiden Verzweigungen aktiv sein kann, ist es möglich den anderen Pfad wegzuoptimieren. Um solche inaktiven Verzweigungen zu identifizieren wird – während der Generierung der Programmnetzliste – ein SAT-basierter Eigenschaftsprüfer eingesetzt. Die notwendigen Eigenschaften sind einfach zu prüfen und können voll automatisiert generiert werden. In der zu prüfenden Eigenschaft wird nur überprüft, ob die jeweilige Verzweigung an dem bedingten Sprung auf einem gültigen Pfad des Programms liegt. Dafür ist es ausreichend das Active-Signal

zu betrachten, das mit dem entsprechenden Programmzustand der aktuell betrachteten Verzweigung assoziiert ist. Kann das Active-Signal aktiv sein, dann liegt der betrachtete Programmzustand auf mindestens einem erreichbaren Pfad des Programms. So eine Verzweigung wird lebendig genannt.

Im Beispiel der FOR-Schleife wird die Verzweigung *Taken* $N - 1$ mal als lebendig geprüft. Beim N-ten Durchlauf wird der *Taken* Pfad als inaktiv erkannt. Entsprechend werden die ausgerollten Verzweigungen *Not-Taken* $N - 1$ mal als inaktiv erkannt und nur beim letzten Durchlauf wird die Verzweigung lebendig. Diese Information hilft das ausgerollte Modell zu vereinfachen.

Die geprüften Pfade, die inaktiv sind, werden als sogenannte tote, nicht erreichbare, Zweige angesehen, da sie in keinem einzigen möglichen Ausführungspfad des Programms liegen. Entsprechend werden die Verzweigungen in dem Modell auch als tot markiert und entsprechend nicht weiter ausgerollt. Das geschieht in dem verwendeten Algorithmus in Listing 4.1 automatisch, da nur lebendige Knoten in die Liste zur weiteren Bearbeitung aufgenommen werden. Was hier exemplarisch für die FOR-Schleife vorgestellt wurde, funktioniert ebenfalls für alle anderen Schleifen, nur dass eventuell dort die Schleifenbedingungen nicht immer so offensichtlich erkennbar sein müssen. Der Beweis wird nur für die Verzweigung durchgeführt und gilt somit ganz allgemein für beliebige Verzweigungen, ob sie nun zur Realisierung einer Schleife verwendet werden oder nicht.

4.2.4 Zusammenführen zweier Knoten

Pfade können – wie in Kapitel 4.1.1 beschrieben – dadurch vereint werden, dass sie über Knoten verfügen, die die Instruktion an derselben Programmstelle abbilden. Als zusätzliche Bedingungen muss gelten, dass durch das Vereinigen der Knoten keine Schleifen erzeugt werden.

Der Algorithmus aus Listing 4.1 lässt sich mit wenig Aufwand um die Fähigkeit des Zusammenführens erweitern, wie es in Listing 4.2 dargestellt ist. Er wurde um eine Funktion erweitert, die testet, ob ein Merge-Partner im bisher erzeugten Execution-Graph vorhanden ist. Zusätzlich kommt noch ein If-Then-Else-Konstrukt hinzu, das abhängig davon, ob ein Merge-Partner gefunden wurde, entweder einen neuen Knoten generiert oder den neuen Knoten mit dem vorhandenen vereint und die notwendige neue Kante erzeugt.

Listing 4.2: Erweiteter Algorithmus mit Merge Funktionalität

```
1  Generiere_EXG(CFG, StartKnoten)
   {
     Liste von unprozessierte Knoten (U_Knoten) = leer;
     EXG = leer;
5    U_Knoten = Liste von Startknoten;
     do
       {
        Nehme Knoten K aus U_Knoten;
        Test_Mergen(K, EXG);
10      if (Test_Mergen(K, EXG))
          {
```

```
            Merge(K, EXG);
          }
        else
15        {
          Fuege K in EXG ein;
          if (K.Nachfolger >1 )
            { // Mehrere Nachfolger
              Teste K.Nachfolger(i) ob lebendig
20            if (i lebendig)
                { // Nur lebendige Nachfolger einfuegen
                  U_Knoten = U_Knoten + i;
                }
            }
25        else
            {  // Nur ein Nachfolger
              U_Knoten = U_Knoten + K.Nachfolger
            }
         }
30     }
    while (U_Knoten != leer)
    return EXG; // Finaler EXG
  }
```

Im einfachsten Fall gibt es nur einen Knoten, der mit dem aktuell betrachteten Knoten zusammengeführt werden kann. In diesem Fall ist keine Strategie für das Zusammenführen notwendig.

Es ist auch möglich, dass es mehrere Knoten im Graphen mit identischer Instruktion gibt. Eine solche Situation ergibt sich, wenn sie auf einem gemeinsamen Pfad liegen und das Zusammenführen zwei dieser Knoten eine verbotene Schleife im Execution-Graph erzeugt hätte. Werden mehrere Knoten gefunden, die als Kandidaten zum Zusammenführen zur Verfügung stehen, so ist eine Wahl zu treffen. Zwei Strategien haben sich als vorteilhaft erwiesen:

- mit dem ersten Knoten in der Sequenz zusammenführen
- mit dem letzten Knoten in der Sequenz zusammenführen

Eine Untersuchung, ob und welche Strategie effizienter ist, wurde in [Kom11] durchgeführt. Die Strategien können unterschiedlich optimierte Graphen erzeugen, die dennoch die gleichen Ausführungspfade des Programms beinhalten.

Durch das Vereinigen von einem Knoten mit einem existierenden Ziel-Knoten werden auch die zugehörigen unterschiedlichen Pfade vereint. Deswegen müssen alle Knoten im Graph, die Nachfolger des Ziel-Knotens sind, nochmals untersucht werden, da sie durch das Zusammenführen nun zu weiteren Pfaden gehören können. Dabei kann es vorkommen, dass bis dahin als tot erkannte Verzweigungen wieder als lebendig geprüft werden, da die neuen Pfade auch diese Verzweigung beinhalten. Zusätzlich können Sprünge eventuell neue Sprungziele bekommen, sofern diese nicht konstant sind. Aber auch Lese- und

Schreib-Operationen, die mehr als auf konstante Adresse zugreifen, könnten unter Umständen wieder auf neue Adressen lesen bzw. schreiben.

Je weniger Nachfolgeknoten beim Zusammenführen neu betrachtet werden müssen, desto weniger Änderungen ergeben sich dann bei der Merge-Operation.

4.2.5 Unbekannte Verzweigungsziele

In den bisherigen Kapiteln wurde davon ausgegangen, dass der Control Flow Graph vollständig aus dem Maschinencode extrahiert wird. Es ist jedoch möglich, dass die Adresse für das Sprungziel bei einer Sprunginstruktion aus komplexeren Operationen errechnet wird. Als Folge kann beim Erstellen des CFGs keine Ausgangskante für den Sprungknoten erstellt werden und einige Sprungziele des CFGs sind zunächst unbekannt.

Im Falle eines unvollständigen Control Flow Graph kann die bis dahin generierte Programmnetzliste für die Berechnung des unbekannten Sprungs verwendet werden, da sie die komplette "Vorgeschichte" der Sprunginstruktion beinhaltet. Die Berechnung dieser Sprungziele kann mittels Pfad-Simulation oder einem iterativen SAT-basierten Ansatz berechnet werden. Eine ausführliche Beschreibung der Methodik ist in [BVS+14b] und [Vil16] zu finden.

Der Vorteil dieses Verfahren ist, dass nicht alle Sprungziele für alle Pfade berechnet werden müssen, sondern nur die Sprungziele für die Pfade, die in der Programmnetzliste an der Stelle aktiv sind, an der die betrachtete Instanz der Sprunginstruktion instanziiert wurde.

Wird zum Beispiel im C-Code eine Switch-Instruktion verwendet, so kann diese in Assembler auf eine Sprunginstruktion mit einer zugehörigen Sprungtabelle abgebildet werden. So müssten im zugehörigen CFG alle möglichen Sprungziele aus der Sprungtabelle als Kanten im Graphen existieren. Sind die Sprungziele jedoch von der Historie des bisher ausgeführten Programms abhängig, ist es möglich, dass nur ein Fall oder ein Teil der Fälle an der betrachteten Stelle in der Programmnetzliste aktiv sind. So müssen an den entsprechenden Stellen in der Programmnetzliste nicht alle Sprungziele aus der Tabelle als Nachfolgekante zum Sprungknoten erstellt werden. Es ist daher möglich die Anzahl an zu berechnenden Kanten für die CASE-Instruktion in der Programmnetzliste im Vergleich zum unausgerollten CFG zu reduzieren, was ebenfalls zu einem reduzierten Aufwand bei der Berechnung der Sprungziele führt. In einer Programmnetzliste ist es jedoch möglich, dass eine Instruktion aus dem CFG mehrfach instanziert wird. Ist dies auch für die CASE-Anweisung der Fall, so muss die Berechnung für ihre Sprungziele mehrfach – also bei jeder Instanzierung – durchgeführt werden. Der Optimierungsvorteil bei der Sprungreduktion wird durch die Mehrfachinstanzierung nicht unbedingt wieder ausgeglichen. Bei der Berechnung der Sprungziele entfällt die sonst separat notwendige Prüfung auf Lebendigkeit aller Verzweigungen, da diese in dem Verfahren aus [BVS+14b] mit abgedeckt ist.

4.2.6 Infinite Schleifen

Sollte sich die Entscheidungsbedingung für einen bedingten Sprung auf Daten beziehen, die von außerhalb des Programms geladen werden, ist die Bedingung bei der Modellgenerierung eingabeabhängig und damit nicht eindeutig zu entscheiden. Dies kann dazu führen, dass bei einer Verzweigung immer beide Fälle (*Taken* und *Not-Taken*) lebendig sein können. Gehört eine solche Verzweigung zu einem Schleifenkörper und entscheidet darüber, ob die Schleife fortgeführt oder abgebrochen wird, führt dies zur einer Endlosschleife mit infiniter Länge. Solche infiniten Schleifen führen bei der Erstellung der Programmnetzliste zu einem endlosen Abrollen des CFG.

Um infinite Schleifen zu vermeiden, werden – ähnlich wie bei gebräuchlichen Ansätzen der Softwareverifikation oder der Worst-Case-Execution Time (WCET) Analyse [BBK09] – zusätzliche vom Anwender definierten Einschränkungen hinzugefügt, die die Anzahl der Schleifendurchläufe begrenzen können. Einschränkungen dieser Art sind generell möglich, da der Entwickler das Verhalten der Umgebung kennt und daraus Constraints erstellen kann, die eine präzisere Beschreibung der möglichen Eingabewerte wiedergibt. Aus der Programmnetzliste oder dem Quellcode selbst können Constraints für die Werte von externen Variablenzugriffe nicht erkannt werden.

Wird ein solcher Constraint bei der Erstellung der Programmnetzliste eingesetzt und auch erreicht, gilt jeder Beweis auf der Programmnetzliste nur unter der Annahme, dass der Constraint ebenfalls gültig ist.

4.3 Instruktionszellen

Zur Generierung der finalen Programmnetzliste werden die Knoten des Execution-Graphen durch die entsprechenden Instruktionszellen ersetzt. Der Satz an Instruktionszellen muss für jede zu unterstützende Architektur (Instruction Set Architecture) einmal generiert werden.

Je nach den Anforderungen, die sich aus den Verifikationszielen ergeben, können unterschiedliche Detailgrade oder Fokussierungen auf unterschiedliche Aspekte in der finalen Programmnetzliste berücksichtigt werden. Die unterschiedlichen Aspekte und Detailgrade lassen sich in der Programmnetzliste durch verschiedene Implementierungen der Instruktionszellen realisieren.

Die Komplexität der Instruktionszellen hängt ab vom gewünschten – zu modellierenden – Detailgrad und von der Komplexität der betrachteten Architektur.

Die Instruktionszellen können zum Beispiel auf der Ebene der Instruction Set Architecture betrachtet werden, wie sie in Prozessorspezifikationen gegeben ist. Dieses ISA-basierte Modell ist nicht taktzyklengenau, da das Zeitverhalten von der konkreten Implementierung abhängen kann und nicht für alle Implementierungen gleich sein muss. Die ISA gibt nur das Verhalten der Instruktion vor, ohne das genaue Zeitverhalten oder jeden Takt genau zu spezifizieren.

Alternativ ist es möglich die Zellen mit taktzyklengenauem Verhalten zu modellieren, das dem der Implementierung eines Prozessors auf RT-Ebene entspricht. Der höhere De-

tailgrad führt jedoch zu höherer Komplexität der Instruktionszellen und damit auch der der Programmnetzliste.

Es sind beliebige andere Abstraktionen oder Präzisierungen möglich. Bei Bedarf können auch mehrere unterschiedliche Detailgrade für eine und dieselbe unterstützte Architektur als Zellen implementiert werden. Diese werden dann parallel vorgehalten. Je nach Verifikationsaufgabe kann der geeignete Satz an Instruktionszellen zur Generierung der finalen Programmnetzliste verwendet werden.

Eine beispielhafte Beschreibung für die Implementierung der Instruktionszellen wird in Appendix A gezeigt.

4.3.1 Generierung der Instruktionszellen

Im letzten Teilkapitel wurde gezeigt, dass für jeden Prozessor ein vollständiger Satz von Instruktionszellen generiert werden muss. Ein vollständiger Satz beinhaltet für alle Instruktionen des Prozessors eine entsprechende implementierte Zelle. Alternativ kann ein Set von Instruktionszellen für mehrere Prozessoren erzeugt werden, wenn die Zellen auf dem abstrakteren ISA-Level arbeiten sollen. Da die Programmnetzliste als fehlerfreies Modell generiert werden soll, um mit ihr eine verlässliche Verifikation durchführen zu können, müssen die Instruktionszellen ebenfalls fehlerfrei sein.

Die Zellen in der Programmnetzliste implementieren das Verhalten des Programms auf der CPU und müssen für die Verifikation dieses Verhalten auch garantieren, damit dieses zuverlässig ist. Deswegen müssen die Zellen gegenüber den Prozessoren als fehlerfrei geprüft werden.

Alternativ können die Zellen auch anhand der ISA verifiziert werden, wenn der betrachtete Prozessor gegenüber der ISA verifiziert wurde.

Es gibt mehrere Möglichkeiten, einen fehlerfreien Satz von Instruktionszellen zu erzeugen, die in den folgenden Absätzen beschrieben werden.

Wenn der Prozessor gegenüber der ISA verifiziert wurde, können die dort erzeugten Eigenschaften ebenfalls für die Verifikation der Instruktionszellen genutzt werden. Dabei ist eine neue Zuordnung zwischen den im Design verwendeten Signalen und den Signalen in der Instruktionszelle zu erstellen. Eine passende Abstraktion des Eigenschaften-Sets kann mittels Makros erreicht werden, die ausschließlich für die Zuordnung der Signale in den zu verifizierenden Designs zu den Signalen der Eigenschaften verwendet werden. Das ist der Stand der Technik und kann bei den meisten Beweisverfahren ohne Probleme durchgeführt werden.

Ist der Prozessor nicht gegenüber der ISA verifiziert, so muss müssen die Instruktionszellen eigenständig verifiziert werden. Dabei gibt es – wie oben beschrieben – die Möglichkeit einen Eigenschaftssatz zu erstellen der auf beiden Designs – Prozessor und Instruktionszellen-Satz – geprüft werden kann. Ein alternativer Ansatz besteht darin, die Prozessorverifikation mittels speziell beschriebener Eigenschaften zu prüfen, die sich danach eins zu eins in die Instruktionszellen überführen lassen. Diese Methode wurde in [Udu11] erfolgreich durchgeführt.

4.3.2 Erweiterung der Instruktionszellen-Logik zur Modellierung des Kontrollflusses

Die Zellen aus einem Satz von Instruktionen beschreiben das Verhalten des Prozessors jeweils für eine bestimmte Instruktion. Das Verhalten wird dabei auf der Abstraktionsebene der ISA mit rein kombinatorischer Logik repräsentiert.

Für die Programmnetzliste unterstützen die Zellen aber noch ein paar Funktionen, die nicht zur eigentlichen Funktion des Prozessors gehören. Diese könnten komplett unabhängig von der Zelle generiert werden, dennoch werden sie immer mit einer spezifischen Zelle assoziiert. Deswegen wird es im Folgenden als Erweiterung der Zelle angesehen, ohne jedoch das ursprüngliche Verhalten zu modifizieren oder zu beeinflussen. Das wichtigste Element ist das in den Programmzustand hinzugefügte Active-Bit. Es dient dazu, den aktiven Pfad in der Programmnetzliste markieren zu können. Zu jedem Programmzustand zwischen zwei Instruktionszellen gibt es ein Active-Bit Signal, kurz Active-Signal. Ist der Wert gleich 1 bzw. $true$, dann liegt der Programmzustand auf dem aktiv betrachteten Pfad. Die darauf folgende Instruktion ist somit auch zwangsläufig aktiv. Die Aufgabe der Instruktionszellen ist es, den Wert des Active-Signal an den Folgezustand zu propagieren. In den Zellen, die einen bedingten Sprung modellieren, wird abhängig durch die zu prüfende Bedingung das Active-Signal nur an einen der möglichen Nachfolger propagiert. Die anderen Nachfolger werden auf 0 bzw. $false$ gesetzt.

Ein Feature ist, dass dieses Active-Signal für die automatisierte Generierung der Eigenschaften genutzt werden kann. Mit diesen Eigenschaften werden die Verzweigungen bei einem Branch auf Lebendigkeit geprüft. Das Signal hat nur ein Bit, sodass auf einen einfachen booleschen Ausdruck geprüft wird.

Das Active-Signal ermöglicht ebenfalls ein einfaches Zusammenführen von Pfaden mit Hilfe eines Multiplexers, wie es in Kapitel 4.1.1 vorgestellt wird.

Die Programmnetzliste modelliert die Ausführung des Programms auf dem Prozessor ohne weitere Maßnahmen nicht taktzyklengenau, sondern stellt lediglich die Abarbeitung von Maschineninstruktionen in abstrakten Schritten dar. Durch das Zusammenführen von Pfaden beim Merging ist es möglich, dass ein Programmzustand in der PN verschiedene Zeitpunkte repräsentieren kann, wenn mehrere Pfade durch diesen Zustand führen. Um genau zu bestimmen, in welchem Zeitschritt dieser Zustand sich gerade befindet, werden die Zeitschritte in den Zuständen auf dem aktuell gültigen Pfad selbstständig aufsummiert.

Der Ansatz sieht vor, dass jede Zelle über einen Zähler verfügt, der die abstrakte Zeit aufaddiert, die die Zelle braucht, und das Ergebnis mit dem Programmzustand weiterleitet. So können Eigenschaften, die auf dem Modell formuliert werden, zeitliche Vergleiche beschreiben und mittels eines SAT-Solvers verifiziert werden.

Wieviel abstrakte Zeit während der Ausführung einer Instruktion vergeht, kann für jeden Instruktionstyp individuell und frei festgelegt werden. Es können die Zeiten verwendet werden, die in der ISA spezifiziert sind, oder die konkreten Zeiten können auch aus einer gegebenen RTL-Implementierung extrahiert werden. Der einfachste Fall ist der, dass immer ganz abstrakt der Wert eins addiert wird und damit nur die Anzahl der ausgeführten Operationen aufaddiert wird.

4.4 Speichermodellierung

Der Programmzustand der Programmnetzliste setzt sich aus dem Architekturzustand des Prozessors und der im Speicher abgelegten Variablen zusammen. Für die Generierung der Programmnetzliste sind nur die Speicherstellen relevant, die vom Programm verwendet werden. Vor dem Generieren der Programmnetzliste ist nicht genau bekannt, welche Speicheradressen vom Programm belegt wären. Prinzipiell könnte man den gesamten verfügbaren Speicher modellieren. In diesem naiven Ansatz würden allerdings zu viele Signale generiert, die zwar durch intelligente SAT-Solver teilweise wieder wegoptimiert werden könnten, was aber Zeit erfordern würde. Daher lohnt eine Optimierung des Programmzustandes auf den tatsächlich verwendeten Speicher.

Bei den Speicherzugriffen müssen drei unterschiedliche Arten von Speicher betrachtet werden:

1. Speicher, der ausschließlich vom betrachteten Programm selbst verwendet wird,
2. Shared-Memory (gemeinsam mit anderen Softwarekomponenten genutzter Speicher),
3. Peripherieregister.

Speicher, der ausschließlich vom Programm selbst benutzt wird, wird beim Lesen einer Variablen, nachdem sie vorher vom Programm geschrieben wurde, immer den zuletzt geschrieben Wert zurückgeben. Dadurch kann ein vereinfachtes Speichermodell generiert werden, in dem Lese- und Schreiboperationen über ihre Ports entsprechend direkt miteinander verbunden werden.

Falls ein Knoten in der Programmnetzliste mit einer Leseinstruktion auf mehreren Pfaden liegt (nach Merging), können – abhängig vom aktiven Pfad – verschiedene Knoten in der Programmnetzliste den letzten Wert an die gelesene Speicherstelle geschrieben haben. Da die Programmnetzliste statisch ist, lassen sich die unterschiedlichen Zugriffe mit linearem Aufwand berechnen und entsprechend modellieren, wie es in Abbildung B.1 gezeigt wird. Der Aufbau der dafür zusätzlichen Logik wird ausführlicher im Anhang B beschrieben.

Wenn das Programm hingegen auch Shared-Memory benutzt, kann der Wert sich innerhalb einer Speicherstelle zu beliebigen Zeitpunkten ändern und so bei mehrfachem Lesen unterschiedliche Werte beinhalten. Die zugehörigen Variablen werden überlicherweise als *volatile* bezeichnet. Ob Shared-Memory verwendet wird, sollte dem Benutzer bekannt sein, sodass diese Speicherbereiche entsprechend markiert werden und nicht das vereinfachte Speichermodell generiert wird. Ist eine Speicherstelle als Shared-Memory markiert, wird sie als offener Input in der Programmnetzliste angesehen, an dem jeder beliebige Wert angelegt sein kann.

Sind alle Programme bekannt, die auf den Shared-Memory zugreifen können, kann auch eine kombinierte Netzliste generiert werden, wie sie z.B. in Kapitel 4.5 erläutert wird, um so zum Beispiel interruptgetriebene Systeme zu beschreiben. In diesem Fall ist es möglich wieder ein vereinfachtes Speichermodell zu generieren.

Ist der Shared-Memory-Bereich nicht bekannt, können mit den jeweiligen Programmnetzlisten der vorhandenen Treiber und Programme die gemeinsam genutzten Speicheradressen berechnet werden. Ein automatisiertes Verfahren wird in [Feh12] vorgestellt.

Memory-mapped IO erlaubt es, mit denselben Instruktionen auf die Peripherie zuzugreifen wie auf den Arbeitsspeicher. Dadurch kann sich hinter bestimmten Speicheradressen ein Register von Hardwarekomponenten verbergen. Diese Register können nicht nur durch das Programm geändert werden, sondern auch durch die Hardware selbst. Das bedeutet, dass sich der Wert zwischen zwei Lesezugriffen ändern kann und es nicht mehr möglich ist, den Wert der letzten Schreiboperation auszulesen. Somit wird für die Modellierung von speziellen Speicherstellen, die sich auf Hardwarekomponenten beziehen, ähnlich zum Shared-Memory ein offener Input als Eingang für das Modell angenommen.

4.5 Modellierung von Interrupts

Hardwarenahe Software kommuniziert mit den im eingebetteten System integrierten Hardwarekomponenten. Wie in Kapitel 2.4.3 beschrieben, findet ein Großteil der Kommunikation zwischen Hardware und Software aus Effizienzgründen oft interrupt-getrieben statt.

In diesem Abschnitt wird gezeigt, wie eine interrupt-getriebene Low-Level-Software-Architektur mittels der bereits vorgestellten Programmnetzliste modelliert werden kann. Dabei wird die Idee anhand einer typischen Interrupt-basierten Softwarearchitektur vorgestellt, die über ein Set von Interrupt-Service-Routinen und einem zugehörigen Hauptprogramm verfügt.

Eine Interrupt-Service-Routine (ISR) ist meistens einem spezifischen Interrupt zugeteilt, den sie spezifisch bearbeiten soll. Dieser Interrupt wird von einer externen Hardwarekomponente erzeugt. Das Hauptprogramm implementiert ein spezifisches Set an Aufgaben, die für den Rest des Software-Systems als Interface dienen. Dabei kommuniziert das Hauptprogramm mit den ISRs über Shared-Memory-Variablen, wie es in Abbildung 4.10 dargestellt ist. Die ISRs lesen und schreiben zusätzlich die Geräteregister der externen Hardware, die als Eingabe- und Ausgabeports modelliert werden.

Das Gesamtmodell wird in zwei Schritten gebaut:

1. Generieren der individuellen Programmnetzlisten für die einzelnen Komponenten des Softwaresystems (ISRs, Hauptprogramm).

2. Verbindung der einzelnen Instanzen der Programmnetzlisten zu einem gemeinsamen Modell, um das globale Softwaremodell zu bauen.

Da Programmnetzlisten kombinatorische Schaltungen sind, lassen sie sich auf einfache Weise zu größeren Systemen zusammensetzen.

Ein Interrupt ist ein asynchrones Event, welches den Programmablauf theoretisch an jeder Stelle unterbrechen kann. Gewöhnlich prüft der Prozessor bei jeder neu geladenen Instruktion, ob ein Interrupt anliegt und das Programm unterbrochen werden muss oder ob die Instruktion ausgeführt wird. In der Programmnetzliste entspricht das der Stelle

zwischen zwei Instruktionszellen. Die Frage ist, wie oft eine einzelne ISR instanziiert werden muss, wenn sie an beliebigen Stellen im Hauptprogramm auftreten kann. Abbildung 4.9 zeigt auf der linken Seite ein Beispiel, in dem der Interrupt an allen Stellen auftritt. Im naiven Ansatz müsste für jede mögliche Position, an der das Programm unterbrochen werden kann, eine Instanz von der Interrupt-Service-Routine eingefügt werden, was offensichtlich nicht praktikabel ist.

Um ein vereinfachte kombinierte Programmnetzliste zu erstellen, die das Interruptverhalten der Software korrekt abbildet, können bestimmte Eigenheiten der Interrupts und des Hardwaretimings genutzt werden.

4.5.1 Vereinfachung mittels "transparenter Interrupts"

Da weder das Programm voraussehen kann, wann ein Interrupt eintritt, noch die ISR erkennen kann, welches Programm vor der Unterbrechung gerade lief, muss die ISR bei der Unterbrechung dafür sorgen, dass das unterbrochene Programm nach Beendigung der ISR an der Stelle weiterarbeitet, an der es unterbrochen wurde. Das bedeutet, dass der Architekturzustand des Prozesses beim Unterbrechen gespeichert und anschließend wiederhergestellt werden muss, damit er nicht durch den Interrupt und die zugehörige ISR beeinflusst bzw. zerstört wird und eventuell dadurch im unterbrochenen Programm ein fehlerhaftes Verhalten erzeugt.

Wenn die Interrupt-Service-Routine das Speichern und Wiederherstellen korrekt implementiert, ist ein Aufruf der ISR *transparent* für den Architekturzustand des aktuell laufenden Hauptprogramms. Diese Transparenz kann mittels einfacher Eigenschaftsprüfungen auf der Programmnetzliste der ISR bewiesen werden, und muss nur einmal für jede einzelne ISR durchgeführt werden.

Wenn die ISR jedoch transparent ist, wird ihr Effekt nicht unmittelbar im Hauptprogramm sichtbar, sondern nur über definierte Shared-Memory-Bereiche, über die die Kommunikation mit dem Hauptprogramm abgewickelt wird. Dank der beiden Programmnetzlisten können diese Schnittstellen mit einfachen Berechnungen identifiziert werden. Die zu kommunizierenden Programme müssen nur auf gemeinsame Speicherzugriffe abgeprüft werden. Dieser Schritt lässt sich gut automatisieren. Zuerst müssen alle Speicheradressen berechnet werden, bei denen das gleiche Verfahren wie in [BVS+14b] verwendet werden kann, mit dem auch die unbekannten Sprungziele von Kapitel 4.2.5 berechnet werden.

Im zweiten Schritt werden die gemeinsamen Adressen berechnet, wie es in [Feh12] vorgestellt wurde.

Die Erkenntnis, dass eine transparente ISR nur über Shared-Memory mit den anderen Programmteilen kommuniziert, ermöglicht eine erhebliche Vereinfachung des globalen Modells, da nur noch eine Instanz an den entsprechenden Kommunikationspunkten generiert werden muss. Abbildung 4.9 zeigt auf der linken Seite, wo der Interrupt tatsächlich passieren könnte und auf der rechten Seite, wie er im Modell entsprechend der Transparenzbedingung ersetzt werden kann.

Die Gesamtnetzliste wird zusammengesetzt aus dem Hauptprogramm und jeweils ei-

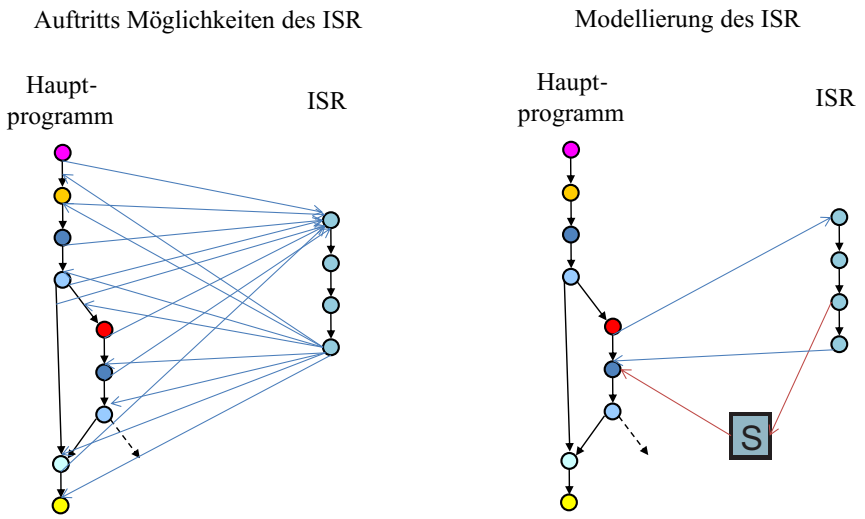

Abbildung 4.9: Interrupt Kommunikation mit dem Hauptprogramm

ner Instanz einer Interrupt-Service-Routine für jeden Abschnitt zwischen zwei Kommunikationspunkten und muss entsprechend verbunden werden. Abbildung 4.10 zeigt, wie die PNs des ISRs und des Hauptprogramms zusammengesetzt werden müssen. Dabei sind Main 1 bis Main 3 Teilblöcke, die zusammen die PN des Hauptprogramms bilden.

Angenommen, der Teil der Programmnetzliste (PN) namens Main 1 in Abb. 4.10 besteht aus einer Sequenz von 30 Instruktionen, in denen das Hauptprogramm nicht auf den Shared-Memory zugreift. Wenn der IRQ innerhalb dieser 30 Instruktionen passieren würde und die ISR ausgeführt wird, wird das Ergebnis trotzdem erst in dem Moment für das Hauptprogramm sichtbar, wenn das Programm nach diesen 30 Instruktionen wieder auf den Shared-Memory-Bereich zugreift. Dies passiert bei der ersten Instruktion in der PN Main 2. Das bedeutet, dass der genaue Zeitpunkt des Interrupts innerhalb dieser 30 Instruktionen irrelevant ist.

Es ist ausreichend, dass der Interrupt innerhalb dieser Instruktionen eintreten kann. Daraus resultiert, dass die Shared-Memory-Variable zwischen den beiden klar definierten Programmzuständen der PN beliebig durch die ISR modifiziert werden kann. Das wird mittels des Einfügens des kombinatorischen Logikblocks ISR-PN Ins. 0 “parallel” zur PN Main 1 erreicht. Dabei repräsentiert ISR-PN Ins. 0 eine Instanz der Interrupt-Service-Routine.

Ein weiterer Vorteil dank der typischen Beschaffenheit eines Interrupt-Handlers ist, dass sein Programmzustand vollständig im Arbeitsspeicher gespeichert sein muss, da er

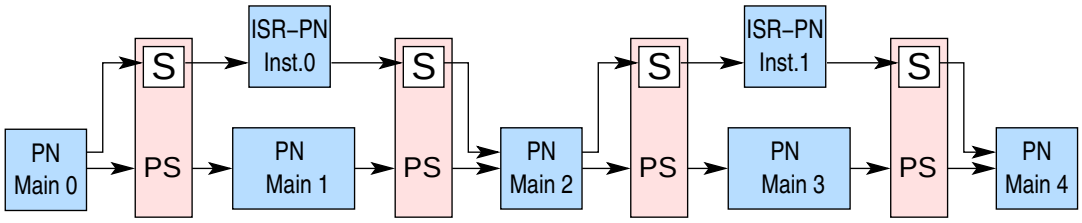

Abbildung 4.10: Beispiel: Modell für ein interrupt-getriebenes Programm

sonst beim nächsten Start keine aktuellen Daten mehr zur Verfügung hat. Das bedeutet auch, dass die ISR ihre Ausführung mit einem beliebigen Architekturzustand beginnen kann. Je nach Anwendungsfall ist es möglich, dass der Interrupt-Handler gar keinen Programmzustand zu speichern braucht und sich stattdessen beim Start immer selbst neu initialisiert. Daher sind die Programmnetzlisten, die die Instanzen der ISR repräsentieren, nicht miteinander durch den Architekturzustand verbunden. Dies reduziert zusätzlich die Komplexität des kombinierten, globalen Modells.

In Abbildung 4.10 sind zum Beispiel die Instanzen der ISR nur mit dem Teil des Speichers (S) verbunden, den sich das Hauptprogramm und die ISRs teilen. Grundsätzlich können Interrupt-Service-Routinen, wie jedes andere Programm, ebenfalls einen eigenen Speicherbereich haben, der ausschließlich zu deren Programmzustand gehört. Dieser wird dann über die Instanzen hinweg verbunden. Die Reihenfolge der auftretenden Instanzen der ISRs sind implizit gegeben durch die Programmnetzliste des Hauptprogramms und durch den aktiven Pfad darin.

Das Beispiel in Abbildung 4.10 modelliert lediglich eine einzelne Interrupt-Service-Routine und das zugehörige Hauptprogramm. Dabei kann ein Interrupt-getriebenes System über mehrere ISRs verfügen, die sich je nach Priorität gegenseitig unterbrechen können. Das Verfahren lässt sich auch ohne Veränderungen auf mehrere Interrupt-Routinen anwenden. Die zu erfüllende Randbedingung ist, dass die Priorität der Interrupts untereinander idealerweise statisch ist und eine Hierarchie berechnet werden kann. Ist das der Fall, wird das Verfahren wie oben beschrieben nicht nur für die Bestimmung der Schnittpunkte zwischen Hauptprogramm und ISR durchgeführt, sondern auch zwischen den ISRs selbst. Eine ausführliche Beschreibung dazu findet sich ebenfalls in [Feh12].

4.5.2 Interrupt-Timing

Die Anzahl der Instanzen einer Interrupt-Service-Routine innerhalb einer Programmnetzliste wird vor allem durch die Anzahl der Kommunikationsschnittpunkte zwischen der ISR und dem Hauptprogramm bestimmt. Dabei wird an jedem Kommunikationsschnittpunkt eine Instanz der ISR eingefügt. Was aber geschieht, wenn ein Interrupt innerhalb eines Bereichs zwischen zwei Kommunikationszeitpunkten zwei- oder mehrmals auftreten kann? Je nach Aufbau des Interrupts kann das dazu führen, dass das Hauptprogramm nicht mehr den Wert liest, der sich bei der Ausführung nur einer Instanz ergibt. Zum Beispiel wäre ein Zähler in einer ISR bei mehreren ISR-Aufrufen einige Schritte weiter, als

wenn nur eine Instanz ihn inkrementiert hätte. Um zu entscheiden, wie viele ISR-Aufrufe zu modellieren sind, werden zusätzliche Informationen benötigt, die vom tatsächlich verwendeten HW/SW-System abhängen. Daher muss entweder durch den Anwender oder durch zusätzliche Tools eine Zeitabschätzung des Systems erfolgen, wie oft ein Interrupt auftreten kann. Daraus resultierend lässt sich eine untere Schranke bestimmen und in Form der Anzahl an Instruktionen ausdrücken, die mindestens zwischen dem Auftreten zweier Interrupts ausgeführt werden können. Sollte eine Sequenz ohne Kommunikationspunkte länger sein, muss auch die entsprechende Anzahl an Instanzen eingefügt werden. Eine genauere Beschreibung der Berechnung befindet sich in [Feh12].

4.5.3 Zusätzliche hardwareabhängige Aspekte

Wenn von Interrupt-Modellierung gesprochen wird, gehören die Priorisierung der Interrupts und der Kontextwechsel dazu.

Die Auflösung von Interrupt-Prioritäts-Schemata wird durch die Hardware des Prozessors oder eines zusätzlichen Interrupt-Controllers realisiert. Es gibt statische und dynamische Schemata. Der Kontextwechsel ist ebenfalls plattformspezifisch. Dabei wird der Kontextwechsel durch die Hardware des Prozessors unterstützt und ein Teil durch Software realisiert. Alternativ kann die Hardware auch einen vollständigen Kontextwechsel durchführen (vergl. 2.3.2). Zunächst muss beim Kontextwechsel der aktuelle PC als Rücksprungadresse in ein spezielles Register gespeichert werden. Anschließend wird die Sprungadresse für das Folgeprogramm geladen. Diese beiden Aufgaben können ausschließlich mittels Hardware durchgeführt werden.

Wenn der Prozessor Verhalten implementiert, wie zum Beispiel die Rücksprungadresse zu speichern, dann ist das ein Element, welches zum korrekten Verhalten des Programmzustandes des ISR gehört, da dieser nach Beendigung als letztes die Rücksprungadresse zurück in den PC kopiert. Somit ist das Hardwareverhalten relevant für den Transparenzbeweis und auch für ein korrektes Modellieren des vollständigen Verhaltens.

Das für den vollständigen Programmablauf notwendige, in Hardware implementierte, Verhalten wird in der Programmnetzliste durch spezielle Hardware-Instruktionszellen ergänzt, so dass das Modell den Kontextwechsel vollständig implementiert. Es werden zwei Hardware-Instruktionszellen benötigt: Die erste ist die sogenannte *IRQ-init cell*, die den Anfang des Kontextwechsels modelliert und am Beginn eines jeden ISR eingefügt wird, wie in Abbildung 4.11 a). Sie beinhaltet den Part des Kontextwechsels, der von der Prozessorhardware durchgeführt wird. Dazu gehören die Sicherung des aktuellen PCs und je nach Plattform noch weitere Aufgaben. Die zweite Zelle ist die sogenannte *IRQ-decision cell*. Sie wird an den Stellen der Programmnetzliste eingefügt, an denen entschieden werden muss, ob die Priorität hoch genug ist, um den aktuell laufenden Prozess zu unterbrechen. Abbildung 4.11 b) zeigt den Aufbau der decision cell. Die Priority Unit berechnet, ob das eingehende IRQ-Event wichtiger ist als der gerade aktuell laufende Pfad. Die Priority-Unit-Zelle kann sowohl mit statischer als auch dynamischer Priorität – entsprechend der gegebenen Architektur – implementiert sein. Bei einer höheren Priorität wird das Aktiv-Signal des aktiven Pfades an die ISR weiter propagiert. Ist der Pfad des

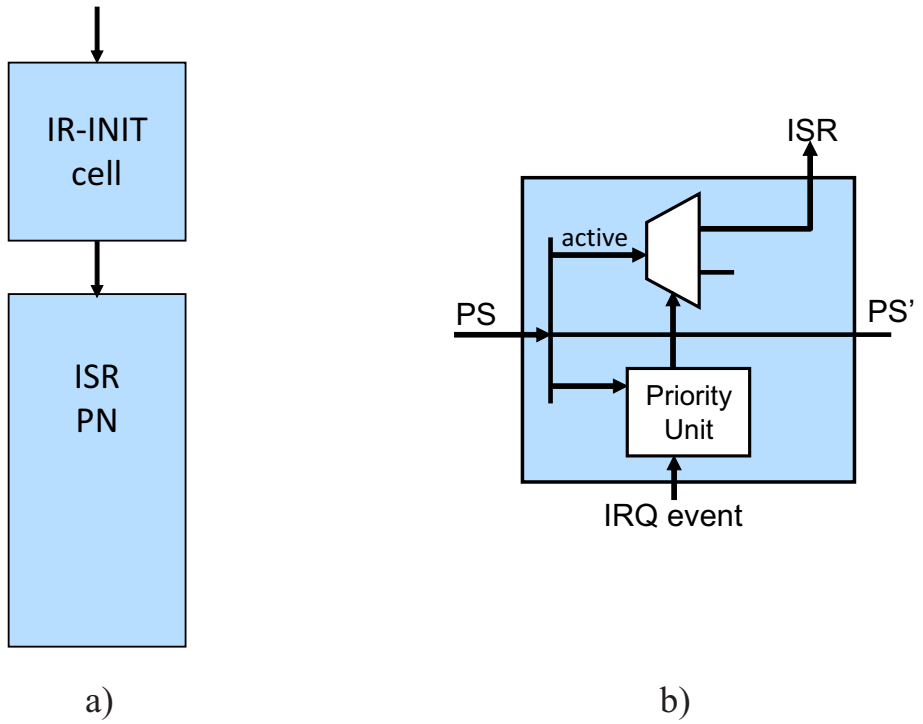

Abbildung 4.11: Instruktionszellen: a) IRQ-init cell b) IRQ-decision cell

Programms nicht aktiv, kann auch die zugehörige Interruptroutine nicht aktiv werden. Eine genaue Beschreibung dieser Zellen, und wie diese in die Programmnetzliste eingefügt werden, ist in [SVF+13] und [Feh12] zu finden.

Kapitel 5

Anwendungen der Programmnetzliste

Dieses Kapitel stellt zwei mögliche Anwendungen der Programmnetzliste vor und beschreibt zusätzlich Ideen, die den Umgang mit der Programmnetzliste im praktischen Einsatz verbessern können.

Das in den Kapitel 3 und 4 beschriebene Modell dient als Grundlage für die formale Verifikation von hardwarenaher Software. Das Modell kann zum einen für die Eigenschaftsprüfung (engl. *Property Checking*) und den Äquivalenzvergleich (engl. *Equivalence Checking*) verwendet werden. In Property Checking (s. Kapitel 5.1) wird Verhalten der Software gegenüber einer Spezifikation oder allgemeiner gegen ein gewünschtes Verhalten verifiziert. Equivalence Checking erlaubt den Vergleich unterschiedlicher Implementierungen auf ein äquivalentes Verhalten und wird in Kapitel 5.2 beschrieben.

Zusätzlich wird eine Eigenschaft der Programmnetzliste erörtert, die genutzt werden kann, um zu überprüfen, ob es Fehler in der Generierung der Programmnetzliste gab. Im letzten Unterkapitel 5.4 wird der Umgang mit Endlosschleifen und nicht berechenbaren Schleifen beschrieben, der im Anschluss in den Experimenten (Kapitel 7) auch angewendet wird.

5.1 Property Checking

Beim Property Checking werden Eigenschaften (engl. *Properties*) spezifiziert und überprüft, ob diese Eigenschaften auf dem Modell gültig sind. Als Modell wird hier die Programmnetzliste verwendet. Mit ihr ist es möglich das Verhalten von Software auf dem betrachteten Prozessor formal zu verifizieren.

Es können unterschiedliche Verifikationsziele betrachtet werden. Im einfachsten Fall arbeitet die Software *nicht-reaktiv*, d.h. aus den Eingaben an das Programm wird ohne weitere Interaktion mit der Umgebung ein Ergebnis berechnet, dessen Korrektheit überprüft werden soll. Auf welchem Weg das Ergebnis erzeugt wird, spielt dabei keine Rolle; entscheidend ist nur, dass das Ergebnis korrekt berechnet wurde. In diesem Szenario wird eine Eigenschaft formuliert, die sich lediglich auf die Eingaben und die Ausgaben des Programmnetzliste bezieht. Die Programmnetzliste wird sozusagen als “Black Box” betrachtet.

Bei eingebetteter Software, die mit der Hardware interagiert, funktioniert dieser Black-Box-Ansatz nicht. Die Betrachtung der Ein- und Ausgaben von der Programmnetzliste zur Peripherie wird in diesem Fall relevant. Ein wesentliches Merkmal dabei ist, dass die Kommunikation mehrfach über ein und dieselbe Schnittstelle erfolgen kann. So werden zum Beispiel bei Protokollen mehrfach Daten gesendet, die aber jeweils als Sequenz an dasselbe Register in der Hardware geschrieben werden. Würde nur nach Ausführung des Programms verifiziert, welcher Wert an der zu prüfenden Speicherstelle geschrieben wurde, wäre nur der letzte geschriebene Wert sichtbar. Eine Aussage über die Kommunikation wäre damit nicht mehr möglich.

Deswegen müssen *Sequenzen* von Kommunikationsoperationen verifiziert werden. Die Sequenz ist über die Pfade definiert und ist im Execution-Graph und somit ebenfalls in der zugehörigen Programmnetzliste eindeutig den einzelnen Kommunikationsoperationen (z.B. LOAD- oder STORE-Instruktionen, die auf Peripherieregister zugreifen), zuzuordnen. Das erleichtert die Generierung der Eigenschaften und kann den Aufwand für Verifikation erheblich vereinfachen.

Eine Verifikation eines Protokolls – eine Reihenfolge von Schreiboperationen auf unterschiedliche Adressen der Peripherie – wird möglich, indem Sequenzen von Kommunikationsoperationen auf die verschiedenen Adressen spezifiert werden. Zum Beispiel sollte ein Programm erst die Konfiguration an eine UART übermitteln, bevor es beginnt die Daten zu übermitteln. In einem solchen Fall ist die Reihenfolge der Schreiboperationen der Software auf die unterschiedlichen Register der Hardware entscheidend.

Es müssen die Sequenzen für die Hardware- bzw. Speicherzugriffe bestimmt werden. Die sind im Graphen – wie bei den Sequenzen auf eine Speicherstelle – statisch. Durch die unterschiedlichen Pfade in der Software kann deren Berechnung aufwendiger werden. Wird ein SAT-Solver zur Verifikation dieser Sequenzen eingesetzt, muss eine zusätzliche Logik implementiert werden, um die Halbordnung der Lese- und Schreiboperationen in der Programmnetzliste für den SAT-Solver sichtbar zu machen.

Eine genaue Beschreibung, wie die zusätzliche Logik erzeugt werden muss, ist in Appendix B zu finden und wird in den Experimenten in Kapitel 7 angewendet.

5.2 Equivalence Checking

Beim Equivalence Checking geht es darum, zwei Implementierungen auf semantische Äquivalenz zu vergleichen. So wird z.B. Equivalence Checking beim Entwurf digitaler Schaltungen verwendet, um die verschiedene Beschreibungen eines Designs, z.B. RTL-Beschreibung und Gatternetzliste, auf funktionale Äquivalenz zu überprüfen. So wird gezeigt, dass Schaltungstransformationen oder Optimierungen das funktionale Verhalten des Ursprungsmodells nicht verändert haben. Der Hauptvorteil dieses Verfahrens ist, dass es zum größten Teil voll automatisiert werden kann.

Durch die Art des Aufbaus der Programmnetzliste kann eine Äquivalenz zwischen unterschiedlichen Softwareimplementierungen mit Hilfe der PN durchgeführt werden. Es gibt verschiedene Anwendungsszenarien, in denen ein Äquivalenzvergleich den Entwick-

ler unterstützen kann.

Ein Szenario ist beispielsweise, wenn ein Treiber weiterentwickelt wird, aber noch zum alten Standard kompatibel sein muss. Dann kann der neue Code mit der aktuellen Implementierung verglichen werden. Auch wenn der geschriebene Code durch unterschiedliche Optimierungsstufen eines Compilers optimiert wurde, kann es von Vorteil sein zu sehen, ob die Programme noch äquivalent zueinander sind, oder ob sie sich durch eine Überoptimierung unterscheiden.

Die Programmnetzliste kann auch für den Äquivalenzvergleich einer Softwareportierung von einer Plattform auf eine andere genutzt werden.

Dabei ist der Schwerpunkt bei der hardwarenahen Softwareverifikation auf die Äquivalenz der Ein- und Ausgabesequenzen der gegebenen Programme gelegt.

Um das zu ermöglichen wird eine zusätzliche Logik zu den Programmnetzlisten generiert, mit der der Äquivalenzvergleich auf einem SAT-Solver durchgeführt wird. Eine ausführliche Beschreibung bietet die Dissertation von Carlos Villarraga [Vil16]. Einen guten Überblick über das Thema gibt [VSB+13].

5.3 Überprüfung der Programmnetzlistenerstellung

In Kapitel 4 wurde gezeigt, wie ein Modell generiert wird, das alle Ausführungspfade der zu analysierenden Software abbildet. Um unnötige Verzweigungen zu minimieren, werden bei der Modellgenerierung Verzweigungen immer auf Lebendigkeit geprüft und nur weiter ausgerollt, wenn die Verzweigungsbedingung erfüllt ist. In Kapitel 4.3.1 wurde gezeigt, wie es möglich ist, die Instruktionszellen gegenüber der Originalimplementierung formal zu verifizieren. Beides zusammen führt zu einer vollständigen und korrekten Darstellung der Software durch die Programmnetzliste. Das Zusammensetzen ist durch die einfache 1:1 Ersetzung fehlerfrei zu implementieren. Hier wird zusätzlich eine einfache Methode gezeigt, die überprüft, ob beim Zusammensetzen alle betrachteten Pfade noch berücksichtigt sind, oder eventuell durch einen Fehler bei der Generierung eine Änderung im Pfadverhalten möglich wird.

Im Fehlerfall sollte die Programmnetzliste nicht mehr, wie angenommen, alle gültigen Ausführungspfade des Programms korrekt abbilden. Ein solcher Fehlerfall kann etwa durch einen als tot markierten Pfad auftreten, der im fehlerhaften Modell wieder lebendig ist. Oder durch fehlerhaftes Zusammensetzen zweier Zellen können in der Programmnetzliste neue Verzweigungen entstehen, die unerwartetes Verhalten erzeugen könnten, so dass ein Pfad nicht wie erwartet endet.

Ein weiterer Sonderfall kann sein, dass etwa eine Sprunginstruktion ein Sprungziel generiert, das beim Generieren der Sprungliste nicht erkennbar war. Sowohl die beschriebenen möglichen Fehlerquellen als auch alle weiteren pfadverändernden Fehler können mit einer einfachen Eigenschaft gefunden werden.

Gibt es keine anderen Pfade, als die im Modell erzeugten, sollte die Eigenschaft aus Listing 5.1 auf der Programmnetzliste gültig sein und kein Gegenbeispiel erzeugen.

Die Idee der Eigenschaft ist, dass alle vom Programm vorgesehene Ausgänge, in de-

nen das betrachtete Programm terminiert, in der Programmnetzliste auf einen gemeinsamen *Endknoten* zusammengeführt werden. Dieser ist ein künstlicher Knoten, der nicht Bestandteil der ursprünglichen Software ist. Er besitzt keine Operationen und wird an die Programmnetzliste angehängt. Somit besitzen alle Pfade in der Programmnetzliste nur einen gemeinsamen Endpunkt.

Listing 5.1: Eigenschaft zum Prüfen der Korrekheit der PN

```
1 property Program_Netlist_check is
  dependencies: constraint_list;

  assume:
5 at t: s.active = true;

  prove:
  at t: z.active = true;

10 end property;
```

Wenn es durch einen Fehler im Modell einen Pfad gibt, der nicht mehr in diesem Endknoten terminiert, so terminiert das zugehörige Active-Signal, das mit den betroffenen Pfaden verknüpft ist, ebenfalls nicht mehr im Endknoten. Die Eigenschaft überprüft, dass das Active-Signal im Endknoten immer aktiv ist, wenn ein Active-Signal am Startzustand der Programmnetzliste gegeben wird. Wenn es einen Pfad gibt, der nicht im Endknoten endet, wird der SAT-Solver ein entsprechendes Gegenbeispiel geben, in dem der fehlerhafte Pfad als aktiv markiert ist. Ist die Programmnetzliste korrekt, wird die Eigenschaft auf dem Modell immer gültig sein.

Um die Eigenschaft in Listing 5.1zu beschreiben wurde ITL als Sprache verwendet. Die Eigenschaft überprüft, ob das am Eingang, dem Startzustand s angelegte Active-Signal auf jeden Fall bei dem eingeführten Endknoten ankommt. z ist der Architekturstatus am Ende des Programms und der Ausgang des Endknotens. Wenn das Verifikationstool die Aussage widerlegen kann, ist das Modell vermutlich fehlerhaft. Anhand des Gegenbeispiels kann dann ziemlich einfach erkannt werden, an welcher Stelle der Fehler vorliegt. Die “dependencies” in der zweiten Zeile des Listing 5.1 sind zusätzliche Informationen, die übergeben werden können, wie z.B. externe Vorgaben für Wertebereiche von Speicherzellen.

Alle in diesem Kapitel beschriebenen Fehlerfälle sollten bei korrekter Modellgenerierung nicht auftreten. Da aber die Generierung mittels Software entstanden ist und die Fehlerfreiheit der Anwendungssoftware denselben Einschränkungen unterliegt wie die Software, die auf Fehlerfreiheit verifiziert werden soll, bringt dieser Nachweis zusätzliche Sicherheit und kann frühzeitig mögliche Fehler im Modell aufdecken, bevor die eigentliche Verifikation auf dem Modell durchgeführt wird.

5.4 Unbestimmte Schleifen

Schleifen setzen sich auf Maschinencode-Level immer aus einer Sequenz zusammen, die am Ende wieder zum Anfang zurückspringt. Dieser Sprung ist in der Regel ein unbedingter Sprung. Dafür gibt es eine Stelle in der Sequenz, an der die Abbruchbedingung der Schleife abgeprüft wird und ein bedingter Sprung entscheidet, ob die Schleife aufs Neue ausgeführt wird.

Beim Ausrollen des CFG zum EXG wird diese Schleife abgerollt. Dabei werden, wie schon in Kapitel 3 beschrieben, die Bedingungen immer mittels eines Verifikationsaufrufs abgeprüft, welche Verzweigung genommen wird. Zählschleifen können so erfolgreich abgerollt werden, da ihre Randbedingung ausschließlich von Informationen abhängen, die im Programm direkt enthalten sind. Bedingt durch das Verhalten des Programms ist es ebenfalls möglich, dass es eine implizite im Code versteckte Abbruchbedingung gibt, die nicht direkt ersichtlich ist.

Ein Beispiel dafür ist die Multiplikation mittels Addition und Shift-Operationen. Spätestens, wenn der Multiplikator vollständig durch das Register geschoben wurde, ist die Multiplikation fertig. Es ist ausreichend, wenn die Abbruchbedingung überprüft, ob der Wert des Multiplikator Null beträgt. Ist der Wert schon vor dem Erreichen der maximalen Schleifenzahl (Registerbreite) Null, kann schon vorzeitig gestoppt werden. Während das vorzeitige Stoppen von dem eingelesenen Multiplikator abhängt, ist der endgültige Stopp dadurch bedingt, dass die Abbruchbedingung auf jeden Fall erfüllt ist, da das komplette Register durch das Schieben mit Nullen aufgefüllt wurde.

Ist der Multiplikator aus dem Kontext des Programms bekannt, kann der Algorithmus zur Generierung der Programmnetzliste so die Grenze erkennen, wenn die Maximalzahl an Schleifendurchläufen erreicht wurde, auch wenn sie nicht explizit im Code implementiert wurde.

Wie bei der Zählschleife und der Multiplikation können alle Verzweigungen, die auf Bedingungen beruhen, die irgendwo im Programm codiert sind, mit dem in Kapitel 3 vorgestellten Algorithmus ebenfalls ohne zusätzlichen Aufwand aufgelöst werden und eine komplette Programmnetzliste erzeugen.

Ist die Sprungbedingung hingegen bekannt und konstant gültig im Programm, wäre eine Endlosschleife implementiert und somit das Schleifenende unbestimmt bzw. nicht vorhanden. Da anhand dieser Bedingungen einfach ersichtlich ist, dass immer wieder die gleiche Verzweigung bei der bedingten Verzweigung genommen wird, kann der Verifikationsingenieur schon im Vorfeld entscheiden, wie diese Endlosschleife behandelt werden soll.

Als letzter Fall kann es auch vorkommen, dass eine Rücksprungbedingung immer erfüllt ist, aber dies nicht unmittelbar ersichtlich ist. Dann ist ebenfalls eine Endlosschleife implementiert, aber diese ist nicht leicht zu erkennen. Die dazugehörige Generierung der Netzliste würde nicht abbrechen.

Im folgenden Abschnitt 5.4.1 wird der Umgang mit solchen Schleifen genauer beschrieben.

Sollte der bedingte Sprung von externen Werten abhängen, die nicht vom Programm

selbst generiert worden sind, ist die Verzweigung in der Regel nicht entscheidbar, wie in Kap 4.2.6 gezeigt wurde. Das führt dazu, dass beide Fälle immer lebendig sind und im Execution-Graph weiter ausgerollt werden. Wenn eine solche nicht entscheidbare Verzweigung die Bedingung für den Abbruch einer Schleife ist, dann wird neben dem Pfad für den Abbruch ebenfalls die Verzweigung zum erneuten Durchlaufen der Schleife immer wieder als lebendig erkannt. Die Schleife wird dann ebenfalls endlos abgerollt.

Ein Programm, das Endlosschleifen enthält, hätte bei einem eingebetteten System eine unbeschränkte Laufzeit. Wegen der Echtzeitanforderungen und der Ressourcenknappheit in eingebetteten Systemen ist ein deterministisches Programm mit einer bekannten Laufzeit gewünscht. Daher ist es wahrscheinlich, dass der Programmierer des Programms zusätzliches Wissen über die einzulesenden Werte besitzt. Sollte das der Fall sein, ist es auch möglich, diese Werte als Constraints für die Eingaben zu nutzen, um die Generierung der Programmnetzliste entsprechend zu beschränken. Wenn solche Werte nicht vorliegen bzw. nicht bekannt sind, kann das ein Hinweis auf einen Fehler im Code sein. Um endlose Schleifen im Tooling erkennen zu können, wird in Kapitel 6 eine künstliche Obergrenze in das Tool integriert. Die Entscheidung über die Höhe der Grenze oder darüber, ob im Fall des Erreichens wirklich ein Fehler vorliegt, kann nur manuell und fallspezifisch entschieden werden.

Ist das endlose Abrollen erwünscht oder erwartet, ist es nicht möglich, ein vollständiges Pfadmodell zu erstellen, da es einen Pfad gibt, der nie abbricht. An diesem Punkt scheitert die automatische Generierung der Programmnetzliste.

Viele Eigenschaften, die mittels Intervall Property Checking auf einer gegebenen Hardware bewiesen werden können, benötigen kein komplett ausgerolltes Modell, in dem alle erreichbaren Zustände vom Start aus erreicht werden können. Es ist ausreichend, nur die notwendige Anzahl an Instanzen über die Zeit auszurollen, um die Länge der zu beweisenden Eigenschaft zu überdecken. Diese Art des Beweisens kann auch in der Programmnetzliste angewendet werden. Anstelle die komplette Netzliste zu generieren, können ebenso nur beschränkte Teile der Netzliste generiert werden, indem der Verifikationsingenieur künstliche Beschränkungen einführt. Auf diesem beschränkten Modell können dennoch viele Eigenschaften bewiesen werden.

Durch die Möglichkeit, einzelne Programmnetzlisten zur einer größeren zu kombinieren, gibt es die Möglichkeit, das Programm in einzelnen Fragmenten zu generieren und diese dann nach Bedarf wieder zusammen zu setzen. Ein solches einzelnes Fragment besitzt keine Vorgeschichte, daher sind alle möglichen Eingaben erlaubt, die vorher durch das Programm eingeschränkt wurden.

Der Algorithmus zum Generieren der Teil-Programmnetzliste kann mehr Pfade erzeugen, als an dieser Stelle im Programm real auftreten können. Das entspricht einer Überapproximation der möglichen Pfade, schließt aber keine real vorkommenden Pfade aus. Eigenschaften, die auf der Teilnetzliste bewiesen werden können, sind dann auch auf dem beschränkteren, realen Programm gültig. Somit ist die Aussagekraft der Eigenschaft nicht eingeschränkt. Umgekehrt ist es jedoch möglich, dass Eigenschaften die auf der Teil-PN nicht gelten, auf der vollständigen Netzliste an dieser Stelle dennoch gültig wären.

Im Kapitel 5.4.2 wird gezeigt, wie diese Zerlegbarkeit der Programmnetzliste ver-

wendet werden kann, um längere Schleifen manuell aufzubrechen und zu verkürzen. Das kann bei Programmen interessant werden, die zum Beispiel aktiv auf Eingaben warten und währenddessen keine weiteren Berechnungen durchführen.

Im Experiment 7.4 wird das gezielte Zerteilen der Programmnetzliste in Teilkomponenten angewendet. Dort wird gezeigt, wie ein Programm gesplittet und mit den einzelnen Teilen das notwendige Gesamtverhalten implementiert wird. Auch die erfolgreiche Verifikation von Eigenschaften auf dem zusammengesetzten Modell wird durchgeführt.

Im Falle einer endlosen Schleife, die künstlich abgebrochen wird, lässt sich natürlich keine zeitliche Einschränkung des Programms mittels der Programmnetzliste zeigen. Da sie aber je nach Implementierung der Zellen keine reale Zeit implementiert und solche Aspekte wie Cache-Miss oder Wait-States auf dem Bus nicht berücksichtigen kann, ist eine Aussage über das echte Zeitverhalten des Programms in der Programmnetzliste generell schwierig bzw. nicht möglich.

5.4.1 Behandlung von endlos langen Schleifen

Wenn eine Endlosschleife in einem Programm implementiert ist, sei es durch unbedingte Verzweigungen, durch indirekte oder unbekannte Schleifenbedingungen, ist ein zusätzlicher Eingriff in die vorgestellte Methode zur Generierung der Programmnetzliste notwendig, da sie von sich aus nicht abbrechen würde. In dem Fall, dass der Wert der Schleifenbedingung von der Peripherie abhängt, kann das Hinzufügen von Beschränkungen, die durch die Umgebung gegeben sind, das Problem lösen. Somit werden die möglichen Werte bei der Schleifenbedingung eingeschränkt. Die Beschränkungen können als Constraints bei der Modellgenerierung mit aufgenommen werden und beeinflussen so die Lebendigkeitsprüfung, wodurch auch die Anzahl der Schleifendurchläufe beschränkt wird.

Sind Verzweigungen zur Wiederholung der Schleifen dauerhaft durch direkt im Programm implementierte Bedingungen erfüllt, müssen diese Schleifen gesondert erkannt und behandelt werden. Das Erkennen kann manuell durch den Ingenieur geschehen, der das Programm geschrieben hat, oder auch mit Hilfe eines zusätzlichen Tools. Wenn eine solche Schleife erkannt wird, gibt es folgende Möglichkeiten:

- maximale Anzahl an auszurollenden Schleifen angeben
- Schleife nur einmal ausrollen und benötigte Anzahl an Instanzen in der Gesamtnetzliste zusammensetzen

Durch beide Möglichkeiten wird die Schleifentiefe künstlich begrenzt. Damit ist die Aussagekraft der resultierenden Programmnetzliste auf Eigenschaften beschränkt, die sich innerhalb dieser Schleifenanzahl beweisen lassen. Die Allgemeingültigkeit für alle Beweisaussagen muss in dem Fall zusätzlich um den Aspekt der beschränkten Schleifentiefe erweitert werden. Wenn in der betrachteten Eigenschaft gezeigt werden kann, dass sie nicht über die gegebene Schleifentiefe hinausgehen kann, dann ist die bewiesene Eigenschaft allgemeingültig. Sollte die Eigenschaft Aspekte abprüfen können, die über die

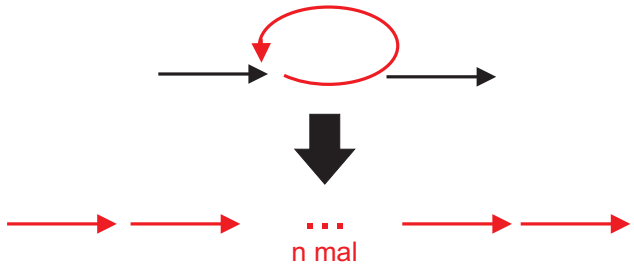

Abbildung 5.1: Ausrollen einer Schleife

Schleifentiefe hinausgehen, dann gilt sie nur beschränkt für die Anzahl der ausgerollten Schleifen.

Statt durch ein beschränktes Ausrollen des Gesamtprogramms kann eine Schleife einzeln ausgerollt werden. Ohne den voranstehenden Programmteil wird diese ausgerollte Schleife eine Überapproximation der tatsächlich möglichen Werte beinhalten, da der Anfangszustand nicht mehr durch das Programm beschränkt wird. Auf dieser einzeln ausgerollten Schleife lassen sich Eigenschaften, die nur eine Iteration der Schleife betreffen, gezielt verifizieren. Ist es möglich, Eigenschaften auf dem allgemeineren Schleifenmodell zu beweisen, so gelten die Eigenschaften auch für die Schleife mit zusätzlich beschränkten Anfangsbedingungen. Da diese Anfangsbedingungen durch den der Schleife vorangegangene Teil der Programmnetzliste beschrieben werden können, gelten die Eigenschaften ebenfalls für die Schleife in der Programmnetzliste.

Je nach der zu beweisenden Eigenschaft kann eine einzige Instanz ($n = 1$) schon ausreichend sein. Sollen Eigenschaften bewiesen werden, die nicht mittels einer einzelnen Instanz der Schleife bewiesen werden können, kann, wie beim Interval Property Checking, die Schleife beliebig oft aneinandergehängt werden und dabei mit einer beliebigen Startbedingung in der Schleife starten, wie in Abbildung 5.1. So kann mit dem offenen Startzustand jeder beliebige Ausrollzeitpunkt der Schleife als Startpunkt angenommen werden. Somit lässt sich jedes Verhalten an jeder beliebigen Stelle der ausgerollten Schleifen testen, sofern die Eigenschaft selbst nur auf einer beschränkten Anzahl von Schleifen zugreift, ohne dabei eine absolute Position in den ausgerollten Schleifen zu zeigen.

Eine weitere Möglichkeit, das Modell zu vereinfachen, ist es mit Hilfe von Eigenschaften das Verhalten der einzelnen Schleife zu verifizieren. Aus diesem verifizierten Verhalten lässt sich dann ein abstrakteres Verhalten in Form eines neuen Basic-Blocks darstellen, der dann die vollständige Sequenz der Schleife beinhaltet und den Graphen vereinfacht. Dieser Basic-Block kann als neue Instruktion eingeführt werden und dann in einer neuen Version der Programmnetzliste verwendet werden.

5.4.2 Beweis von Idle Funktionen

Schleifen, die beliebig lange ausgerollt werden können, entstehen oft, wenn Programmteile aktiv auf ein Event warten und ansonsten nichts tun. Im Kapitel 7.4 wird ein solches

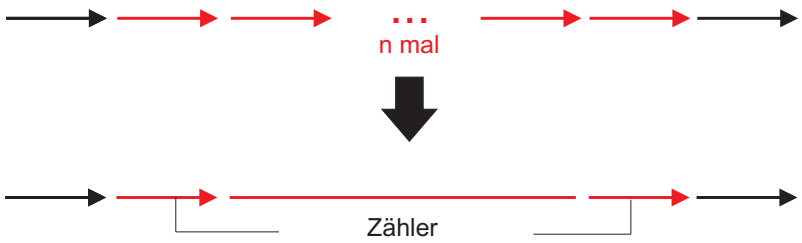

Abbildung 5.2: Verkürzen des aktiven Wartens

Programm besprochen, wo das Hauptprogramm aktiv wartet, bis vom Interrupt eine bestimmte Information gesetzt wird.

Während des aktiven Wartens mittels einer Schleife sollte sich der Programmzustand nicht verändern. Das kann mit einer Eigenschaft beschrieben werden, die überprüft, dass der Zustand nach dem Ausführen der Schleife identisch zum Zustand vor der Ausführung einer Schleifeniteration ist, sofern das erwartete Event nicht eingetreten ist. Ist der Beweis möglich, können Implementierungen, die mit Hilfe von Schleifen aktiv warten, vereinfacht werden. Die Schleife muss nicht mehr beliebig lange ausgerollt werden, sondern kann durch eine Iteration ersetzt werden, bei der das zu erwartende Event eintritt. Abbildung 5.2 zeigt abstrakt, wie die Schleife in der Programmnetzliste ersetzt werden kann.

Zu beachten ist, ob zusätzliche Zähler im Programm vorhanden sind und diese innerhalb der Schleife aufaddiert werden, um ein Timeout zu ermöglichen. Ob der Zustand unverändert ist, wird bei einem Zähler nicht nachweisbar sein. Das kann umgangen werden, indem der Zähler aus der Betrachtung außen vor gelassen wird. Lässt sich ein unveränderter Zustand in der Schleife ohne den Zähler beweisen, kann die Schleife genau wie in Kapitel 5.4.1 ersetzt werden. Zusätzlich muss der Zähler im resultierenden Modell, in dem nur die reduzierte Anzahl an Schleifen generiert wurde, separat berücksichtigt werden. Anstelle eines konkreten Wertes, kann der Zähler, abhängig von den Schleifendurchläufen beim Warten, eine Menge unterschiedlicher Werte besitzen. In BMC und IPC kann das durch einen offenen Eingang modelliert werden, der einen beliebigen Wert annehmen kann. Das finale Modell sieht dann aus wie in Abbildung 5.2. Der Zählerzustand wird nicht weiterverwendet, und nach dem aktiven Warten wird ein beliebiger Wert über den offenen Input gelesen. Bei Bedarf kann dieser offene Input durch zusätzliche Constraints beschränkt werden, wenn bekannt ist, welchen Zahlenbereich der Zähler annehmen kann.

Kapitel 6

Tool

Die in Kapitel 3 und 4 vorgestellte Methode wurde im Tool “Formal Checker Kaiserslautern” (FCK) implementiert und ist in C/C++ geschrieben. Abbildung 6.1 zeigt den überwiegend automatisierten Toolflow.

Als Eingabe benötigt FCK den Maschinencode des zu verifizierenden Programms. Ausgehend vom Assemblercode, C/C++-Code, gemischten Assembler- und C-Code oder von anderem Code, die auf der entsprechenden Zielplattform kompiliert werden können, werden alle Programme in Maschinencode übersetzt. Aus dem Maschinencode wird der Kontrollflussgraph (CFG) extrahiert. Dafür muss für jede unterstützte Zielarchitektur ein Parser zur Verfügung gestellt werden. Ein Parser für den OpenCore Aquarius, der für die Experimente in Kapitel 7 verwendet wird, wurde im Rahmen der Arbeit [Feh11] implementiert. Der extrahierte Kontrollflussgraph kann unvollständig sein. Die Funktion build_EXG aus Abbildung 6.1 nimmt als Eingabe den CFG und startet den Aufbau des Execution-Graphen. Erreicht die Methode beim Abrollen des CFG einen Punkt, an dem mittels SAT oder Simulation Werte ermittelt oder Verzweigungsbedingungen geprüft werden müssen, so wird mit dem bis dahin generierten EXG (incompl. EXG) eine vorläufige Programmnetzliste generiert. Diese basiert auf den Instruktionszellen (IC), die dem Tool für die jeweilige Architektur zur Verfügung stehen. Wenn die erforderlichen Werte ermittelt wurden, wird die Funktion build_EXG fortgesetzt, um den Execution-Graph zu vervollständigen. Bei Bedarf wird wieder eine neue vorläufige Programmnetzliste erzeugt, um weitere unbekannte Werte zu ermitteln oder Verzweigungen auf Lebendigkeit zu überprüfen.

Wenn die Funktion build_EXG aus Abbildung 6.1 den EXG final gebaut hat, wird daraus mit den Instruktionszellen die finale Programmnetzliste generiert. Im Anschluss wird die eigentliche formale Verifikation der hardwarenahen Software mit Hilfe der erstellten Programmnetzliste durchgeführt.

Die final erstellte Programmnetzliste kann in jedes beliebige Verifikationstool geladen werden, welches als Eingabe VHDL unterstützt. Dort kann das Programm entsprechend auf korrektes Verhalten formal verifiziert werden. Das ist möglich, weil die Instruktionszellen in einer Sprache modelliert werden, aus der sich einfach eine VHDL-Beschreibung generieren lässt. Diese Sprache ist dem VHDL sehr ähnlich und basiert auf den für die

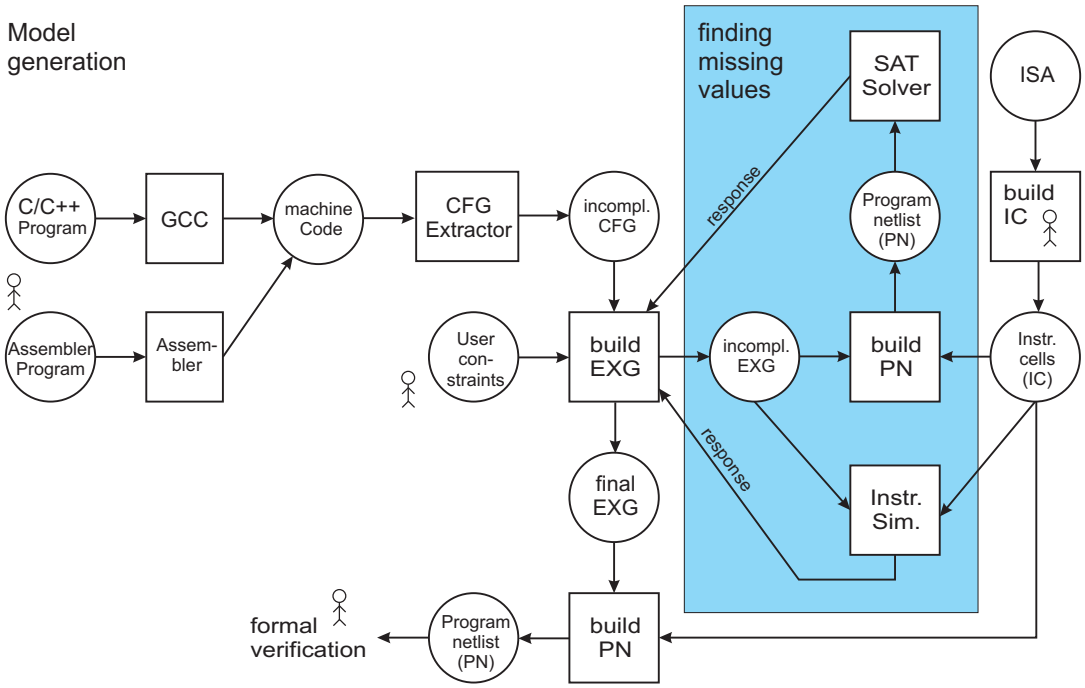

Abbildung 6.1: Verkürzen des aktiven Wartens

Instruktionszellen notwendigen synthetisierbaren Elementen, die in VHDL eins zu eins ebenfalls als Sprachkonstrukte zur Verfügung stehen. Die Instruktionszellen werden in maschinenlesbaren Form als XML-Dateien gespeichert. Diese XML-basierte Hardwarebeschreibung wurde im Rahmen der Arbeit [Wil09] entworfen und wird seitdem am Lehrstuhl weiterentwickelt.

Nicht nur das finale Werkzeug zur formalen Verifikation der Software kann beliebig ausgetauscht werden, sondern auch bei der Generierung des Execution-Graphen können beliebe SAT- oder auch SMT-Solver verwendet werden. In der aktuellen Implementierung haben wir MiniSAT [ES, ES03] als Solver eingesetzt. Über ein definiertes Interface gibt es die Möglichkeit neben anderen akademischen Solvern auch kommerzielle Tools zu verwenden. So kann zum Beispiel auch OneSpin Solution MV360 [One] über diese Schnittstelle angesprochen werden. Die Herausforderung liegt bei der Erweiterung der unterstützten Werkzeuge vor allem darin, wie die Werkzeuge gesteuert werden. Für OneSpin wurde deswegen ein TCL [TCL88]-basierter Server implementiert, der innerhalb des OneSpin-Tools gestartet wird. Der Server empfängt die Kommandos von FCK und leitet sie an OneSpin weiter. Der im FCK verwendete Simulator wurde im Rahmen der Arbeit [Bar13] entwickelt und ist ein Instruction-Set-Simulator, der alle möglichen Pfade simuliert.

Um den Umgang von Endlosschleifen, der in Kapitel 5.4.1 besprochen wurde, auch im Tool umzusetzen zu können, ist es möglich zu jedem Knoten im Kontrollflussgraphen anzugeben, ob er nur beschränkt oft abgerollt werden darf. Soll die Anzahl der Knoten eingeschränkt werden, kann noch angegeben werden, wie oft dieser Knoten im Graphen

erzeugt werden darf. Dabei ist zu beachten, dass der Knoten – bedingt durch Verzweigungen – möglicherweise mehrfach an unterschiedlichen Stellen im Execution-Graph instanziiert wird. Durch das Mergen bei der Generierung der Netzliste ist es unwahrscheinlich, dass dieselbe Instruktion eines Programms mehrfach im Execution-Graph vorkommt, sofern sie nicht auf demselben Pfad – z.B. innerhalb einer ausgerollten Schleife – hintereinander liegen. Daher kann die Obergrenze so gut beschrieben werden.

Um den Verifikationsingenieur bei seiner Arbeit zu unterstützen verfügt FCK über ein Zusatztool zur graphischen Darstellung des Execution-Graphen und der Programmnetzliste. Neben der Graphenstruktur kann es zusätzlich die Informationen für jeden Knoten darstellen, wie zum Beispiel Instruktionstyp, Parameter, Vorgänger und Nachfolger. Das Tool basiert auf der Graphischen Library udraw [Uni05] der Universität Bremen.

6.1 Modellierung von Speicher

Der Programmzustand setzt sich, wie in Kapitel 4.1 und Kapitel 4.4 beschrieben, aus Architekturzustand und den verwendeten Speicherstellen zusammen. Die vorhandenen Register im Architekturzustand sind dabei von der jeweiligen Architektur gegeben und somit schon vor der Generierung der Programmnetzliste bekannt. Der verwendete Speicher hängt hingegen vom betrachteten Programm ab. Somit müsste der Programmzustand während der Generierung der Programmnetzliste ständig erweitert werden.

Da der Programmzustand die Eingabe jeder Instruktionszelle ist, müssten die Zellen immer entsprechend mit dem – nach und nach erweiterten – Programmzustand umgehen können. Dieses Verhalten zu implementieren ist aufwendig. Die “einfache Lösung” wäre es, den vollständigen Speicher, den das Programm vielleicht belegen könnte, von Anfang an mit zu modellieren. Wenn der Bereich nicht genau bekannt ist, muss der ganze Speicher der verwendeten Hardware modelliert werden. Das würde allerdings eine riesige zusätzliche Logik erfordern. Beim Generieren der SAT-Instanz können dank Optimierungen alle unnötigen Speicherstellen in jedem Programmzustand wegoptimiert werden, aber diese Optimierung kostet beim Generieren der SAT-Instanz viel Zeit.

Der im FCK-Tool implementierte Ansatz ist eine Art getrennte Modellierung des Architekturzustands und des Speichers. Der Architekturzustand wird durch jede Instruktionszelle geleitet, während der Speicher separat modelliert wird. Das ermöglicht es nur Speicherstellen in die Programmnetzliste aufzunehmen, die tatsächlich verwendet werden, ohne dass die Instruktionszellen modifiziert werden müssen. Die Lade- und Speicherinstruktionszellen verfügen über zusätzliche Ein- und Ausgänge, durch die sie die Kommunikation mit dem Speicher durchführen. Diese Ein- und Ausgänge werden dann mittels einer Logik für die Adresszuordnung (“Mapping”) verbunden. Das Mapping gibt vor, welche Schreibinstruktionen in der Programmnetzliste berücksichtigt werden müssen. Die zusätzliche Logik muss entsprechend diesem Mapping für jede Instruktion generiert werden, die lesend auf den Speicher zugreift. Die Schreiboperationen liefern neben den Werten implizit – abhängig von ihrer Position in der Programmnetzliste – auch die Sequenz, in der geschrieben wird. Da der Execution-Graph und somit auch die Programmnetzliste

eine Halbordnung bilden, in der immer nur ein Pfad auf einmal aktiv sein kann, können alle tatsächlichen Sequenzen entsprechend der Ordnung modelliert werden. Die einzelnen aktiven Teil-Sequenzen sind dank dem Active-Signal innerhalb der einen Sequenz eindeutig zu identifizieren. Dann sind in der einen Speicher-Modellierungs-Sequenz nicht alle Schreibelemente aktiv, sondern ausschließlich diejenigen, die tatsächlich auf dem aktiven Pfad liegen. Die Reihenfolge der parallel liegenden Instruktionen spielt keine Rolle, da sie auf getrennten Pfaden liegen. Ein Beispiel, wie diese Logik generiert wird, ist im Appendix B beschrieben und in Abbildung B.1 graphisch dargestellt.

Ein weiterer Aspekt ist die Modellierung des Stack-Pointers. Dieser hat zwar einen festen Startpunkt, aber der Stack kann von vielen Programmen auf dem System verwendet werden. Wenn in der Analyse nur ein Programm betrachtet wird, ist der Analyse nicht bekannt, in wie weit der Stack schon verwendet und der Stackpointer entsprechend modifiziert wurde. Somit kann der Stack beliebig im Speicher liegen, wodurch theoretisch der gesamte Speicher wieder modelliert werden müsste. Da der Stackpointer aber nur relativ zu seinem Startpunkt verwendet wird, ist es als Ansatz zunächst möglich, ihn als konstant anzunehmen. Danach ist es möglich – ähnlich, wie es in der Arbeit [Feh12] vorgestellt wurde – die Zugriffe auf dem Stack so zu prüfen, dass diese nichts von dem darunter liegenden Stack gelöscht bzw. modifiziert haben. Sollte mittels Stack eine Kommunikation zwischen den Programmen generiert werden, so muss das gesondert betrachtet werden. Es ist dann eine Anforderung an den Verifikateur zu entscheiden, wie der Stack abgeprüft werden soll.

Wenn die Programme von C/C++ in Maschinencode übersetzt werden, erzeugt der Compiler zusätzlich zum Maschinencode eine Tabelle mit der Zuordnung von Assembler-Variablen zu den C-Variablen/Pointern. Diese Datei enthält auch die Labels aus den Programmfragmenten und die Adresse, wo diese im Speicherbereich des Programms liegen. So kann der Maschinencode dem C-Programm zugeordnet werden, und so können die Eigenschaften mit einem Bezug zu den C/C++-Variablen geschrieben werden, was für den Softwareentwickler hilfreich ist.

Kapitel 7

Experimentelle Ergebnisse

Es werden nun Ergebnisse von Experimenten mit verschiedenen Beispielen hardwarenaher Software vorgestellt. Dabei hat jede Software eine unterschiedliche Aufgabe, wodurch die Verifikationsziele unterschiedlich sind und somit ebenfalls die zu verifizierenden Eigenschaften.

Die erste Software ist ein einfacher Seriell-zu-Parallel-Konverter, der Werte seriell von einem I/O-Device einliest und das empfangene Wort im Speicher ablegt. Das Programm generiert viele Verzweigungen und damit viele Ausführungspfade. In diesem Experiment soll untersucht werden, wie wirksam die Maßnahmen zum Zusammenführen von Pfaden (Merging) im Hinblick auf die Kompaktheit des generierten Modells sind.

Die zweite Software ist ein Teil eines Selbsttests, der die Funktionalität der unterliegenden Hardware testen soll. Dabei wurde auf der Programmnetzliste verifiziert, ob das Programm erwartete Fehler findet oder potentielle Fehler übersieht.

Das dritte Beispiel ist eine industrielle Implementierung eines LIN-Master-Knotens. Mit Hilfe der Programmnetzliste wurde das Verhalten der Software gegenüber der LIN-Spezifikation verifiziert.

7.1 Seriell-zu-Parallel-Konverter

Der Seriell-zu-Parallel-Konverter liest Bits seriell von einer E/A-Peripherie ein und speichert die Werte parallel als Bytes im Speicher. Das Einlesen geschieht über zwei Eingabe-Signale. Ein Signal gibt an, ob gerade ein Bit gesendet wird. Das zweite Signal gibt den eigentlich gesendeten Wert an. Da seriell über einen Bus gesendet wird, wird das Signal dreimal auf dem Bus abgetastet und anschließend werden die eingelesenen Werte gemittelt.

Wird dreimal eine 1 gelesen, wird eine 1 gespeichert, ist es zweimal ein 1 und einmal eine 0, dann wird eine 1 gespeichert. Entsprechend bei zweimal 0 und einmal 1 wird eine 0 gespeichert und drei 0 werden zur 0 gemittelt und gespeichert. Das erste Signal, das angibt, ob überhaupt ein gültiger Wert empfangen wird, wird mittels Polling abgefragt. Dabei gilt die zusätzliche Annahme, dass spätestens nach dem fünften Leseversuch ein

gültiger Wert anliegt. Dieses Programm kann für beliebige Vielfache von 8 Bits konfiguriert werden. In dem Experiment wurden die Programmnetzlisten für 8, 16, 32 und 64 Bit gebaut und entsprechend verifiziert. Anhand der Ergebnisse aus dem Experiment in Tabelle 7.1 ist leicht zu erkennen, dass bei einer Verdoppelung der zu empfangenden Bitzahl auch eine Verdoppelung der Knoten in der Programmnetzliste einhergeht. Bei der Anzahl der Verifikationsaufrufe während der Generierung der Programmnetzliste ist es entsprechend. Der verwendete Speicher und die Laufzeit wachsen dagegen jeweils um den Faktor 4. Insgesamt wachsen der zeitliche Aufwand und der Speicherbedarf quadratisch. Dabei hängt der Aufwand von der Anzahl an Aufrufen des Verifikationsbeweises ab.

Die Generierung der Programmnetzliste wurde auf einem Rechner mit Intel(R) Xeon(R) CPU E5-2650 mit 2GHz und 32GB Speicher durchgeführt. Als Backend wurde MiniSAT verwendet.

Tabelle 7.1: Seriell zu Parallel Konverter

Programm	**# Instruktionen**		**CPU**	**Mem.**	**#Aufrufe**
Version	**CFG**	**PN**	**(min)**	**(MB)**	**SAT**
8 bit	98	1675	1m38	360	130
16 bit	98	3335	6m19	1600	258
32 bit	98	6655	25m59	5200	514
64 bit	98	13295	109m45	24000	1026

Zur Verifikation des Protokollverhaltens des Programms wurde eine Property geschrieben, die überprüft, ob für beliebig empfangenen Werte der korrekte Wert im Speicher abgelegt wird. Die Eigenschaft wurde für die jeweilige Bitbreite (8,16 und 32-Bit) separat mit OneSpin verifiziert, wobei die Laufzeiten jeweils unter einer Minute betrugen.

7.2 Software-basierter Selbsttest

Ein Selbsttest der CPU dient zur Überprüfung des korrekten Verhaltens. Dabei wird versucht Fehler während der Laufzeit des Prozessors zu erkennen, die zum Beispiel durch Alterung, Überhitzung oder sonstige äußerliche Einwirkungen entstehen. Dafür werden arithmetische und logische Instruktionen beim Selbsttest ausgeführt und alle Register mitverwendet. Die berechneten Ergebnisse müssen von einer zweiten Quelle gegengeprüft werden. Die zweite Quelle kann eine andere CPU sein, die ebenfalls diese Berechnungen durchgeführt hat bzw. die Werte mit einer Wertetabelle abgleicht, in der eine “Musterlösung” gespeichert ist.

Die Qualität des Selbsttests ist abhängig von der Zahl der möglichen Fehler, die das Programm finden kann.

Der getestete Prozessor ist der Peripheral-Control-Processor (PCP), ein Coprozessor im Tricore [Inf08], der hauptsächlich für die Unterstützung der Kommunikation zwischen

dem Tricore selbst und der Umgebung sorgt. Dieser Prozessor besitzt 8 Register, die auch für die Konfiguration des Prozessors verwendet werden.

Die Generierung der Programmnetzliste ist in diesem Beispiel sehr einfach, da das Programm überwiegend sequentiell arbeitet und kaum Verzweigungen besitzt. Es berechnet die gewünschten Werte und speichert sie im Speicher, damit ein anderer Prozessor die Daten auswerten kann.

Die Zeit zur Generierung der zugehörigen Programmnetzliste beträgt 34,75 Sekunden. Als Verifikations-Engine zum Überprüfen der Verzweigungen wurde der Property Checker OneSpin 360 [One] verwendet.

Mit Hilfe dieser Programmnetzliste kann mittels Property Checking überprüft werden, ob die Selbsttest-Software mögliche Fehler nicht erkennt. Es gibt in der Fehlersimulation diverse Fehlermodelle, die als Fehlermuster verwendet werden können. Klassische, einfache Fehlermodelle sind Stuck-at-Fehler, in der ein Registerbit entweder fest den Wert 1 oder 0 annimmt. Wird bei einem Register mit einem Stuck-at-0 eine 1 hineingeschrieben, wird beim Auslesen eine 0 gelesen und umgekehrt. Einen Überblick über weitere und komplexere Fehlermodelle wird in [ZAV04] gegeben.

In diesem Experiment wurde der Selbsttest auf Stuck-at-Fehler untersucht. Dafür wurde eine Eigenschaft geschrieben, die überprüft, ob einzelne Bits in den Registern während des ganzen Selbsttests nie ihren Wert wechseln müssen. Die Verifikation wurde mit OneSpin MV360 durchgeführt.

Um Stuck-at-Fehler zu erkennen, muss in ein solches betroffenes Register mindestens ein Wert hineingeschrieben werden, der vom entsprechenden Fehler abweicht. Hat ein Register einen Stuck-at-0-Fehler, so kann dieser nur durch das Schreiben des Wertes 1 erfolgreich erkannt werden. Wird dagegen nur der Wert 0 hineingeschrieben, dann wird der Fehler nicht als solcher erkannt. Das Ziel der zu prüfenden Eigenschaften war somit, Registerbits zu erkennen, die während der Ausführung des Selbsttests konstant nur einen Wert annahmen, selbst wenn diese mehrfach geschrieben wurden.

Listing 7.1: Eigenschaft zum Erkennen von Registerbits mit konstanten Wert

```
1  property never_modified_bit is
   dependencies: constraint_list;
   -- Anzahl Register:
   for: i=0..7,
5  -- Die einzlenen Bits (31)
         j=0..31;

   assume:
   -- Startannahmen
10 at t: s.active = true;
   at t: s.cycles = 0;

   prove:
   -- Teste ob ein Bit waehrend der Laufzeit einen konstanten Wert hat.
15 at t: not(s.regfile(i)(j) = s0.regfile(i)(j) and
    s.regfile(i)(j) = s1.regfile(i)(j) and
    s.regfile(i)(j) = s2.regfile(i)(j) and
```

```
    ...
    s.regfile(i)(j) = s143.regfile(i)(j) and
20  s.regfile(i)(j) = s144.regfile(i)(j));

  end property;
```

Listing 7.1 zeigt die zu prüfende Eigenschaft in verkürzter Version. Die vollständige Eigenschaft findet sich im Anhang C.1. In der Eigenschaft werden einzelne Bit-Positionen im Registersatz abgeprüft. Dabei wird der Wert des jeweiligen Bits mit der gleichen Position nach dem Ausführen jeder Instruktion verglichen.

Ist dieses Bit immer konstant 0, wird vom Selbsttest ausschließlich der Wert 0 in dieses Bit gespeichert. Wenn das Bit immer konstant 1 ist, wird bei jeder Schreiboperation auf dem Bit der Wert 1 in das entsprechende Bit geschrieben. Ist es mit der gegeben Eigenschaft möglich zu prüfen, dass das Bit konstant den Wert 1 hat und es auch möglich ist das Bit auf Konstant den Wert 0 zu prüfen, dann wird es vom Selbsttest überhaupt nicht modifiziert. Im Fall, das der Wert weder auf Konstant 1 noch auf Konstant 0 geprüft werden kann, wird das Bit im Laufe der Ausführung des Selbsttest sowohl den Wert 0 als auch den Wert 1 annehmen.

Um genau diese beschriebenen Fälle zu finden, wird das invertierte Verhalten in der Eigenschaft abgeprüft, also die Annahme, dass diese Bedingung nie erfüllt ist. Findet der SAT-Solver eine Belegung, in der die Eigenschaft nicht erfüllt wird, wird ein Gegenbeispiel generiert, in dem genau ein solches Bit gefunden wurde, das während der gesamten Ausführung des Selbsttests konstant ist. Um nun alle Gegenbeispiele zu finden, wird die Eigenschaft iterativ neu verifiziert und dabei die schon gefundenen Gegenbeispiele durch das Hinzufügen von Annahmen im "*assume:*"-Teil ausgeschlossen. Die Eigenschaft wird solange aufs Neue geprüft, bis die formale Verifikation der Eigenschaft auf dem gegebenen Design hält.

Zu beachten ist, dass die Ergebnisse in Bezug auf die Hardware interpretiert werden müssen. Register 7 dient dem PCP als Konfigurationsregister. Deswegen können in diesem Bereich einige Bits gar nicht getestet werden. Bit 16-31 sind für Systemfunktionen reserviert. Die Bits 0-3 beinhalten Status Bits wie carry und overflow. Die weiteren Bits dienen zur Konfiguration der Priorität der Softwareprogramme und sind somit während der Ausführung des Programms ebenfalls konstant.

Die Ergebnisse in Tabelle 7.2 beruhen auf einer einzeln geprüften Eigenschaft, die für jeden potentiellen Fehler iterativ wieder neu aufgerufen wurde und hatte für jeden einzelnen Laufs eine Laufzeit von 0,03 Sekunden und benötigte 294 MB an Speicher.

Mit diesem Verfahren konnten mehrere Bits ermittelt werden, in denen ein potentieller Stuck-at-Fehler übersehen werden könnte. Die erste Spalte in Tabelle 7.2 bezeichnet die geprüften Register. Im Anschluss werden die einzelnen Bits in Abhängigkeit ihren möglichen übersehenen Fehlern gezeigt. Die Bits, die beim Überprüfen auf beide möglichen Belegungen 0 und 1 getestet werden, befinden sich in der Spalte "Fehlerfrei". Der Selbsttest kann bei diesen Bits sowohl Stuck-at-0-Fehler als auch Stuck-at-1-Fehler erkennen. Somit ermittelt der Selbsttest die korrekten Werte nur in dem Fall, dass diese Bits fehlerfrei sind. Sollte ein Bit nur mit dem Wert 0 beschrieben werden, könnte ein Stuck-at-

Tabelle 7.2: Selbsttest: Übersicht über getestete Bits

Register-Nummer	Fehlerfrei	Betroffene Bits Stuck-at-0	Stuck-at-1	Stuck-at-0/1
Reg 0	0-31	-	-	-
Reg 1	0-8, 10-12, 14-16, 18-20, 22-24, 26-28, 30,31	9,13,17,21,25,29	-	-
Reg 2	0-31	-	-	-
Reg 3	0-17,19-21,23-30	18,22,31	-	-
Reg 4	0-31	-	-	-
Reg 5	0-9,13-25,29,31, 32	10,12,26,28, 30	11,27	-
Reg 6	0-15,17-19, 21-25, 27-29, 31	16,20,26,30	-	-
Reg 7	0-3	4-15	-	16-31

0-Fehler übersehen werden. Das ist in der entsprechenden Spalte angegeben. Äquivalent dazu sind die Stuck-at-1 Fehler. In der letzten Spalte stehen die Bits in den Registern, die nicht bzw. nicht korrekt geprüft werden.

Wie zu erwarten, werden die Bits in Register 7 von Bit 16 bis Bit 31 nicht überprüft und stehen deswegen in der Spalte “Stuck-at-0/1”. Das sind allerdings ausschließlich Bits, die mit dem Selbsttest bedingt durch die Architektur der CPU nicht getestet werden können.

Auch die Bits 0-3 in Register 7 werden zwar von dem Selbsttest mit verschiedenen Werten beschrieben, doch werden sie nicht wirklich aktiv ausgelesen, da sie ausschließlich Status Bits sind. Somit könnten potentiell dennoch Fehler in diesen Bits vorkommen.

Die Register 1, 3 und 5 besitzen hingegen Registerbits, die vom Selbsttest nicht optimal überprüft werden, wodurch ein Fehler übersehen werden kann. Mit diesem Wissen könnte der Selbsttest optimiert werden, indem er andere Rechenoperationen durchführt, die gezielt diese Stellen auch mit der fehlenden Wertebelegung abprüfen. Infolge dessen müsste die Software an diesen Bit-Stellen mindestens einmal den Wert 1 zuweisen, um die potentiellen Stuck-at-0 Fehler zu erkennen.

7.3 Sensorplattform

Die Sensorplattform ist ein Beispiel für die automatische Kombination von Programmnetzlisten aus verschiedenen Interrupts und dem Hauptprogramm zur einer gemeinsamen Programmnetzliste. Die Sensorplattform implementiert einen Slave, der an eine serielle Kommunikationsschnittstelle (SCI) angeschlossen ist. Sie besitzt mehrere Sensoren und funktioniert interruptgetrieben. Die Plattform empfängt zwei Analogwerte über Analog/Digital-Konverter (ADC) und führt einfache arithmetische Operationen auf diesen Werten durch. Die empfangenen Werte und das Ergebnis der Rechenoperation werden

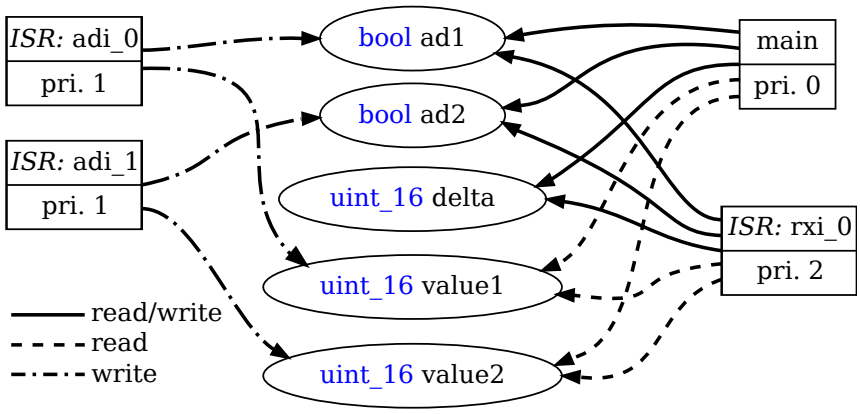

Abbildung 7.1: Sensorplattform – Kommunikationsstruktur

nach einer Aufforderung (engl. *Request*) vom Master über die SCI versendet.

Zwei ISRs kümmern sich um die Interrupt-Request der ADCs (adi_0 und adi_1) und der dritte ISR (rxi_0) um den Ablauf der Kommunikation über den SCI. Die Hauptroutine übernimmt die Werte, die die ADC-ISRs geschrieben haben und führt eine Berechnung auf den Werten aus, um anschließend das Ergebnis in den gemeinsamen Speicher zurückzuschreiben, der vom ISR rxi_0 gelesen werden kann.

Ziel dieses Experiments ist es, die automatische Erstellung der kombinierten Programmnetzliste, mit den verschachtelten Interrupts und der Hauptroutine zu demonstrieren. Abbildung 7.1 beschreibt die Kommunikation zwischen den Interrupts und dem Hauptprogramm über den gemeinsamen Speicher. Die eckigen Boxen stehen dabei für die jeweilige Softwarekomponente und beinhalten eine Priorität (“‘pri.”). Routinen mit höherer Priorität besitzen einen Interrupt, der Routinen mit niedrigerer Priorität unterbrechen kann. Die Lese- und Schreibzugriffe von den Routinen auf die Variablen im Speicher (Ovale) sind entsprechend der Pfeile entweder nur lesend, nur schreibend oder sowohl lesend als auch schreibend.

Tabelle 7.3: Sensorplattform: Programmnetzlistengenerierung

Programm	**# Instruktionen**		**Komm.-**	**# Instanzen**	
Komponente	**CFG**	**PN**	**Punkte**	**Optimiert**	**Naiv**
main	100	179	27	1	1
ISR adi_0	12	12	2	21	71
ISR adi_1	12	12	2	21	71
ISR rxi_0	64	68	9	105	1775

Für jede einzelne Komponente wird die Programmnetzliste generiert. Anschließend werden die Kommunikationspunkte berechnet. Basierend auf den berechneten Punkten und den einzelnen PNs wird die Gesamtprogrammnetzliste erstellt. Die Rechenzeit zur Generierung der Gesamtprogrammnetzliste beträgt 4 Sekunden.

Die Tabelle 7.3 zeigt, wie viele Instanzen der jeweiligen Programmnetzliste benötigt werden, um das Gesamtmodell zu erstellen. Die Spalten für die Anzahl der Instruktionen beziehen sich auf die Anzahl der Instruktionen, die in einer einzelnen Programmnetzliste für die jeweilige Softwarekomponente benötigt werden.

Die Anzahl an Assemblerinstruktionen des Gesamtprogramms ist 161. Die Summe der einzelnen CFGs von allen Komponenten enthält mehr Knoten als Instruktionen, wie in Kapitel 4.2 beschrieben, da noch zusätzliche Knoten von Daten im .text Segment generiert werden.

Während für das Hauptprogramm nur eine Instanz benötigt wird, werden – entsprechend der Beschreibung in Kapitel 4.5 – für jeden Kommunikationspunkt bei den Interrupts jeweils eigene Instanzen generiert. Da die ISR rxi_0 neben dem Hauptprogramm "main" auch noch die Interrupts adi_0 und adi_1 unterbrechen kann, gibt es von ihr die meisten Instanzen. Im naiven Ansatz (Spalte "Naiv") wird die Anzahl an Instanzen angegeben, wenn nicht die in Kapitel 4.5.2 vorgestellten Optimierungen genutzt werden. Mittels der Optimierung werden wesentlich weniger Instanzen benötigt. Anstelle 1,4 Millionen Instruktionen in der Gesamtnetzliste werden nur noch knapp über 86000 Instruktionen benötigt.

7.4 LIN Master

Die Abkürzung LIN steht für *Local Interconnect Network* und ist ein Bit-serieller Bus, der im Umfeld der Automobilindustrie entwickelt wurde, und der als Subbus für den CAN-Bus verwendet werden kann. Der LIN-Bus ist ein Single-Master- und Multi-Slave-Bus. Der Master setzt sich laut Spezifikation aus einer Slave-Komponente und einer Master-Komponente zusammen. Die Master-Komponente initiiert jede Transaktion auf dem LIN-Bus mittels eines Headers, der als Nachrichtenkopf dient. Die eigentlichen Daten in der LIN-Bus-Transaktion werden mit Hilfe der Slave-Komponente gesendet. Zusammen ergeben sie das Verhalten des vollständigen Masterknotens. Ein solcher Knoten kann mit Hilfe eines kleinen Prozessors und einer UART vollständig in Software implementiert werden. Die Transaktionen werden *Message-Frames* genannt, und deren Kennung gehört – wie beim CAN-Bus – zur Nachricht und ist nicht an einen bestimmten Knoten gekoppelt. Der Frame setzt sich aus einzelnen Datenbytes zusammen wie in Abbildung 7.2. Zuerst wird der Header übertragen, der ein dominantes Break-Signal über den Bus sendet, um eventuell noch laufende Kommunikationen zu unterbrechen und die an den Bus angeschlossenen Knoten auf eine neue Nachricht hinzuweisen. Im Anschluss folgt eine Synchronisationsphase, die alle Knoten miteinander synchronisiert, da jeder Knoten über seinen eigenen Takt verfügt. Das letzte Byte des Headers enthält die Nachrichten-ID inklusive zwei zusätzlicher Parity-Bits um Verfälschungen der ID durch Signalstörungen zu erkennen. Im Anschluss an den Header erfolgt die Antwort eines Slaves mit maximal 8 Datenbytes und einer Checksumme. Abbildung 7.2 beschreibt einen vollständigen Frame auf dem LIN-Bus.

Die hier verwendete Software ist von Infineon Technologies AG und besteht aus 1309

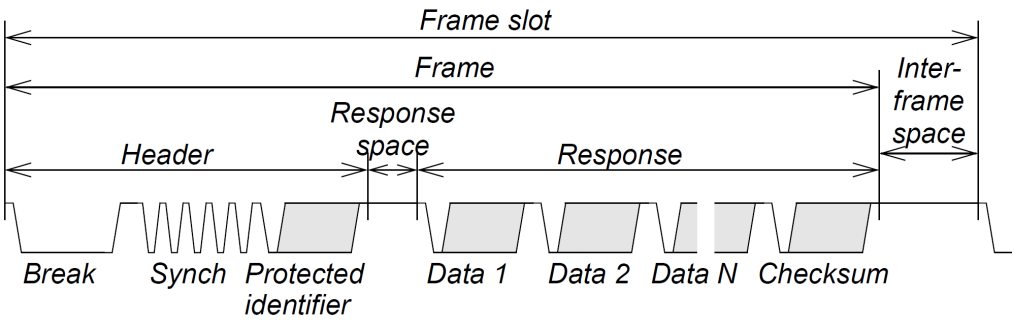

Abbildung 7.2: Aufbau des LIN Frames [LIN03]

Zeilen C-Code und unterstützt den LIN1.3 Standard [LIN02]. Dabei setzt sich der Master aus drei Teilen zusammen. Der erste Teil ist die Initialisierung des Knotens. Der zweite Teil ist der Frame-Scheduler, der entscheidet, welcher Frame als nächster gesendet wird, und der diesen ebenfalls initiiert. Der letzte Teil des Programms ist für das Senden oder Empfangen der Daten zuständig. Dabei ist die Implementierung Interrupt-basiert und wird jedes Mal wieder aufgerufen, nachdem die UART erfolgreich ein Datenbyte gesendet oder empfangen hat.

Die Message-IDs, die für die Kommunikation verwendet werden können, sind feste Parameter, die beim Kompilieren des Programms schon bekannt sein müssen, da die Länge der Nachrichten laut LIN 1.3-Spezifikation ausschließlich von der Message-ID abhängt. Sind nur Message-IDs gewählt, die Frames mit nur zwei Datenbytes versenden, werden im kompilierten Code die Speicherbereiche entsprechend klein erstellt, so dass Nachrichten mit vier oder acht Datenbytes nicht abgearbeitet werden können. Daher sind bei der Konfiguration IDs gewählt worden, die Nachrichten mit acht Datenbytes erfordern.

Zur Generierung des Modells werde die Programmnetzlisten der drei Teilkomponenten der Software separat generiert. Der LIN-Scheduler selbst ist als Endlosschleife implementiert und wird nach der in Kapitel 5.4.1 vorgestellten Methodik nur einmal ausgerollt und dann nach Bedarf mehrfach instanziiert.

Die Laufzeiten zur Generierung sind in Tabelle 7.4 angegeben. Dabei entspricht die Anzahl der Instanzen von Instruktionen in der erzeugten Programmnetzliste ebenfalls der Anzahl der Knoten des Execution-Graphen. Die ausgerollten Programmnetzlisten sind nicht wesentlich größer als das ursprüngliche Programm als CFG dargestellt. Der Interrupt als größte Programmnetzliste verfügt über 1138 Instanzen von Instruktionen. Entsprechend kurz sind auch die Laufzeiten der Generierung.

Der erste Teil, zum Initialisieren der Software, wird nur einmal ausgeführt und danach springt die LIN-Software zum Scheduler in die Endlosschleife. Da nach der Initialisierung auch entschieden wird, welcher LIN-Frame zuerst gesendet wird, würde bei der gemeinsamen Ausführung vom Startpunkt der Software immer derselbe LIN-Frame ausgeführt. Deshalb wird die Initialisierung unabhängig vom Rest betrachtet, und im Anschluss wird die aufgebrochene Endlosschleife im Interval-Property-Checking ohne die Initialisierung

Tabelle 7.4: LIN Master Knoten: Modellgenerierung

Programm Teil	**# Instruktionen CFG**	**PN**	**CPU (s)**	**Mem. (MB)**	**# Aufrufe SAT**
LIN-Init	225	385	1.32	36	22
LIN-Scheduler	85	84	0.13	27	4
LIN-ISR	790	1138	11.00	102	38

verifiziert.

Bei der Initialisierung werden alle Speicherstellen, die geschrieben werden, verifiziert und daraus eine Liste des gültigen Programmzustands erstellt, die beim ersten Eintritt in den Scheduler gelten.

Die Betrachtung der Schleife ohne einen definierten Programmzustand zu Beginn bedeutet, dass der Programmzustand eine Überapproximation des tatsächlichen Programmzustands ist. Dadurch können sogenannte “false negatives” auftreten (vgl. Abbildung 2.8).

Mit Hilfe des Wissens, das aus der Verifikation der Initialisierung gewonnen werden kann, ist es möglich die “false negatives” zu erkennen und aus der Eigenschaft auszunehmen.

Entsprechend der Vorgehensweise in Kapitel 4.5.1 zum Erstellen der kombinierten Netzliste werden die entsprechenden Speicherstellen identifiziert, die zur Kommunikation zwischen Hauptprogramm und Interrupt verwendet werden.

Dabei wurden zwei wesentliche Punkte identifiziert: zum einen zwei Statussignale, die den Status des LIN-Knotens beschreiben und zum anderen die Speicherstelle, an der die zu versendenden Daten gespeichert sind bzw. die Daten, die empfangen werden sollen.

Zusätzlich wird mittels formaler Verifikation auch getestet, ob der Interrupt transparent ist.

Im Anschluss müssen die Instanzen des ISRs nur an die entsprechenden Stellen des LIN-Scheduler, das Hauptprogramm, eingefügt werden. Da das Hauptprogramm nur über 83 Instruktionen verfügt und der UART den IRQ erzeugt, ist es technisch nicht möglich, dass die UART in der Ausführungszeit von 83 Instruktionen zwei Interrupts erzeugt. Daher ist pro Durchlauf durch die Schleife des Hauptprogramms eine Instanz des Interrupts ausreichend. Dennoch muss die Instanz für alle Pfade erreichbar sein. Das so erzeugte und kombinierte Element ist eine Instanz des Hauptprogramms, kombiniert mit einer Instanz des Interrupts. Diese Programmnetzliste reicht jedoch für die vollständige Verifikation eines LIN-Frames nicht aus.

Bei der Betrachtung eines LIN-Frames und wie dieser auf die Software abgebildet wird, ergeben sich je nachdem – ob gesendet oder empfangen wird – zwei unterschiedliche Anzahlen von ISR-Aufrufen. In dem Fall, dass der Masterknoten eine Nachricht empfängt, initialisiert er zunächst einmal einen neuen Frame durch das Senden des Headers. Nachdem die UART das Datenbyte mit der PID über den LIN-Bus versendet hat, wird ein Interrupt erzeugt, der genau diese versendete PID wieder ausliest, sofern das Senden

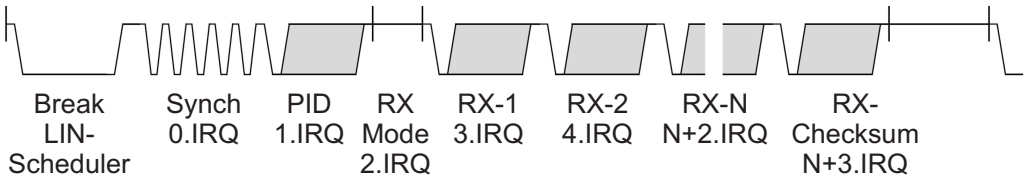

Abbildung 7.3: Programmaufrufe für Ablauf eines LIN-RX-Frames

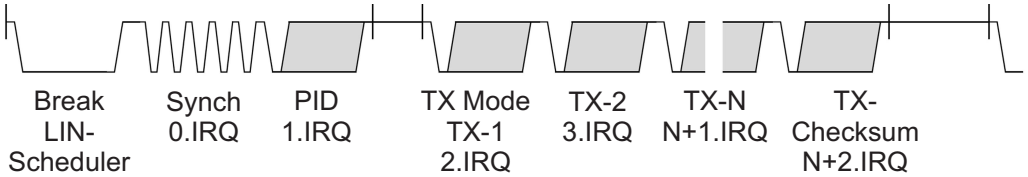

Abbildung 7.4: Programmaufrufe für Ablauf eines LIN-TX-Frames

fehlerlos war. Anhand dieser empfangenen PID entscheidet sich, ob der Slave-Task des Programms Daten sendet oder empfängt.

Wenn die PID dem Empfang einer Nachricht zugeordnet ist, wird der LIN-Knoten entsprechend konfiguriert und wartet auf den nächsten Interrupt mit dem ersten empfangenen Datenbyte. Anschließend wird beim Empfangen eines jeden Datenbytes die ISR von dem Interruptrequest der UART aufs Neue angestoßen. Die Abfolge der Interrupt-Aufrufe für das Empfangen von Daten zeigt Abbildung 7.3.

Wird vom Masterknoten selbst eine Nachricht verschickt, wird ein Interruptaufruf weniger benötigt, da bei der Entscheidung, dass der Masterknoten als Sender fungieren soll, das erste Datenbyte direkt an die UART übertragen wird. Abbildung 7.4 zeigt die Aufrufreihenfolge für einen entsprechenden Frame.

Für einen Datenframe mit 8 Datenbytes, den der Master empfängt, werden somit mindestens $8 + 4 = 12$ Instanzen des Interrupts benötigt. Da diese aber nicht unmittelbar aufgerufen werden, sondern stattdessen beliebig viele Schleifendurchläufe des Schedulers passieren können, müsste das Model um ein Vielfaches der 12 Instanzen ausgerollt werden.

Bei genauerer Betrachtung des Hauptprogramms ist zu erkennen, dass der Scheduler zwar einen neuen Kommunikationszyklus startet, aber von da an wartet, bis der LIN-Knoten wieder in den IDLE-Zustand zurückwechselt und somit bereit ist für eine neue Nachricht. In dieser Zeit wartet das Hauptprogramm durch die Endlosschleife aktiv auf diesen entsprechenden Zustandswechsel. Entsprechend dem Vorgehen in Kapitel 5.4.2 kann dieses aktive Warten mittels eines einfachen Satzes an Eigenschaften formal verifiziert werden, und so lässt sich zeigen, dass diese Schleife in dieser Zeit nichts tut.

Listing 7.2: Eigenschaft zum Erkennen von Registerbits mit konstantem Wert

```
1 property verification_from_waiting_to_waiting is
  dependencies: constraint_list;
  assume:
  -- Default Assumtion for the EXG-Graph
```

```
5 at t: s.active = true;
  -- LIN-Node is not idle
  at t: Node_is_not_idle_state_at_loop_start;

  prove:
10 at t: Node_is_not_at_idle_state;
  at t: z.active = true;

  end property;
```

Die dafür beschriebene Eigenschaft wird in Listing 7.2 gezeigt und überprüft, dass die entscheidenden Variablen des Programmstatus unverändert bleiben. Die Eigenschaftsüberprüfung hat 149 MB Speicher benötigt und wurde in weniger als 0,01 Sekunden nachgewiesen. Die Eigenschaft besitzt als Annahme, dass der LIN-Knoten zu Beginn der Eigenschaft nicht im Idle-State ist und auch nach der Ausführung der Schleife nicht zurück in den Idle-State wechselt, ungeachtet unter welchen zusätzlichen Annahmen.

Der Beweis zeigt, dass der Loop aus dem Hauptprogramm diesen Zustand nicht verlassen wird, solange der Knoten nicht in den Idle-Mode zurückspringt. Diese Erkenntnis kann dazu verwendet werden, um im Gesamtmodell die Anzahl der Instanzen zwischen zwei Interrupts auf eine einzelne Instanz des Hauptprogramms zu reduzieren.

Das Gesamtmodel modelliert dann das Gesamtverhalten durch 12 notwendige Instanzen für den Interrupt und den entsprechend zugehörigen Instanzen des Hauptprogramms und besteht aus insgesamt 24001 Instruktionen und 62592 binären Inputsignalen. Die einzelnen Architektur-Anfangs-Zustände der Instanzen des Interrupts sind – wie in Kapitel 4.5.1 beschrieben – nicht miteinander verbunden und verfügen über offene Eingänge.

Auf dem so entstandenen Gesamtmodell kann das Gesamtverhalten eines vollständigen LIN-Frames formal verifiziert werden. Die Eigenschaften, die mit diesem Model verifiziert werden sollen, werden direkt aus der LIN-Spezifikation abgeleitet.

Folgende Eigenschaften der LIN-Spezifikation werden dabei überprüft:

- Empfangen eines kompletten LIN-Frames mit fehlerfreier Prüfsumme,
- Verhalten beim Empfang eines Frames mit fehlerhafter Prüfsumme,
- Senden eines Datenframes mit Prüfsumme,
- Verhalten der Software im Falle einer fehlerhaften ID bzw. einer nicht korrekten PID.

Der Aufbau der Programmnetzliste ist somit eventgetrieben, da der Aufruf einer Interruptinstanz dann geschieht, wenn die UART entweder ein Datenbyte empfangen oder versendet hat. Somit sind auch die Eigenschaften eventorientiert und prüfen das Verhalten des Softwaremodells am Ende der Ausführung der Interruptinstanzen.

Beispielhaft wird folgende Eigenschaft genauer besprochen: Ein LIN-Frame mit der Länge von vier Datenbytes wird empfangen, nachdem der Master den entsprechenden Header der Nachricht gesendet hat. Der vollständige Auszug aus der Eigenschaft für vier

Datenbytes wird in Appendix C.2 gezeigt. In Listing 7.3 wird eine verkürzte Variante der Eigenschaft vorgestellt. Sie beschränkt sich auf die wesentlichen Merkmale, die von der LIN-Spezifikation gegeben werden.

Die Eigenschaft besitzt die Werte RX1 bis RX4, die die entsprechenden Datenbytes des empfangenden Frames darstellen und beliebige Werte beinhalten können. RX5 steht für die Prüfsumme des übertragenden Frames. In LIN 1.3 erstreckt sich die Prüfsumme ausschließlich über die Datenbytes.

Im Prove-Teil der Eigenschaft werden vier verschiedene Punkte verifiziert. Der erste prüft die Zustandswechsel beim Lesen der einzelnen Schritte. Dabei wechselt der Zustand nach dem Empfang des Frame-Headers in den Zustand, dass Daten empfangen werden, und bleibt solange in diesem Zustand, bis der LIN-Knoten wieder in den Idle-Modus zurückwechselt.

Zusätzlich wird überprüft, ob der Error-Status keinen Fehler anzeigt, und ob auch die richtige Anzahl zu empfangender Datenbytes erkannt wurde.

Der letzte Teil der Eigenschaft verifiziert das korrekte Speichern aller empfangenen Daten. Dabei wird geprüft, ob die richtigen Werte an die richtigen Speicheradressen geschrieben wurden.

Listing 7.3: Eigenschaft für ein LIN-Frame mit 4 Datenbytes

```
1  property verification_LIN_RX_Frame_4_DB_checksum is
   assume:
   -- RX1 bis RX5 stehen f{\"u}r die Empfangenen Werte,
   -- RX0 beinhaltet die PID der empfangenen Nachricht
5  -- Berechnung der Checksume (Annahme, das empfangene Checksumme
       korrekt ist)
   at t:  RX5 = checksum_4bytes(RX1, RX2, RX3, RX4);

   prove:

10 at t:  -- State of LIN Node
                LIN_Transceiver_Status_after_irq2= receiving_data
            and LIN_Transceiver_Status_after_irq3= receiving_data
            and LIN_Transceiver_Status_after_irq4= receiving_data
            and LIN_Transceiver_Status_after_irq5= receiving_data
15          and LIN_Transceiver_Status_after_irq6= receiving_data
            and LIN_Transceiver_Status_after_irq7= Idle
            -- Error status
            and irq7_memory_0x2060(7 downto 0) = no_error
            -- Correct number of Databytes
20          and irq3_memory_0x2056(7 downto 0) = X``04''
            -- Data Saved at correct position
            and  irq3_memory_0x204e(7 downto 0) = RX1
            and  irq4_memory_0x204f(7 downto 0) = RX2
            and  irq5_memory_0x2050(7 downto 0) = RX3
25          and irq6_memory_0x2051(7 downto 0) = RX4;
   end property;
```

Ähnlich zu der vorgestellten Eigenschaft – für das Empfangen eines vier Datenbyte

langen Frames – wurden alle anderen Eigenschaften erzeugt. Das Verifizieren der Eigenschaften wurde mittels OneSpin auf einem Intel Xeon E5440 System mit 16GB Speicher durchgeführt.

Die Tabelle 7.5 zeigt die Übersicht der Laufzeiten der einzelnen Eigenschaften. In den Eigenschaften werden unterschiedlich lange Datenframes gleichzeitig geprüft. Zusätzlich ist der Inhalt der Datenbytes nicht festgelegt und kann beliebige Werte annehmen. Entsprechend muss die Prüfsumme korrekt berechnet werden. Die ID's hingegen sind vorgegeben, da abhängig von der ID unterschiedlich lange Frames erwartet bzw. versendet werden.

Tabelle 7.5: LIN Master Node Driver: Property Checking

Property	**CPU (s)**	**Mem. (MB)**
RX frame 2or4 data bytes incl. checksum	7	1459
RX frame 4or8 data bytes incl. checksum	17	1641
RX frame 2or4 data bytes wrong checksum	7	1361
RX frame 4or8 data bytes wrong checksum	28	1545
TX frame 2or4 data bytes incl. checksum	6	1206
TX frame 4or8 data bytes incl. checksum	15	1548
Wrong PID or not matching ID	6	1205
Wrong PID or not matching ID (8 Bytes)	14	1566

Die Laufzeiten betragen wenige Sekunden, und der Speicherverbrauch ist sichtbar unter dem Limit für aktuelle Hardware. Die Ergebnisse zeigen, dass das Verhalten der Spezifikation über ganze Frames gezeigt werden kann. Da das ganze System eventgesteuert arbeitet, ist auch die Eigenschaft in diesem Fall am eventbasierten Verhalten orientiert, wodurch das Abprüfen der Eigenschaft gut funktioniert.

Die Reihenfolge für das richtige Schreiben auf dem LIN-Bus wird durch die hier verwendete UART vorgegeben. Diese besagt, dass die Konfiguration zuerst übertragen werden muss und anschließend die Daten gesendet werden. Um zu überprüfen, ob die Reihenfolge des Schreibens bei der UART korrekt ist, genügt es, diese Eigenschaft an einer einzelnen Instanz der jeweiligen Netzliste zu überprüfen.

Kapitel 8

Ergebnisse und Ausblick

In den vorherigen Kapiteln wurde gezeigt, wie die Programmnetzliste entworfen wird, welche Eigenschaften sie besitzt und warum sie eine korrekte Darstellung der Software ist. In diesem Kapitel wird zunächst alles noch einmal zusammengefasst und hervorgehoben, warum die Programmnetzliste für die Verifikation gut geeignet ist. Danach werden noch die durch diese Arbeit motivierten Themen angesprochen und noch überlegt, wozu die Programmnetzliste außerdem genutzt werden kann.

8.1 Zusammenfassung

In dieser Arbeit wurde die Programmnetzliste, ein Modell für die hardwarenahe Softwareverifikation vorgestellt, welche eine explizite Darstellung des Programmablaufs und eine implizite Modellierung des Datenpfads besitzt. Die Betrachtung der Software passiert auf Maschinencode-Level. Die Modellgenerierung besteht aus wenigen und gut automatisierbaren Schritten. Zuerst wird aus einer Software ein Kontrollfluss Graph (CFG) extrahiert. Im Anschluss daran wird die Modellgenerierung in zwei Schritten durchgeführt.

Der erste Schritt ist die Erzeugung des expliziten Programmablaufs, indem der CFG mit der in Kapitel 3 vorgestellten Idee abgerollt wird. Dabei wird ein sogenannter Execution-Graph (EXG) erzeugt, der alle möglichen Ausführungspfade des betrachteten Programms beinhaltet. Um dieses Modell so kompakt wie möglich zu halten, werden unterschiedliche Techniken verwendet – wie das Zusammenführen gemeinsamer Pfade und das Erkennen von "toten" Verzweigungen im Programm, die an der entsprechenden Stelle niemals ausgeführt werden.

Im Anschluss wird im zweiten Schritt der Execution-Graph in die Programmnetzliste (PN) übersetzt. Dabei werden alle Knoten im EXG, die jeweils einer konkreten Instruktion des Programms entsprechen, durch eine entsprechende Instruktionszelle – eine Logikblock der das Prozessor-spezifische Verhalten einer einzelnen Instruktion modelliert – ersetzt. Die Kanten des Graphen entsprechen dabei dem Programmzustand. Der Programmzustand setzt sich aus den Variablen im Speicher wie auch dem Architekturzustand des unterliegenden Prozessors zusammen.

Ergänzt wurde der Programmzustand in der Programmnetzliste um ein sogenanntes Active-Bit, welches es ermöglicht den aktiven Pfad in der Netzliste zu markieren. Das ist notwendig, da die Software immer nur einen Pfad gleichzeitig ausführen kann, aber die PN alle möglichen Pfade beinhaltet. Auf der Programmnetzliste können dann mit Hilfe von Hardware Property Checkern basierend auf BMC oder IPC diverse Eigenschaften bewiesen werden.

Auch zum Überprüfen der Verzweigungen bei der Erstellung EXG kann dieses Verfahren mittels einer unfertigen PN genutzt werden. Dafür werden einfache Eigenschaften, die das Active-Bit prüfen, automatisch erzeugt und mit Hilfe eines SAT-Solvers auf ihre Gültigkeit geprüft.

Zusätzlich wurde die Programmnetzliste um die Fähigkeit zur Interruptmodellierung erweitert (Kap. 4.5). Die Kommunikation zwischen Interrupt und Hauptprogramm geschieht dabei ausschließlich über gemeinsame Speicheradressen. Mit Hilfe eines Transparenzbeweises lässt sich diese Annahme auch beweisen. Die Anzahl an Instanzen eines Interrupts wird dabei auf ein notwendiges Minimum reduziert, um das Modell möglichst kompakt zu halten.

Es wurde ein Tool (Kap. 6) implementiert, das aus einem Maschinencode die Programmnetzliste automatisiert generiert. Mit Hilfe dieses Tools wurde ein Software-basierter Hardware-Selbsttest (Kap.7.2) auf seine Fehleranfälligkeit untersucht und ein LIN-Master (Kap. 7.4) auf Fehlerfreiheit formal verifiziert.

8.2 Ergebnisse

Die Verifikation der funktionalen Korrektheit von Hardware und Software wird schon seit einigen Jahren mit Hilfe von formalen Methoden und Werkzeugen in der akademischen Forschung und ebenfalls in der Industrie erfolgreich eingesetzt. Dabei werden meistens Hardware und Software separat betrachtet, was für viele Fällen ausreichend ist. Bei eingebetteten Systemen, in denen die Software mehr und mehr Aufgaben übernimmt, die früher der Hardware vorbehalten waren, nimmt die Notwendigkeit einer gemeinsamen Betrachtung stetig zu. Deswegen ist es interessant, die Software und die Hardware genau an der Stelle zu beobachten, an der die Kommunikation tatsächlich stattfindet: Das ist aus Software-Sicht auf dem Level des Maschinencodes. Diese Ebene wurde z.B. auch von den Arbeiten [NWSK11], [GKD06] und [HTV+13] verwendet und unterscheiden sich nur in der Art, wie das Modell aussieht und wie es implementiert wurde.

Und genau auf diesem Level setzt das hier in dieser Arbeit vorgestellte Modell an. Doch anstelle den kompletten Prozessor und die Software zu modellieren, wie in [NWSK11] wird eine abstraktere Darstellung gewählt. Die Software wird abstrahiert und auf jedem möglichen Ausführungspfad durch eine entsprechende Sequenz an Hardwareinstruktionen der CPU ersetzt. Die resultierende Programmnetzliste ist eine wesentlich kompaktere Darstellung, als ein über viele Zeitschritte ausgerolltes Modell, das den kompletten Prozessor beinhaltet ([NWSK11]) oder in jedem Ausführungsschritt alle Programmteile vorhält ([HTV+13]). Zusätzlich muss so die Solver-Engine nicht noch selbst errechnen,

welche mögliche Komponente in jedem Zeitschritt aktiv sein kann. Trotz dieser Vereinfachungen entsprechen die Sequenzen in der Programmnetzliste dem realen Verhalten der ausgeführten Software auf dem entsprechenden eingebetteten System. Somit ist eine aussagekräftige formale HW/SW-Co-Verifikation möglich.

Eine weitere Eigenschaft der Programmnetzliste ist, dass sie mit überschaubarem Aufwand an neue Plattformen angepasst werden kann. Es muss lediglich ein Satz von Instruktionszellen für jede zu unterstützende Plattform generiert werden. Zusätzlich ist es auch möglich für dieselbe Plattform unterschiedlich detailreiche Instruktionszellen zu verwenden. So können die Instruktionszellen beispielsweise mit Zeitannotationen versehen werden, um genauere Aussagen über die Ausführungszeiten von Instruktionen und Programmen berechnen zu können.

In Experimenten haben wir gezeigt, dass Eigenschaften auf einer Programmnetzliste mit ca. 24000 Instruktionen in wenigen Sekunden bewiesen werden können und die längste Laufzeit unter 30sec lag. Diese 24000 Instruktionen sind aus ca. 1300 Zeilen C-Code entstanden und die Generierung der Teilprogrammnetzlisten dauerte insgesamt keine 15 s.

Die Experimente zeigen, wie das Modell zur formalen Verifikation von hardwarenaher Software verwendet werden kann, wobei die zu prüfenden Eigenschaften sich ausschließlich auf Hardwareverhalten oder Daten im Speicher beziehen. Dabei können grundsätzlich Fehler gefunden werden, die durch Spezifikationslücken in der Hardware- oder Software-Spezifikation ein fehlerhaftes Verhalten ermöglichen, welches bei einer kompletten separaten Verifikation nicht auffallen würde. Zum Beispiel wird bei dem Selbsttest aus Kapitel 7.2 durch das zusätzliche Wissen um das Verhalten von Hardwarefehlern eine qualitative Aussage mittels der formalen Verifikation ermöglicht. Somit ist eine Verbesserung des Selbsttests überhaupt erst möglich.

Der aktuell limitierende Faktor der Methodik und Generierung der Programmnetzliste ist die Anzahl an Verzweigungen bzw. die Anzahl von Schleifeniterationen. Wie am Beispiel des Seriell-zu-Paralle-Konverters zu sehen, steigt der Speicherbedarf zur Generierung der Programmnetzliste sehr stark an und hat eine starke Korrelation zur Anzahl an Verzweigungen (#SAT Aufrufe) und Schleifentiefe (starker Anstieg von Knoten in der PN im Vergleich zum CFG). In diesem Beispiel wird im 64 bit-Experiment der SAT-Solver 1026 Mal aufgerufen. Und selbst das kleinste Beispiel benötigt bereits 130 SAT-Aufrufe.

Eine weitere Einschränkung der Programmnetzliste besteht zur Zeit in der Darstellbarkeit der Gegenbeispiele, da es als kombinatorisches Modell nicht über eine wirkliche Zeit verfügt, die mit Standard-Simulationstraces angezeigt werden kann. Somit müssen bei der Betrachtung des Gegenbeispiels am Originalmodell die internen Signale wieder den zeitlichen Zuständen zugeordnet werden. Dafür wurde im Lehrstuhl parallel auch ein Werkzeug für die graphische Darstellung des Graphen erzeugt, das zumindest die Pfade und Knotenbelegungen anzeigen kann. Aber auch das Beschreiben einer Eigenschaft muss zur Zeit anhand der Programmnetzliste orientiert geschrieben werden. Für den Softwareverifikateur und Entwickler wäre es wesentlich einfacher, wenn er die Eigenschaften anhand des Originalprogramms oder der Sequenzen von der Hardware beschreiben könnte, ohne sich dabei intensiv mit dem Modell der PN beschäftigen zu müssen. In Kapitel 8.3 wird ein Ansatz zur besseren Darstellung der Gegenbeispiele beschrieben, an dem schon

gearbeitet wird.

8.3 Ausblick

In dieser Arbeit wurde ein neues Grundmodell zur formalen Verifikation hardwarenaher Software eingeführt. Dabei wurden wesentliche Anwendungsbeispiele aufgeführt. Es wurde im Rahmen dieser Arbeit ausschließlich auf das klassische Property Checking eingegangen. Das Equivalence Checking ist eine weitere Anwendung, die in [Vil16] ausführlich beschrieben wird. Inspiriert von diesen beiden Anwendungen sind während der Durchführung der Experimente die Methodik und das Tool kontinuierlich verbessert worden.

8.3.1 Neue Anwendungsszenarien

Ein neues Anwendungsszenario kann entsprechend dem Selbsttest-Beispiel aus Kapitel 7.2 auch die Fehler-Injektion in den Instruktionszellen sein. Die Fehler können eingefügt werden, um die Robustheit von Programmen zu erforschen. Dabei können auch noch andere Fehlermodelle anstelle der in dieser Arbeit vorgestellten Stuck-at-0/1-Fehler verwendet werden. Die am Lehrstuhl entstandene Arbeit [BVSK17] zeigt, wie mit Hilfe der Programmnetzliste "programmredundante" Hardwarefehler gefunden werden können. Hierbei handelt es sich um Fehler in der Prozessorhardware, die für *für eine gegebene Software* nicht beobachtbar sind und sich damit auf das Systemverhalten nicht auswirken, sofern der Prozessor lediglich diese Software ausführt.

Bei der Modellgenerierung werden viele Daten ermittelt, wie z.B. alle Speicheradressen, auf die zugegriffen wird, aber auch, ob bestimmte Pfade überhaupt vorkommen. Diese Informationen können auch für Applikationen außerhalb der klassischen formalen Verifikation dienen. Sie können beispielsweise benutzt werden, um sowohl bei simulativen Techniken oder allgemeiner Fehlersuche die möglichen Fehlerquellen effizienter einzugrenzen, ohne dass eine ausführliche formale Verifikation durchgeführt werden muss. Wenn zum Beispiel bei einem Prototyp ein falscher Wert an eine Stelle im Speicher geschrieben wird, kann mit Hilfe der Programmnetzliste ermittelt werden, welche Instruktion als Ursache in Frage kommt. Dabei sind die betrachteten Werte nicht die Werte, die vom Programmierer erwartet werden, sondern die, die tatsächlich im Programm möglich sind. Im Falle eines Programmierfehlers kann so die entsprechende Stelle direkt erkannt und der Fehler behoben werden.

Eine weitere Überlegung ist, dass dieses Softwaremodell für automatisierte Tests verwendet werden kann, ähnlich zur Dead-Code Erkennung. Dabei können potentiell neben Dead-Code auch ungültige Bedingungen, Schleifenbegrenzungen oder ähnliche Werte ermittelt werden. Es könnte auch untersucht werden, ob die Programmnetzliste für Optimierungen verwendet werden kann, indem z.B. oft genutzte Sequenzen anhand der Programmnetzliste ermittelt werden können. Optimierungen auf oft verwendeten Codes wirken sich am stärksten auf die Laufzeiten oder andere Optimierungskriterien wie viel-

leicht auch Energieverbrauch aus. Ein weiteres Optimierungspotential kann bei bedingten Verzweigungen als ein If-Then-Else-Konstrukt ermittelt werden. So sind bei gepipelineten Prozessoren das Springen zu einer neuen Adresse mit dem Anhalten und Leeren der Pipeline verbunden, was je nach Pipelinetiefe einige Zeit in Anspruch nimmt. Da in der Programmnetzliste alle möglichen Pfade ermittelt wurden, kann nun bestimmt werden, welcher Fall gehäufter eintritt, und dann dieser so implementiert werden, dass der Code an dieser Stelle möglichst selten der bedingten Sprung nimmt und stattdessen öfter dem sequentiellen Pfad folgt.

8.3.2 Erweiterungen des Modells

Da eingebettete Systeme immer komplexer werden, nimmt auch die Komplexität der Software weiter zu, doch nicht nur in der Größe des Codes, sondern auch in der Art und Weise, wie dieser aufgebaut ist. So gibt es neben den klassischen Interrupts noch nebenläufige Programme, die dann über – meistens vom Betriebssystem bereitgestellte – definierte Schnittstellen kommunizieren. Somit setzt sich das Gesamtverhalten des Systems nicht nur aus den einzelnen Programmen und der Hardware zusammen, sondern auch durch die Art und Weise, wie sie mit einander kommunizieren. Dabei kann dies nicht nur bei Multitaskingsystemen geschehen; auch verteilte Systeme können ein Gesamtverhalten erzeugen. Deswegen müssen die Kommunikationsschnittstellen gesondert untersucht werden und entsprechend in der Programmnetzliste modellierbar sein. Zusätzlich sollte auch eine adäquate Abstraktion für diese Kommunikationskanäle hergeleitet werden, damit das Modell möglichst kompakt bleibt.

Zur Zeit werden bei der Erzeugung der Programmnetzliste alle Maschineninstruktionen einzeln als eigener Basic-Block aufgeführt, die jeweils einem Knoten im Graphen entsprechen. Daraus resultiert eine große Menge an Knoten im Graph. Eine der von den Basic-Block vorgesehenen Eigenschaften ist, in diese mehrere Instruktionen in einer zusammenzufassen, die eine zusammenhängende Sequenz ohne Verzweigungen bzw. Sprünge in die Sequenz ergeben. Neben diesem Zusammenfassen könnten größere Sequenzen oder eventuell auch Unterfunktionen, die öfters im Programm oder in verschiedenen Programmen vorkommen, mittels separater Beweise verifiziert werden, um daraus eine neue Instruktionszelle zu generieren, die dann den vollständigen Aufruf bzw. die Sequenz ersetzt. Dies ist das gleiche Vorgehen wie in [NWSK11].

Da das Modell ein rein kombinatorisches Modell ist, befinden sich in dem Modell keine echten Zustände, sondern es gibt eine Menge von Signalen, die die jeweiligen Zustände des Programms zu unterschiedlichen Zeitschritten wiedergeben. Auch gibt es in der Programmnetzliste ein und dieselbe Instruktion aus dem Programm an unterschiedlichen Stellen. Dies gilt ebenfalls für die Lese- und Schreiboperationen. Da nun der Entwickler oder auch der Verifikationsingenieur die Korrektheit des Programms beweisen will, ohne sich dabei in das Verifikationsmodell einzuarbeiten, wird an einer Sprache gearbeitet [BVS+14a], die der Verifikationsingenieur oder auch der Entwickler selbst verwenden kann, um die zu verifizierenden Eigenschaften zu beschreiben. Dabei kann er sich auf Adressen in der Hardware oder dem Speicher beziehen und die Interaktion des

Programms mit diesen Speicherstellen verifizieren. Es ist dann nötig, Sequenzen zu beschreiben und die Reihenfolge von Schreib- und Leseoperationen zu bestimmen, um zu prüfen, ob die Software das richtige Protokoll mit der Hardwareumgebung einhält. Da das Modell all solche Informationen unterstützt, ist ein Mapping der benutzerfreundlich beschriebenen Eigenschaften auf das Modell möglich. Dabei lässt sich diese Transformation vollständig automatisieren.

Anhang A

Implementierung der Instruktionszellen

In diesem Kapitel wird die Implementierung der Instruktionszellen ausführlicher beschrieben und die unterschiedlichen Typen dabei vorgestellt. Ausgehend von der in Kapitel 4 vorgestellten abstrakten Sichtweise (Abbildung 4.2) auf den Prozessor, wird für jede Instruktion des Prozessors eine separate Logik angenommen, wie in Abbildung 4.4 dargestellt.

Die Instruktionszellen modellieren dabei die jeweilige Instruktion, die von der ISA bzw. Prozessorimplementierung vorgegeben ist. So ist jede Zelle einmalig, doch lassen sich viele Instruktionen zu größeren Gruppen zusammenfassen, bei denen die Struktur der Instruktionszellen sehr ähnlich sind. Zum einen bilden jene Instruktionen eine Gruppe, die nur arithmetische oder logische Operationen auf den Daten durchführen. Der zweite Typ an Instruktionen sind die Lese- und Schreiboperationen für den Arbeitsspeicher und die Eingabe- und Ausgabegeräte. Den letzten Typ bilden die ablaufmodifizierenden Instruktionen. Sie können die Ausführung des Programms unterbrechen, verzweigen oder zu alten – schon durchlaufenen – Programmteilen zurückspringen. Je nach Architektur gibt es auch noch zusätzliche Konfigurations-Instruktionen für die Verwendung von Interrupts oder architekturspezifische Verhaltensweisen. Je nach Prozessorarchitektur sind die Gruppen von Instruktionen nicht so klar zu trennen. So sind zum Beispiel bei CISC-Architekturen auch Instruktionen vorhanden, die sowohl Daten im Speicher lesen bzw. schreiben und gleichzeitig arithmetische Operationen durchführen. Andere komplexe Instruktionen von unterschiedlichen Architekturen können neue Klassen erfordern, doch lässt sich im Allgemeinen deren Verhalten mittels einer Kombination der drei beschriebenen Gruppen darstellen. Bei einigen ISAs gibt es auch noch zusätzliche architekturspezifische Befehle, zum Beispiel spezielle Befehle zur Konfiguration des Prozessors, kurzzeitiges Unterdrücken von Interrupts, oder unterschiedliche User-Modes. Ein Teil dieser Instruktionen lässt sich in keine der drei genannten Gruppen einordnen.

Im Folgenden wird von einer reinen Load-Store-basierten RISC-Architektur ausgegangen. Die Art und Weise der Implementierung der Instruktionszellen lässt sich aber auch auf andere Architekturen anwenden. Die drei vorgestellten Typen von Instruktionen sind die Typen, die in einer ISA einer RISC-Architektur vorkommen:

- Arithmetische und logische Operationen;

- Lese- und Schreib-Operationen;
- Instruktionen, die den Programmablauf modifizieren.

Die Instruktionszellen bestehen dabei, wie in Kapitel 3 beschrieben, aus rein kombinatorischer Logik ohne Speicherelemente und beschreiben, wie der aktuelle Programmzustand in den nächsten Zustand überführt wird. Dabei wird in dem in Kapitel 6 implementiertem Tool der Programmzustand in zwei Teilen, dem Architekturzustand und den Datenspeicher des Programms, modelliert. Der Architekturzustand wird direkt in den Instruktionszellen dargestellt. Der Speicher variiert je nach Programm und wird deswegen separat modelliert (vgl. Appendix B).

Um die Zellen zu beschreiben, ist eine Sprache, die kombinatorisches Verhalten beschreiben kann, ausreichend. Daher lassen sich die Zellen in beliebigen Sprachen, wie VHDL, Verilog oder auch CTL, ITL ausdrücken, und können somit auch als KNF für den SAT-Solver dargestellt werden. Hier in den folgenden Beispielen wird VHDL als Sprache verwendet, da diese eine für den Menschen gut lesbare ist. Die im Tool implementierte Sprache ist eine XML-basierte Sprache, die an VHDL angelehnt ist und eine Teilmenge der synthetisierbaren VHDL-Instruktionen unterstützt.

Arithmetische und logische Operationen

Die erste Gruppe bilden die Instruktionen, die ausschließlich auf den internen Registern des Prozessors arbeiten. Somit werden nur Werte innerhalb der Datenregister und der für den Programmierer sichtbaren Statusregister modifiziert. Ein einfaches Beispiel zeigt das Listing A.1.

Listing A.1: Instruktionszelle für bitweises logisches UND (AND)

```
1  -- Instruction AND
   entity inst_AND is
   generic(
     Rm : integer;
5    Rn : integer
   );
   port(
     signal Z : out arch_state_type;
     signal S : in arch_state_type
10 );
   end inst_AND;

   architecture arch of inst_AND is
   begin
15 process( S )
   begin
     Z <= S;
     Z.REGFILE(Rn) <= S.REGFILE(Rm) and S.REGFILE(Rn);
   end process;
20 end arch;
```

Die Entity beschreibt die Eingaben der Instruktion. Die Parameter der Instruktionen, wie z.B. Registeradressen, werden als *Generic* übergeben und erst bei der Instanzierung einer Instruktionszelle konkretisiert. Als Moduleingang ist der Zustand S definiert. Anschließend wird mit bitweisen logischen UND (engl. AND) die eigentliche Operation auf den Registern durchgeführt. Das Ergebnis, der komplette Nachfolgezustand Z, ist die Ausgabe des VHDL-Moduls, welches durch die konkrete Implementierung "$arch$" erzeugt wird. Zu dieser Gruppe gehören auch Addition, Subtraktion, Vergleiche, Shift Operationen und ähnliche Operationen, die alle ausschließlich auf den internen Prozessorregistern operieren.

Lese- und Schreib-Operationen

Die zweite Gruppe der Instruktionen bilden die Lade- und Speicher-Operationen, die sich ausschließlich mit Zugriffen auf den Speicher und die Peripherie beschäftigen. In den meisten RISC-basierten Architekturen gibt es nur einen Typ Instruktionen, der sowohl auf Speicher als auch die Peripherie zugreift (vgl. Memory Mapped IO in 2.3.1).

Zwei Beispiele zeigen die Listings A.2 und A.3. Listing A.2 implementiert eine Instruktion, die einen 32-bit Wert aus einer Speicherstelle liest, die in Register Rm gegeben ist. Der gelesene Wert wird in Register Rn gespeichert. Er wird erst im Folgezustand $Z.regfile(Rn)$ sichtbar. Der Speicher ist Teil des Programmzustands und könnte mit den Zuständen S und Z zusammen modelliert werden. Hier in der Beispiel-Zelle wird der Speicher separat vom Ein- und Ausgangszustand modelliert. Der Speicher gehört aus Sicht der Programmnetzliste weiter zum Programmzustand. Durch die separate Modellierung ist die Kommunikation des Prozessors mit dem Businterface einfach zu erkennen:

Die zu lesende Adresse, die an den Bus angelegt werden muss, wird auf den Systembus geschrieben und die Daten, die als Antwort geschickt werden, werden über den Eingabeport $Port_Data_in.DATA$ gelesen.

Zusätzlich wird über den Infoport $Port_Info_out$, neben der Adresse, auch das Active-Bit übermittelt, anhand dessen erkannt werden kann, ob die Instruktion auf dem gerade betrachteten Pfad liegt. Dieses Wissen ist für die spätere Softwareverifikation notwendig.

Listing A.2: Instruktionszelle MOVLL (Lade 32bit Wert)

```
1  -- Abstract Cell: MOVLL
   entity inst_MOVLL is
   generic(
     Rn : integer;
5    Rm : integer
   );
   port(
     signal Z : out arch_state_type;
     signal S : in arch_state_type;
10   signal Port_Info_out : out port_info_type;
     signal Port_Data_in : in port_type
   );
   end inst_MOVLL;
   architecture arch of inst_MOVLL is
```

```
15 begin
   process( S, Port_Data_in )
   begin
     Z <= S;
     Z.REGFILE(Rn) <= Port_Data_in.DATA;
20   Port_Info_out.ADDRESS <= S.REGFILE(Rm);
     Port_Info_out.Active <= S.Active;
   end process;
   end arch;
```

Das Listing A.3 modelliert eine Instruktion, die Daten in den Speicher schreibt. Sie ist vom Aufbau ähnlich zur Lese-Instruktion. So verfügt die Instruktionszelle über denselben Info-Port, welcher die Adresse und das Active-Bit nach außen weiterleitet. Anstelle des Eingabeports für die Daten verfügt die Zelle über einen Pport $Port_Data_out.DATA$ für die Ausgabe, welcher die Daten nach außen leitet, so dass diese auf dem Systembus angelegt werden können. In dieser Instruktion wird der Folgezustand Z nicht modifiziert, da der Programmzustand ausschließlich über die zu schreibenden Daten im Speicher manipuliert wird.

Listing A.3: Instruktionszelle MOVLS (Speicher 32bit Wert)

```
1  entity inst_MOVLS is
   generic(
     Rn : integer;
     Rm : integer
5  );
   port(
     signal Z : out arch_state_type;
     signal S : in arch_state_type;
     signal Port_Info_out : out port_info_type;
10   signal Port_Data_out : out port_type
   );
   end inst_MOVLS;
   architecture arch of inst_MOVLS is
   begin
15 process( S )
   begin
     Z <= S;
     Port_Data_out.DATA <= S.REGFILE(Rm);
     Port_Info_out.ADDRESS <= S.REGFILE(Rn);
20   Port_Info_out.Active <= S.Active;
   end process;
   end arch;
```

Programmablauf modifizierende Instruktionen

Die Klasse der Instruktionen, die den Programmablauf manipulieren, zeichnet sich dadurch aus, dass die zugehörigen Instruktionen im Allgemeinen mehr als einen Nachfolger

haben. Alternativ, wenn sie nur über einen Nachfolger verfügen, ist die Nachfolgeinstruktion nicht die Instruktion, die an der nachfolgenden Speicherstelle im Programm steht.

Eine der wichtigsten Instruktionen ist ein bedingter Sprung, der, je nachdem, ob eine gegebene Bedingung erfüllt ist, den Sprung durchführt oder einfach die nächste Instruktion in der Sequenz ausführt. Listing A.4 zeigt ein Beispiel für eine bedingte Verzweigung mit dem Namen "Branch False"(kurz: BF). Die Sprungbedingung ist dann erfüllt, wenn die eigentliche Bedingung nicht erfüllt ist. In diesem Beispiel wird die Bedingung durch die vorherigen Instruktionen getestet und in einem Konfigurationsregister des Prozessors gespeichert, welches im Zustand als $S.flag_T$ bezeichnet ist. Die Entity der Instruktion definiert neben dem Eingangszustand S zwei Ausgangszustände Z_t und Z_nt. Z_t ist der Nachfolger für den Fall, dass der Sprung ausgeführt wird. Entsprechend steht Z_nt für den Pfad, wenn der Sprung nicht durchgeführt wurde. Da das Active-Bit nur zu Beginn des Programms einmal generiert wird und danach nur noch propagiert werden darf, ist entsprechend für die Verzweigung auch nur eine Propagierung vorgesehen. Somit wird anstelle des Werts 1 bei dem aktiven Pfad der Wert 0 bei dem inaktiven Pfad geschrieben. Dies garantiert, dass nicht mehrere Pfade im Programm gleichzeitig aktiv werden können.

Listing A.4: Instruktionszelle Branch False (Bedingter Sprung)

```
1  entity inst_BF is
   -- no generics
   port(
     signal Z_t : out arch_state_type;
5    signal Z_nt : out arch_state_type;
     signal S : in arch_state_type
   );
   end inst_BF;
   architecture arch of inst_BF is
10 begin
   process( S )
   begin
     Z_t <= S;
     Z_nt <= S;
15   --control
     if S.flag_T = '0' then
       Z_t.Active <= S.Active;
       Z_nt.Active <= false;
     else
20     Z_t.Active <= false;
       Z_nt.Active <= S.Active;
     end if;
   end process;
   end arch;
```

Der Jump-Befehl hingegen hat nur einen Nachfolgerzustand. Doch dabei ist es wichtig das Sprungziel zu kennen und zu überprüfen, ob es konstant ist. Dies wird in Kapitel 4.2.6 genauer beschrieben.

A.1 Zyklenzähler innerhalb der Instruktionen

Die vorher vorgestellten drei Gruppen und zugehörigen Zellen modellieren das vollständige Verhalten, welches beim Aufruf der einzelnen Instruktionen vom Prozessor durchgeführt wird entsprechend der Beschreibung in der Spezifikation. Mit den Zellen kann somit auch das Gesamtverhalten des Programms in der Programmnetzliste dargestellt werden. Die vorgestellten Zellen sind zeitabstrakt auf dem ISA-Level beschrieben und verfügen somit über keine konkrete Zeitinformation. Nur die Abarbeitungsreihenfolge der Instruktionen ist bekannt. Für einige Beweise ist es dennoch notwendig, die Ausführungszeitpunkte abzugleichen, oder bestimmte Sequenzen zu erkennen. Dafür ist eine zeitliche Beschreibung hilfreich. In der Programmnetzliste ist keine konkrete Zeit vorgesehen, aber es kann eine abstrakte Zeit eingeführt werden. Jeder Instruktion wird eine fest vorgegebene Ausführungszeit zugeordnet, die dann über den Pfad aufsummiert werden kann. Dieser Ansatz erlaubt sogar exakte Berechnungen von Ausführungszeiten bei bestimmten einfachen Rechnerarchitekturen, die entweder keine Pipeline besitzen oder bei denen die Pipeline statisch ist und die Bedingungen für *pipeline stalls* einfach berechnet werden können (z.B. Xilinx PicoBlaze™).

Zur Realisierung in der Programmnetzliste wird der Programmzustand zusätzlich um eine Zählervariable ergänzt und die Logik in den Zellen um einen Addierer erweitert. In Listing A.5 ist die Instruktion Branch False aus dem Listing A.4 um die zusätzlichen Eigenschaften für die Abarbeitung der Zählervariable ergänzt.

Listing A.5: Instruktionszelle Branch False mit Zählervariable

```
1  entity inst_BF is
   generic(
     Ex_Cycles_t : integer;
     Ex_Cycles_nt : integer
5  );
   port(
     signal Z_t : out arch_state_type;
     signal Z_nt : out arch_state_type;
     signal S : in arch_state_type
10 );
   end inst_BF;
   architecture arch of inst_BF is
   begin
   process( S )
15 begin
     Z_t <= S;
     Z_nt <= S;
     --control
     if S.flag_T = '0' then
20     Z_t.Active <= S.Active;
       Z_nt.Active <= false;
       Z_t.Cycles <= S.Cycles + Ex_Cycles_t;
       Z_nt.Cycles <= S.Cycles;
     else
```

```
25      Z_t.Active <= false;
        Z_nt.Active <= S.Active;
        Z_t.Cycles <= S.Cycles;
        Z_nt.Cycles <= S.Cycles + Ex_Cycles_nt;
      end if;
30 end process;
   end arch;
```

In den Generics wird nun die Zeitdauer der Instruktion übergeben. Im Falle einer Branch-Instruktion verfügt die Instruktionszelle als Parameter über zwei Ausführungszeiten. Die eine für den Fall, dass der Zweig weitergeführt wird, der vom Prozessor erwartet wurde und der zweite für den Fall, dass vom erwarteten Pfad abgewichen wird. Eine normale Instruktion verfügt nur über einen Wert.

Nachdem der Eingangszustand S auf die beiden Ausgangszustände gelegt wurde, werden neben den Active-Bits auch die neuen Ausführungszeitpunkte berechnet. Dabei wird der Wert aus $S.cycles$ mit dem jeweiligen Fall zugehörigen Generic zusammen addiert und in den Nachfolgezustand $Z_t.cycles$ bzw. $Z_nt.cycles$ gespeichert.

Je nach betrachteter Architektur können die Zeiten variieren. Wird als Beispiel ein Prozessor mit einer mehrstufigen Pipeline und ohne Sprungvorhersage (*branch prediction*) betrachtet, muss in dem einen Fall die Pipeline geleert werden, in dem anderen kann sie einfach weiterarbeiten. Dies führt zu unterschiedlichen Ausführungszeiten, je nach Wahl der Verzweigung. Der Grund liegt daran, dass die Sprungbedingung erst berechnet wird, wenn die Instruktion schon geladen und decodiert wurde. So ist die Branch-Instruktion schon mindestens in der dritten Pipelinestufe und zwei Nachfolgeinstruktionen sind schon in die Pipeline geladen. Wird der Sprung nicht wahrgenommen, sondern einfach die Sequenz weiter ausgeführt, dann hat die Instruktion nur die Ausführungszeit einer Instruktion in einer gefüllten Pipeline. Dies ist in der zeitabstrakten Betrachtungsweise eine Zeiteinheit. Im Falle, dass der Sprung genommen wird, befinden sich mindestens zwei falsche Instruktionen in den ersten zwei Stufen der Pipeline. Somit müssen diese erst entleert und anschließend neu befüllt werden. Die Ausführungszeit des Sprungs errechnet sich aus der Gesamtzeit für den Wechsel zu den neuen Befehlen. Die Summe errechnet sich aus einer Zeiteinheit für die Instruktion selbst, eine für das Leeren der Pipelinestufen und zwei Zeiteinheiten um die Pipelinestufen wieder mit neuen Befehlen zu befüllen.

Wenn das Entleeren und Befüllen 4 Zeitschritte dauern würde und das Weitermachen nur einen, dann würde in dem Listing A.5 $Ex_Cycles_t = 4$ und $Ex_Cycles_nt = 1$ als Generic übergeben.

Die Zählvariable im Listing wird mittels eines 32-Bit Integer repräsentiert, entsprechend werden für die Summenbildung 32-Bit-Addierer benötigt, die eine gewisse Komplexität besitzen, so dass sie nur in den instanzierten Zellen implementiert werden sollten, wenn die Zähler tatsächlich für den Beweis benötigt werden. Zur Generierung der Programmnetzliste werden diese hingegen nicht benötigt. Die Implementierung des Tools aus Kapitel6 berücksichtigt dies und optimiert die Addierer bei den Programmnetzlisten weg, die für die Beweise der Verzweigungen oder bei der Adressberechnung während der

Generierung benötigt werden.

Als Hinweis: Eine Zelle mit zwei Nachfolgerzuständen muss nur einen Addierer besitzen, da die jeweilige Addition nur für den entsprechenden aktiven Pfad benötigt wird, der nur durch einen der zwei Ausgänge läuft. Daher muss nur eine Addition durchgeführt werden, bei der mittels eines Multiplexers der Addierer die richtigen Werte erhält. Dieser Multiplexer benötigt weit weniger Hardware als ein zweiter Addierer.

Anhang B

Zusätzliche Logik für das Property Checking

Das Ziel der Generierung der Programmnetzliste ist es, ein Modell für die formale Verifikation zu erstellen, auf dem das Verhalten der Software auf dem entsprechenden Prozessor verifiziert werden kann. Dabei können unterschiedliche Verifikationsziele betrachtet werden.

Im einfachen Fall übernimmt die Software eine Aufgabe, in der nur das Ergebnis zählt, nicht jedoch, wie es generiert wird. Es ist in dem Fall ausreichend ausschließlich die Startbedingungen und das Ergebnis zu betrachten. Typische Beispiele sind Sortier-Algorithmen oder auch Berechnungsalgorithmen.

Die zugehörigen Eigenschaften prüfen das Verhalten der Software ausschließlich nach Beendigung des Programms, ob die korrekten Berechnungen oder Datenbewegungen ausgeführt worden. Somit wird die Programmnetzliste als eine Art "Black-Box" verwendet. Es werden in den Eigenschaften ausschließlich Signale betrachtet, die primäre Ein- oder Ausgänge der Programmnetzliste sind.

Dabei kann auch mittels einfacher If-Then-Else-Konstrukte automatisierte Makros für jede Speicherzelle erstellt werden, die den Wert des Speichers zum Ende des Programms zeigen.

In Abbildung B.1 wird ein Beispiel für die notwendige Logik eines solchen Makros gezeigt. Dabei greifen die Schreibzugriffe in diesem Beispiel nur auf eine einzige Speicherstelle zu. Diese werden mittels Multiplexer hintereinander in der Reihenfolge ihres Auftretens geschaltet. Die Entscheidung, ob der geschriebene Wert verwendet wird, hängt immer davon ab, ob die zugehörige Instruktionszelle auf dem aktiven Pfad liegt.

Der Wert des Schreibzugriff von der Instruktionszelle b wird auf jeden Fall von einer anderen Instruktion überschrieben, da beide Pfade, die durch b führen, im Anschluss eine 2te Schreibinstruktion passieren (d oder e). Je nach Pfad ist der geschriebene Wert von der Instruktionszelle d oder Zelle e der korrekte letzte Wert.

Die Modellierung der Schreibzugriffe ist einfach, da nur einer der beiden Pfade gleichzeitig aktiv sein kann. Daraus folgt, dass d und e nicht gleichzeitig aktiv sein können und keine zueinander zeitliche Abhängigkeit besitzen. Die Speicherzugriffe von d und e kön-

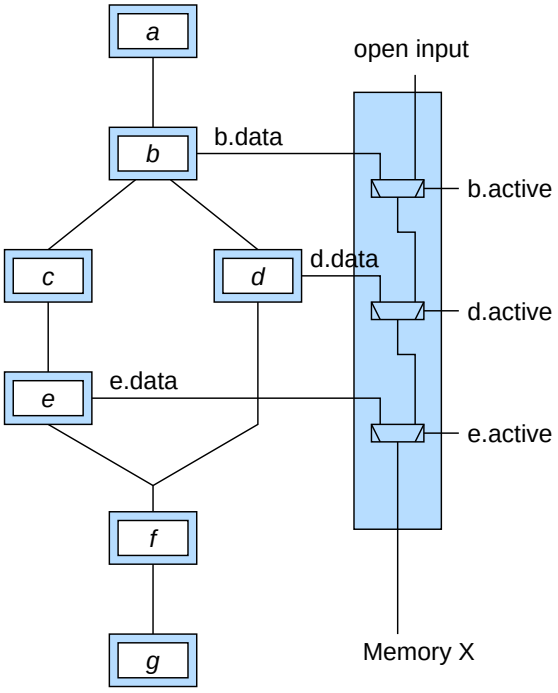

Abbildung B.1: Logik zum generieren des Speicherinhaltes

nen in der Sequenz in beliebiger Reihenfolge zueinander in die Gesamtsequenz eingereiht werden. In dem Makro (blaue Fläche mit den Multiplexern) wird die Halbordnung der Speicherzugriffe der Instruktionszellen erhalten, die schon im Execution-Graph erzeugt wurde.

Diese Makros entsprechen der Logik, die für die Generierung der separaten Speichermodellierung verwendet wird (s. Kapitel 4.4). Dadurch können diese Makros – ohne großen Mehraufwand – automatisiert für die Programmnetzliste generiert werden.

Die generierten Makros sind nicht optimiert, bilden aber das Verhalten korrekt ab. Im Beispiel aus Abbildung B.1 könnte der Schreib-Zugriff von der Instruktionszelle b auf den Speicher im Makro vernachlässigt werden, da er, sofern b aktiv wird definitiv von der Instruktion d oder e überschrieben wird. Eine solche Optimierung ist mittels statischer Methoden möglich.

Der Fokus für hardwarenahe Software liegt aber eher bei Programmen, die mit der Hardware interagieren. Dabei ist die Betrachtung von der Kommunikation zur Peripherie wichtig. Ein wesentliches Merkmal dabei ist, dass die Kommunikationssequenz für das korrekte Verhalten relevant ist. So werden zum Beispiel bei Protokollen mehrfach Daten gesendet, die aber jeweils als Sequenz an dasselbe Ziel-Register in der Peripherie geschrieben werden.

Würde ausschließlich nach Ausführung des Programms verifiziert, welcher Wert an der zu prüfenden Speicherstelle geschrieben wurde, wäre nur der zuletzt geschriebene Wert sichtbar.

Deswegen müssen die kompletten Sequenzen verifiziert werden. Die Sequenz ist über

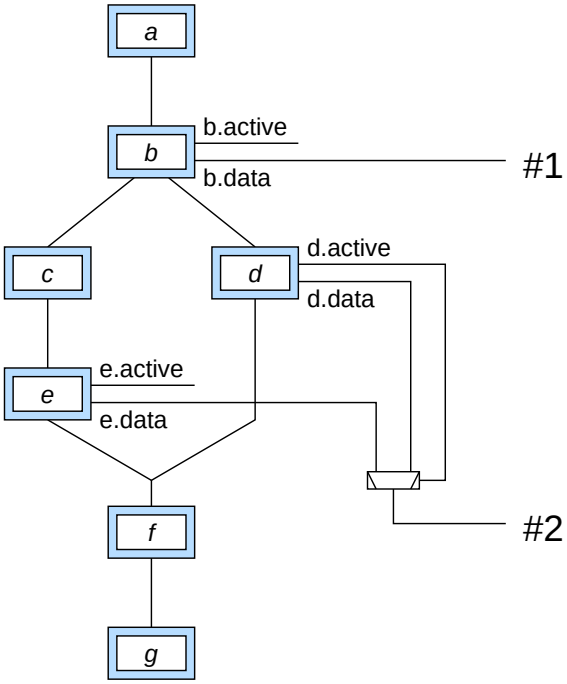

Abbildung B.2: Logik zum überprüfen der Sequenz

die Pfade definiert und im Execution-Graph statisch. In Abbildung B.2 ist die erste Speicheroperation (#1) immer die Ausgabe der Instruktion b. Die zweite Speicheroperation kann je nach Pfad entweder von der Instruktionszelle d oder e ausgegeben werden. Da sie selbst wieder auf unterschiedlichen Pfaden liegen, werden die beiden Speicherwerte wieder gemultiplext und anhand des Active-Bit entschieden, welche der beiden Instruktionen die aktuell betrachtete ist.

Ein weiteres Verifikationsziel ist das Überprüfen eines sequentiellen Programms auf die Reihenfolge von Schreiboperationen, die auf unterschiedliche Speicheradressen schreiben. Zum Beispiel sollte eine UART erst konfiguriert werden, bevor sie beginnt die Daten zu übermitteln. Dadurch ist die Reihenfolge der Schreiboperationen entscheidend für das korrekte verwenden der UART wichtig und sollte auch von der Verifikation überprüfbar sein.

Um die Reihenfolge solcher Sequenzen von unterschiedlichen Speicherelementen zu überprüfen, werden Portmapping Makros aus Kapitel 4.4 verwendet und zusätzlich auch die in Kapitel A.1 eingeführten Zähler benötigt. Dabei sind diese eingeführten Zählvariablen im Status auch als Information für die Ein- und Ausgabe-Ports verfügbar, so dass an den entsprechenden Stellen die gezählten Werte verglichen werden können. Um zu überprüfen, in welcher Reihenfolge die Schreiboperationen ausgeführt werden, ist dann ein Vergleich der Zählvariablen ausreichend. Die Schreiboperation mit dem größeren Wert, ist die Operation, die später ausgeführt wird. Dieses Verfahren kann genutzt werden, um diese Aussage mittels eines SAT-Solvers bestätigen zu lassen.

Angenommen, dass in Abbildung B.3 die Instruktion b an Speicherstelle X schreibt

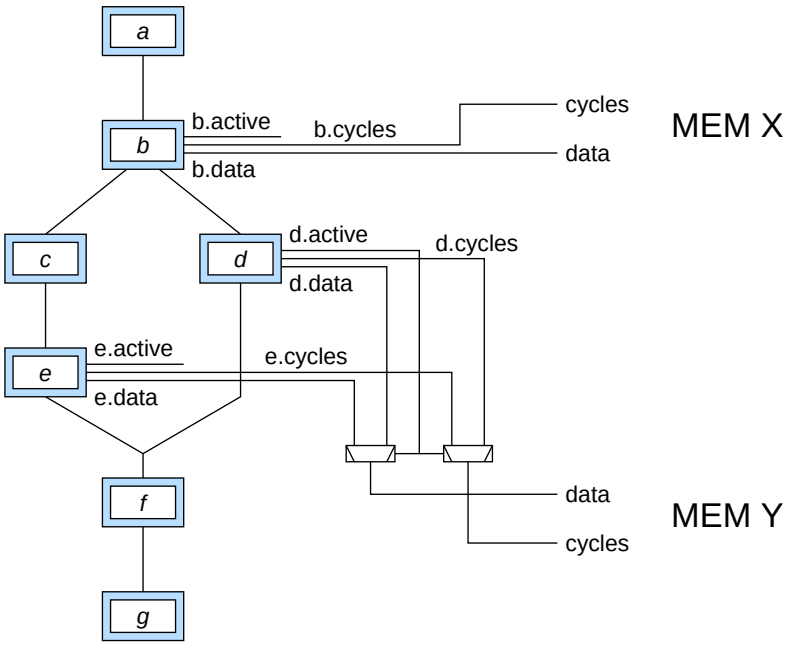

Abbildung B.3: Logik zum Überprüfen der Reihenfolge in einer Sequenz

und die Instruktionen d und e schreiben auf die Speicherstelle Y. Um zu zeigen, dass zuerst auf die Speicherstelle X und dann auf Y geschrieben wird, werden die Sequenzen der einzelnen Speicherbereiche erstellt. Die Eigenschaft, die zeigt, dass der Speicherzugriff auf X immer vor dem Speicherzugriff auf Y geschieht, muss folgende Bedingung abprüfen: $X.cycles < Y.cycles$.

Gilt diese Bedingung, wurde erfolgreich gezeigt, dass die zu überprüfende Bedingung immer erfüllt ist und somit die korrekte Schreibreihenfolge eingehalten wird.

In Abbildung B.3 ist die Reihenfolge der Schreibsequenzen auch ohne eine Eigenschaftsprüfung mittels SAT offensichtlich. Die Reihenfolge der Schreibsequenzen ist durch den Execution-Graphen in der Programmnetzliste implizit schon gegeben, da sie durch die beschränkte Anzahl an möglichen Pfaden beschränkt ist. Es ist möglich, mit Hilfe von statischen Verfahren, die Reihenfolge direkt in dem Graphen zu berechnen, auch ohne den Einsatz eines SAT-Solvers.

Welche der beiden Möglichkeiten genutzt wird, bleibt dem Anwender überlassen.

Anhang C

Code-Beispiele

C.1 Eigenschaft: Selbsttest Stuck-at-Fehler erkennen

Die Eigenschaft in Listing C.1 wurde, wie in Kapitel 7.2 beschrieben, iterativ so oft gestartet, bis alle Gegenbeispiele gefunden wurden. Dabei werden die Gegenbeispiele, bei jedem Durchgang, durch zusätzliche Annahmen ausgenommen, so dass diese nicht mehr auftreten können.

Listing C.1: Eigenschaft zum Erkennen von Registerbits mit konstanten Wert

```
1  property never_modified_bit is
   dependencies: constraint_list;
   -- Anzahl Register:
   for: i=0..7,
5  -- Die einzelnen Bits (31)
         j=0..31;

   assume:
   -- Startannahmen
10 at t: s.active = true;
   at t: s.cycles = 0;

   -- Potentielle Fehler die uebersehen werden koennen, bzw komplett
      ungetestet:

15 -- Regfile(7) Bit (31): ungetestet
   at t: if i=7 and j=31 then false end if;
   -- Regfile(7) Bit (30): ungetestet
   at t: if i=7 and j=30 then false end if;
   -- Regfile(7) Bit (29): ungetestet
20 at t: if i=7 and j=29 then false end if;
   -- Regfile(7) Bit (28): ungetestet
   at t: if i=7 and j=28 then false end if;
   -- Regfile(7) Bit (27): ungetestet
   at t: if i=7 and j=27 then false end if;
25 -- Regfile(7) Bit (26): ungetestet
   at t: if i=7 and j=26 then false end if;
```

```
-- Regfile(7) Bit (25): ungetestet
at t: if i=7 and j=25 then false end if;
-- Regfile(7) Bit (24): ungetestet
at t: if i=7 and j=24 then false end if;
-- Regfile(7) Bit (23): ungetestet
at t: if i=7 and j=23 then false end if;
-- Regfile(7) Bit (22): ungetestet
at t: if i=7 and j=22 then false end if;
-- Regfile(7) Bit (21): ungetestet
at t: if i=7 and j=21 then false end if;
-- Regfile(7) Bit (20): ungetestet
at t: if i=7 and j=20 then false end if;
-- Regfile(7) Bit (19): ungetestet
at t: if i=7 and j=19 then false end if;
-- Regfile(7) Bit (18): ungetestet
at t: if i=7 and j=18 then false end if;
-- Regfile(7) Bit (17): ungetestet
at t: if i=7 and j=17 then false end if;
-- Regfile(7) Bit (16): ungetestet
at t: if i=7 and j=16 then false end if;
-- Regfile(7) Bit (15): Stack-at-0
at t:  s.regfile(7)(15)=1;
-- Regfile(7) Bit (14): Stack-at-0
at t:  s.regfile(7)(14)=1;
-- Regfile(7) Bit (13): Stack-at-0
at t:  s.regfile(7)(13)=1;
-- Regfile(7) Bit (12): Stack-at-0
at t:  s.regfile(7)(12)=1;
-- Regfile(7) Bit (11): Stack-at-0
at t:  s.regfile(7)(11)=1;
-- Regfile(7) Bit (10): Stack-at-0
at t:  s.regfile(7)(10)=1;
-- Regfile(7) Bit (9): Stack-at-0
at t:  s.regfile(7)(9)=1;
-- Regfile(7) Bit (8): Stack-at-0
at t:  s.regfile(7)(8)=1;
-- Regfile(7) Bit (7): Stack-at-0
at t:  s.regfile(7)(7)=1;
-- Regfile(7) Bit (6): Stack-at-0
at t:  s.regfile(7)(6)=1;
-- Regfile(7) Bit (5): Stack-at-0
at t:  s.regfile(7)(5)=1;
-- Regfile(7) Bit (4): Stack-at-0
at t:  s.regfile(7)(4)=1;

-- Regfile(6) Bit (30): Stack-at-0
at t:  s.regfile(6)(30)=1;
-- Regfile(6) Bit (26): Stack-at-0
at t:  s.regfile(6)(26)=1;
-- Regfile(6) Bit (20): Stack-at-0
at t:  s.regfile(6)(20)=1;
-- Regfile(6) Bit (16): Stack-at-0
```

```
at t:  s.regfile(6)(16)=1;

-- Regfile(5) Bit (30): Stack-at-0
at t:  s.regfile(5)(30)=1;
-- Regfile(5) Bit (28): Stack-at-0
at t:  s.regfile(5)(28)=1;
-- Regfile(5) Bit (27): Stack-at-1
at t: s.regfile(5)(27)=0;
-- Regfile(5) Bit (26): Stack-at-0
at t:  s.regfile(5)(26)=1;
-- Regfile(5) Bit (12): Stack-at-0
at t:  s.regfile(5)(12)=1;
-- Regfile(5) Bit (11): Stack-at-1
at t:  s.regfile(5)(11)=0;
-- Regfile(5) Bit (10): Stack-at-0
at t:  s.regfile(5)(10)=1;

-- Regfile(3) Bit (31): Stack-at-0
at t:  s.regfile(3)(31)=1;
-- Regfile(3) Bit (22): Stack-at-0
at t:  s.regfile(3)(22)=1;
-- Regfile(3) Bit (18): Stack-at-0
at t:  s.regfile(3)(18)=1;
-- Regfile(3) Bit (6): Stack-at-0
at t:  s.regfile(3)(6)=1;

-- Regfile(1) Bit (29): Stack-at-0
at t: s.regfile(1)(29)=1;
-- Regfile(1) Bit (25): Stack-at-0
at t: s.regfile(1)(25)=1;
-- Regfile(1) Bit (21): Stack-at-0
at t: s.regfile(1)(21)=1;
-- Regfile(1) Bit (17): Stack-at-0
at t: s.regfile(1)(17)=1;
-- Regfile(1) Bit (13): Stack-at-0
at t: s.regfile(1)(13)=1;
-- Regfile(1) Bit (9): Stack-at-0
at t: s.regfile(1)(9)=1;

prove:
-- Test ob ein Bit im Registerfile nie einen anderen Wert
   geschrieben bekommt.
at t: not(s.regfile(i)(j) = s0.regfile(i)(j) and
 s.regfile(i)(j) = s1.regfile(i)(j) and
 s.regfile(i)(j) = s2.regfile(i)(j) and
 s.regfile(i)(j) = s3.regfile(i)(j) and
 s.regfile(i)(j) = s4.regfile(i)(j) and
 s.regfile(i)(j) = s5.regfile(i)(j) and
 s.regfile(i)(j) = s6.regfile(i)(j) and
 s.regfile(i)(j) = s7.regfile(i)(j) and
 s.regfile(i)(j) = s8.regfile(i)(j) and
 s.regfile(i)(j) = s9.regfile(i)(j) and
```

```
130 s.regfile(i)(j) = s10.regfile(i)(j) and
    s.regfile(i)(j) = s11.regfile(i)(j) and
    s.regfile(i)(j) = s12.regfile(i)(j) and
    s.regfile(i)(j) = s13.regfile(i)(j) and
    s.regfile(i)(j) = s14.regfile(i)(j) and
135 s.regfile(i)(j) = s15.regfile(i)(j) and
    s.regfile(i)(j) = s16.regfile(i)(j) and
    s.regfile(i)(j) = s17.regfile(i)(j) and
    s.regfile(i)(j) = s18.regfile(i)(j) and
    s.regfile(i)(j) = s19.regfile(i)(j) and
140 s.regfile(i)(j) = s20.regfile(i)(j) and
    s.regfile(i)(j) = s21.regfile(i)(j) and
    s.regfile(i)(j) = s22.regfile(i)(j) and
    s.regfile(i)(j) = s23.regfile(i)(j) and
    s.regfile(i)(j) = s24.regfile(i)(j) and
145 s.regfile(i)(j) = s25.regfile(i)(j) and
    s.regfile(i)(j) = s26.regfile(i)(j) and
    s.regfile(i)(j) = s27.regfile(i)(j) and
    s.regfile(i)(j) = s28.regfile(i)(j) and
    s.regfile(i)(j) = s29.regfile(i)(j) and
150 s.regfile(i)(j) = s30.regfile(i)(j) and
    s.regfile(i)(j) = s31.regfile(i)(j) and
    s.regfile(i)(j) = s32.regfile(i)(j) and
    s.regfile(i)(j) = s33.regfile(i)(j) and
    s.regfile(i)(j) = s34.regfile(i)(j) and
155 s.regfile(i)(j) = s35.regfile(i)(j) and
    s.regfile(i)(j) = s36.regfile(i)(j) and
    s.regfile(i)(j) = s37.regfile(i)(j) and
    s.regfile(i)(j) = s38.regfile(i)(j) and
    s.regfile(i)(j) = s39.regfile(i)(j) and
160 s.regfile(i)(j) = s40.regfile(i)(j) and
    s.regfile(i)(j) = s41.regfile(i)(j) and
    s.regfile(i)(j) = s42.regfile(i)(j) and
    s.regfile(i)(j) = s43.regfile(i)(j) and
    s.regfile(i)(j) = s44.regfile(i)(j) and
165 s.regfile(i)(j) = s45.regfile(i)(j) and
    s.regfile(i)(j) = s46.regfile(i)(j) and
    s.regfile(i)(j) = s47.regfile(i)(j) and
    s.regfile(i)(j) = s48.regfile(i)(j) and
    s.regfile(i)(j) = s49.regfile(i)(j) and
170 s.regfile(i)(j) = s50.regfile(i)(j) and
    s.regfile(i)(j) = s51.regfile(i)(j) and
    s.regfile(i)(j) = s52.regfile(i)(j) and
    s.regfile(i)(j) = s53.regfile(i)(j) and
    s.regfile(i)(j) = s54.regfile(i)(j) and
175 s.regfile(i)(j) = s55.regfile(i)(j) and
    s.regfile(i)(j) = s56.regfile(i)(j) and
    s.regfile(i)(j) = s57.regfile(i)(j) and
    s.regfile(i)(j) = s58.regfile(i)(j) and
    s.regfile(i)(j) = s59.regfile(i)(j) and
180 s.regfile(i)(j) = s60.regfile(i)(j) and
    s.regfile(i)(j) = s61.regfile(i)(j) and
```

```
s.regfile(i)(j) = s62.regfile(i)(j) and
s.regfile(i)(j) = s63.regfile(i)(j) and
s.regfile(i)(j) = s64.regfile(i)(j) and
s.regfile(i)(j) = s65.regfile(i)(j) and
s.regfile(i)(j) = s66.regfile(i)(j) and
s.regfile(i)(j) = s67.regfile(i)(j) and
s.regfile(i)(j) = s68.regfile(i)(j) and
s.regfile(i)(j) = s69.regfile(i)(j) and
s.regfile(i)(j) = s70.regfile(i)(j) and
s.regfile(i)(j) = s71.regfile(i)(j) and
s.regfile(i)(j) = s72.regfile(i)(j) and
s.regfile(i)(j) = s73.regfile(i)(j) and
s.regfile(i)(j) = s74.regfile(i)(j) and
s.regfile(i)(j) = s75.regfile(i)(j) and
s.regfile(i)(j) = s76.regfile(i)(j) and
s.regfile(i)(j) = s77.regfile(i)(j) and
s.regfile(i)(j) = s78.regfile(i)(j) and
s.regfile(i)(j) = s79.regfile(i)(j) and
s.regfile(i)(j) = s80.regfile(i)(j) and
s.regfile(i)(j) = s81.regfile(i)(j) and
s.regfile(i)(j) = s82.regfile(i)(j) and
s.regfile(i)(j) = s83.regfile(i)(j) and
s.regfile(i)(j) = s84.regfile(i)(j) and
s.regfile(i)(j) = s85.regfile(i)(j) and
s.regfile(i)(j) = s86.regfile(i)(j) and
s.regfile(i)(j) = s87.regfile(i)(j) and
s.regfile(i)(j) = s88.regfile(i)(j) and
s.regfile(i)(j) = s89.regfile(i)(j) and
s.regfile(i)(j) = s90.regfile(i)(j) and
s.regfile(i)(j) = s91.regfile(i)(j) and
s.regfile(i)(j) = s92.regfile(i)(j) and
s.regfile(i)(j) = s93.regfile(i)(j) and
s.regfile(i)(j) = s94.regfile(i)(j) and
s.regfile(i)(j) = s95.regfile(i)(j) and
s.regfile(i)(j) = s96.regfile(i)(j) and
s.regfile(i)(j) = s97.regfile(i)(j) and
s.regfile(i)(j) = s98.regfile(i)(j) and
s.regfile(i)(j) = s99.regfile(i)(j) and
s.regfile(i)(j) = s100.regfile(i)(j) and
s.regfile(i)(j) = s101.regfile(i)(j) and
s.regfile(i)(j) = s102.regfile(i)(j) and
s.regfile(i)(j) = s103.regfile(i)(j) and
s.regfile(i)(j) = s104.regfile(i)(j) and
s.regfile(i)(j) = s105.regfile(i)(j) and
s.regfile(i)(j) = s106.regfile(i)(j) and
s.regfile(i)(j) = s107.regfile(i)(j) and
s.regfile(i)(j) = s108.regfile(i)(j) and
s.regfile(i)(j) = s109.regfile(i)(j) and
s.regfile(i)(j) = s110.regfile(i)(j) and
s.regfile(i)(j) = s111.regfile(i)(j) and
s.regfile(i)(j) = s112.regfile(i)(j) and
s.regfile(i)(j) = s113.regfile(i)(j) and
```

```
    s.regfile(i)(j) = s114.regfile(i)(j) and
235 s.regfile(i)(j) = s115.regfile(i)(j) and
    s.regfile(i)(j) = s116.regfile(i)(j) and
    s.regfile(i)(j) = s117.regfile(i)(j) and
    s.regfile(i)(j) = s118.regfile(i)(j) and
    s.regfile(i)(j) = s119.regfile(i)(j) and
240 s.regfile(i)(j) = s120.regfile(i)(j) and
    s.regfile(i)(j) = s121.regfile(i)(j) and
    s.regfile(i)(j) = s122.regfile(i)(j) and
    s.regfile(i)(j) = s123.regfile(i)(j) and
    s.regfile(i)(j) = s124.regfile(i)(j) and
245 s.regfile(i)(j) = s125.regfile(i)(j) and
    s.regfile(i)(j) = s126.regfile(i)(j) and
    s.regfile(i)(j) = s127.regfile(i)(j) and
    s.regfile(i)(j) = s128.regfile(i)(j) and
    s.regfile(i)(j) = s129.regfile(i)(j) and
250 s.regfile(i)(j) = s130.regfile(i)(j) and
    s.regfile(i)(j) = s131.regfile(i)(j) and
    s.regfile(i)(j) = s132.regfile(i)(j) and
    s.regfile(i)(j) = s133.regfile(i)(j) and
    s.regfile(i)(j) = s134.regfile(i)(j) and
255 s.regfile(i)(j) = s135.regfile(i)(j) and
    s.regfile(i)(j) = s136.regfile(i)(j) and
    s.regfile(i)(j) = s137.regfile(i)(j) and
    s.regfile(i)(j) = s138.regfile(i)(j) and
    s.regfile(i)(j) = s139.regfile(i)(j) and
260 s.regfile(i)(j) = s140.regfile(i)(j) and
    s.regfile(i)(j) = s141.regfile(i)(j) and
    s.regfile(i)(j) = s142.regfile(i)(j) and
    s.regfile(i)(j) = s143.regfile(i)(j) and
    s.regfile(i)(j) = s144.regfile(i)(j));
265 end property;
```

C.2 Eigenschaft: LIN Frame Empfangen von 4 Datenbytes

Diese hier gezeigte Eigenschaft in Listing C.2 stammt aus einer komplexeren Eigenschaft, welche unterschiedlich große LIN-Frames abprüfen kann. Ebenfalls wurden die realen Adressen und Zustandswerte mittels Makros abstrahiert bzw. durch entsprechende selbsterklärende Variablennamen ersetzt. Zum Beispiel wurde der Zustand X“04” des Transivers durch “receiving_data” ersetzt um die Eigenschaft lesbar zu machen.

Listing C.2: Eigenschaft zum Verifizieren eines empfangenen LIN Frames mit 4 Datenbytes

```
1  property vericifaction_LIN_RX_Frame_4_DB_checksum is
   dependencies: constraint_list_all;

   assume:
5  -- Active_bits werden gesetzt.
   at t: main_active;
   -- RX1 bis RX5 stehen f{\"u}r die Empfangenen Werte,
   -- RX0 beinhaltet die PID der empfangenen Nachricht
   -- Berechnung der Checksume (Annahme, das empfangene Checksumme
      korrekt ist)
10 at t: if (RX0 = RX_ID2)
         then
         RX5 = checksum_4bytes(RX1, RX2, RX3, RX4)
         end if;

15 prove:
   --LIN Frame:
   --LIN Header:
   --LIN Break Field, Main Loop: Send Break Signal
   at t: UART_IO_Register_after_Scheduler_Call0 = break_signal;
20 at t: UART_Status_Register_after_Scheduler_Call0 =
      UART_send_signal_set;

   -- LIN Break Field, Transceiver-State:
   at t: LIN_Transceiver_Status_after_Scheduler_Call0 =
      Break_Signal_sended;

25 -- LIN Synch Field, First IRQ: Send Synch Signal
   at t: UART_IO_Register_after_IRQ0 = sync_signal;
   at t: UART_Status_Register_after_IRQ0 = UART_send_signal_set;
   at t: LIN_Transceiver_Status_after_irq0 = Break_Signal_received;

30 -- LIN PID Field, Second IRQ: Send ID Signal (no other PIDs possible
      )
   at t: UART_IO_Register_after_IRQ1 = RX_ID0 or
      UART_IO_Register_after_IRQ1 =
   RX_ID1 or UART_IO_Register_after_IRQ1 = RX_ID2
      or UART_IO_Register_after_IRQ1 = TX_ID0 or
         UART_IO_Register_after_IRQ1 =
```

```
  TX_ID1 or UART_IO_Register_after_IRQ1 = TX_ID2;
35 at t: UART_Status_Register_after_IRQ1 = UART_send_signal_set;
  at t: LIN_Transceiver_Status_after_irq1 = Snchronisation empfangen;

  -- LIN Response - Datafields
  at t: if (UART_IO_Register_after_IRQ1 = RX_ID2)
40          -- State of LIN Node
         then LIN_Transceiver_Status_after_irq2= receiving_data
          and LIN_Transceiver_Status_after_irq3= receiving_data
          and LIN_Transceiver_Status_after_irq4= receiving_data
          and LIN_Transceiver_Status_after_irq5= receiving_data
45        and LIN_Transceiver_Status_after_irq6= receiving_data
          and LIN_Transceiver_Status_after_irq7= Idle
          -- Error status
          and irq7_memory_0x2060(7 downto 0) = no_error
          -- Correct number of Databytes
50        and irq3_memory_0x2056(7 downto 0) = X
          -- Data Saved at correct position
          and  irq3_memory_0x204e(7 downto 0) = RX1
          and  irq4_memory_0x204f(7 downto 0) = RX2
          and  irq5_memory_0x2050(7 downto 0) = RX3
55   and irq6_memory_0x2051(7 downto 0) = RX4
        end if;
  -- Test if active signal is not lost
  at t: z.active = true;

60 end property;
```

D Abbildungsverzeichnis

2.1 Gerichtete Graphen: a) Baum, b) DAG 10
2.2 Automatenmodell: a) Mealy, b) Moore, c) Vereinfacht 13
2.3 Prozessormodell [PH07] 14
2.4 Softwareschichten 19
2.5 Beispiel CFG – Graphische Darstellung eines Programms 21
2.6 Aufbau einer Simulationsumgebung [HT10] 24
2.7 BMC: Abrollen der Schaltung über die Zeit 29
2.8 Ablauf von Interval Property Checking [NTW+08] 31
2.9 Beispiel für die Erzeugung des zu prüfenden Modells bei CBMC [CKL04] 35
2.10 F-Soft: Beispiel Programm und zugehöriger CFG [IYG+08] 37
2.11 HW/SW-Modell Generierung nach [NWSK11] 38
2.12 Zerteilung der Eigenschaften in kleinere Teileigenschaften [NWSK11] . . 38

3.1 Abrollen eines CFG zu einem Execution Graph 44
3.2 Instruktionszelle: Nicht-verzweigende Instruktion 46
3.3 Instruktionszelle: Bedingter Sprung 48
3.4 Gesamtnetzliste Generierung: EXG in PN übersetzt. 49

4.1 Beispiel CFG aus Kapitel 2.4.2 57
4.2 Abstraktes kombiniertes HW/SW Modell 58
4.3 Programm Pfad FSM 58
4.4 Datenpfad-Modell 60
4.5 Ausrollen des abstrakten Modells, *IO wird nicht gezeigt* 60
4.6 Ausrollen des abstrakten Models mit bedingtem Sprung 61
4.7 Duplizierte Logik nach Verschieben der Multiplexer 62
4.8 Programmnetzliste nach dem Zusammenführen. 64
4.9 Interrupt Kommunikation mit dem Hauptprogramm 78
4.10 Beispiel: Modell für ein interrupt-getriebenes Programm 79
4.11 Instruktionszellen: a) IRQ-init cell b) IRQ-decision cell 81

5.1 Ausrollen einer Schleife 89
5.2 Verkürzen des aktiven Wartens 90

6.1 Verkürzen des aktiven Wartens 92

7.1 Sensorplattform – Kommunikationsstruktur 100
7.2 Aufbau des LIN Frames [LIN03] 102
7.3 Programmaufrufe für Ablauf eines LIN-RX-Frames 104
7.4 Programmaufrufe für Ablauf eines LIN-TX-Frames 104

B.1 Logik zum generieren des Speicherinhaltes 123
B.2 Logik zum überprüfen der Sequenz 124
B.3 Logik zum Überprüfen der Reihenfolge in einer Sequenz 125

E Tabellenverzeichnis

7.1 Seriell zu Parallel Konverter 96
7.2 Selbsttest: Übersicht über getestete Bits 99
7.3 Sensorplattform: Programmnetzlistengenerierung 100
7.4 LIN Master Knoten: Modellgenerierung 103
7.5 LIN Master Node Driver: Property Checking 107

Anhang F

Literaturverzeichnis

[20009] IEEE Standard for System Verilog – Unified Hardware Design, Specification, and Verification Language. In: *IEEE Std. 1800-2009* (2009). http://www.systemverilog.org/

[20111] Standard SystemC Language Reference Manual. In: *IEEE Std 1666-2011* (2011)

[20112] *Datasheet VLSI VS1053b MP3 decoder chip*. 2012

[Abs] ABSINT ANGEWANDTE INFORMATIK GMBH. GERMANY: *Astrée*. http://www.absint.com/astree/index_de.htm

[AEO+08] ARONS, T. ; ELSTER, E. ; OZER, S. ; SHALEV, J. ; SINGERMAN, E.: Efficient Symbolic Simulation of Low Level Software. In: *Design, Automation and Test in Europe, 2008. DATE '08*, 2008, S. 825 –830

[Ait03] AITCH, Thorn: *Aquarius: a pipelined RISC CPU*. http://opencores.org/project,aquarius. Version: 2003

[Ash01] ASHENDEN, Peter J.: *The Designer's Guide to VHDL*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001. – ISBN 1558606742

[Bar13] BARTSCH, Christian: *Erweiterung eines formalen Werkzeugs für die Hardware-Software-Coverifikation mit Hilfe vom Simulationstechniken*, TU Kaiserslautern, Diplomarbeit, 2013

[BBK09] BARTLETT, M. ; BATE, I. ; KAZAKOV, D.: Guaranteed Loop Bound Identification from Program Traces for WCET. In: *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, 2009. – ISSN 1545–3421, S. 287–294

[BBM+07] BORMANN, Jörg ; BEYER, Sven ; MAGGIORE, Adriana ; SIEGEL, Michael ; SKALBERG, Sebastian ; BLACKMORE, Tim ; BRUNO, Fabio: Complete Formal Verification of TriCore2 and Other Processors. In: *Design & Verification Conference & Exhibition (DVCon)*, 2007

[BCC+99] BIERE, A. ; CIMATTI, A. ; CLARKE, E. M. ; FUJITA, M. ; ZHU, Y.: Symbolic Model Checking Using SAT Procedures Instead of BDDs. In: *Proc. International Design Automation Conference (DAC)*, 1999, S. 317–320

[BCC+15] BERTRANE, Julien ; COUSOT, Patrick ; COUSOT, Radhia ; FERET, Jérôme ; MAUBORGNE, Laurent ; MINÉ, Antoine ; RIVAL, Xavier u. a.: Static analysis and verification of aerospace software by abstract interpretation. In: *Foundations and Trends® in Programming Languages* 2 (2015), Nr. 2-3, S. 71–190

[BH08] BABIC, Domagoj ; HU, Alan J.: Calysto: scalable and precise extended static checking. In: *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA : ACM, 2008 (ICSE '08). – ISBN 978–1–60558–079–1, S. 211–220

[BHJM07] BEYER, Dirk ; HENZINGER, Thomas A. ; JHALA, Ranjit ; MAJUMDAR, Rupak: The software model checker Blast: Applications to Software Engineering. In: *Int. J. Softw. Tools Technol. Transf.* 9 (2007), October, S. 505–525. – ISSN 1433–2779

[Bor09] BORMANN, J.: *Vollständige Verifikation*, Technische Universität Kaiserslautern, Dissertation, 2009

[BPR01] BALL, Thomas ; PODELSKI, Andreas ; RAJAMANI, Sriram K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. London, UK : Springer-Verlag, 2001 (TACAS 2001). – ISBN 3–540–41865–2, S. 268–283

[BR] BALL, T. ; RAJAMANI, S. K.: The SLAM Project: Debugging System Software via Static Analysis

[Bry86] BRYANT, Randal E.: Graph-based Algorithms for Boolean Function Manipulation. In: *IEEE Transactions on Computers* 35 (1986), August, Nr. 8, S. 677–691

[BVS+14a] BAO, Binghao ; VILLARRAGA, Carlos ; SCHMIDT, Bernard ; STOFFEL, Dominik ; KUNZ, Wolfgang: A New Property Language for the Specification of Hardware-Dependent Embedded System Software. In: *Proc. Forum on Specification And Design Languages (FDL)*. Munich, Germany, Oct 2014. – (accepted for publication)

[BVS+14b] BARTSCH, Christian ; VILLARRAGA, Carlos ; SCHMIDT, Bernard ; STOFFEL, Dominik ; KUNZ, Wolfgang: Efficient SAT/Simulation-based model generation for low-level embedded software. In: *17. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2014, S. 147–157

[BVSK17] BARTSCH, Christian ; VILLARRAGA, Carlos ; STOFFEL, Dominik S. ; KUNZ, Wolfgang: A HW/SW Cross-Layer Approach for Determining Application-Redundant Hardware Faults in Embedded Systems. In: *Journal of Electronic Testing* 33 (2017), Feb, S. 77–92

[CDE08] CADAR, Cristian ; DUNBAR, Daniel ; ENGLER, Dawson: KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2008 (OSDI'08), S. 209–224

[CGJ+03] CLARKE, Edmund ; GRUMBERG, Orna ; JHA, Somesh ; LU, Yuan ; VEITH, Helmut: Counterexample-guided abstraction refinement for symbolic model checking. In: *Journal of the ACM* 50 (2003), Nr. 5, S. 752–794. `http://dx.doi.org/http://doi.acm.org/10.1145/876638.876643`. – DOI http://doi.acm.org/10.1145/876638.876643. – ISSN 0004–5411

[CGP99] CLARKE, E. M. ; GRUMBERG, O. ; PELED, D. A.: *Model Checking*. London, England : MIT Press, 1999

[CKL04] CLARKE, Edmund ; KROENING, Daniel ; LERDA, Flavio: A Tool For Checking ANSI C Programs. In: *In Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2004, S. 168–176

[CKY03] CLARKE, Edmund ; KROENING, Daniel ; YORAV, Karen: Behavioral consistency of C and Verilog programs using Bounded Model Checking. In: *Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA : ACM, 2003 (DAC '03). – ISBN 1–58113–688–9, S. 368–371

[Cla07] CLAESSEN, Koen: A Coverage Analysis for Safety Property Lists. In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, IEEE Computer Society, 2007. – ISBN 0–7695–3023–0, S. 139–145

[CLM89] CLARKE, E. ; LONG, D. ; MCMILLAN, K.: Compositional model checking. In: *Proceedings of the Fourth Annual Symposium on Logic in computer science*. Piscataway, NJ, USA : IEEE Press, 1989. – ISBN 0–8186–1954–6, 353–362

[CS13] CADAR, Cristian ; SEN, Koushik: Symbolic Execution for Software Testing: Three Decades Later. In: *Commun. ACM* 56 (2013), Februar, Nr. 2, 82–90. http://dx.doi.org/10.1145/2408776.2408795. – DOI 10.1145/2408776.2408795. – ISSN 0001–0782

[Der] DERSHOWITZ, Nachum: *Software Horror Stories*. http://www.cs.tau.ac.il/~nachumd/horror.html

[Dil96] DILL, David L.: The Murphi Verification System. In: *Proceedings of the 8th International Conference on Computer Aided Verification*. London, UK : Springer-Verlag, 1996 (CAV '96). – ISBN 3–540–61474–5, S. 390–393

[DKW08] D'SILVA, Vijay ; KROENING, Daniel ; WEISSENBACHER, Georg: A Survey of automated techniques for formal software verification. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 27 (2008), July, Nr. 7, S. 1165–1178

[EES04] ECKER, Dr. W. ; ESEN, Volkan ; STEININGER, Thomas: Memory Models for the Formal Verification of Assembler Code Using Bounded Model Checking. In: *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.*, 2004, S. 129 –135

[ES] http://minisat.se/

[ES03] EÉN, Niklas ; SÖRENSSON, Niklas: An Extensible SAT Solver. In: *SAT*, 2003, S. 502–518

[Feh11] FEHMEL, Thomas: *Extraktion des Kontrollflussgrafen (CFG) aus Maschienencode für die Software-Verifikation*, TU Kaiserslautern, Studienarbeit, 2011

[Feh12] FEHMEL, Thomas: *Modellgenerierung von Interrupt basierten HW/SW-Systemen für die formale Verifikation*, TU Kaiserslautern, Diplomarbeit, 2012

[GKD06] GROSSE, Daniel ; KÜHNE, Ulrich ; DRECHSLER, Rolf: HW/SW co-verification of embedded systems using bounded model checking. In: *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, 2006, S. 43–48

[GLM12] GODEFROID, Patrice ; LEVIN, Michael Y. ; MOLNAR, David: SAGE: Whitebox Fuzzing for Security Testing. In: *Queue* 10 (2012), Januar, Nr. 1, 20:20–20:27. http://dx.doi.org/10.1145/2090147.2094081. – DOI 10.1145/2090147.2094081. – ISSN 1542–7730

[God05] GODEFROID, Patrice: Software Model Checking The Verisoft Approach. In: *Formal Methods in System Design, 2005* 26 (2005), S. 77 – 101

[Gri05] GRIMM, Klaus: Anforderungen an das Software-Engineering aus der Sicht der Automobilindustrie. In: *ASIM-STS-Fachgruppentreffen 2005, Simulations- und Testmethoden für Software in Fahrzeugsystemen*, 2005

[Häb11] HÄBERLEIN, Tobias: *Technische Informatik - Ein Tutorium der Maschinenprogrammierung und Rechnertechnik*. Vieweg+Teubner, 2011

[Hol97] HOLZMANN, Gerard J.: The SPIN Model Checker. In: *IEEE Transactions on Software Engineering* 23 (1997), S. 279–295

[HP00] HAVELUND, Klaus ; PRESSBURGER, Thomas: Model Checking Java Programs Using Java PathFinder. In: *International Journal on Software Tools for Technology Transfer STTT* 2 (2000), Nr. 4, S. 366–381

[HP07] HENNESSY, John L. ; PATTERSON, David A.: *Computer architecture: a quantitative approach*. Fourth Edition. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007. – ISBN 1–55860–596–7

[HR04] HUTH, Michael ; RYAN, Mark: *Logic in Computer Science - Modelling and Reasoning about Systems*. Cambridge University Press, 2004

[HT10] HAUBELT, Christian ; TEICH, Juergen: *Digitale Hardware/Software-Systeme Spezifikation und Verifikation*. Springer-Verlag Berlin Heidelberg, 2010

[HTV+13] HORN, A. ; TAUTSCHNIG, M. ; VAL, C. ; LIANG, L. ; MELHAM, T. ; GRUNDY, J. ; KROENING, D.: Formal Co-validation of Low-Level Hardware/Software Interfaces. In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*, 2013, S. 121–128

[Inf08] INFINEON TECHNOLOGIES AG: *TriCore TC1797 User's Manual, Doc v1.0*. `http://www.infineon.com/tricore`. Version: 2008

[IYG+08] IVANCIC, Franjo ; YANG, Zijiang ; GANAI, Malay K. ; GUPTA, Aarti ; ASHAR, Pranav: Efficient SAT-based bounded model checking for software verification. In: *Theor. Comput. Sci.* 404 (2008), September, Nr. 3, S. 256–274. `http://dx.doi.org/10.1016/j.tcs.2008.03.013`. – DOI 10.1016/j.tcs.2008.03.013. – ISSN 0304–3975

[JM09] JHALA, Ranjit ; MAJUMDAR, Rupak: Software model checking. In: *ACM Comput. Surv.* 41 (2009), October, S. 21:1–21:54. – ISSN 0360–0300

[KLM+15] KROENING, Daniel ; LIANG, Lihao ; MELHAM, Tom ; SCHRAMMEL, Peter ; TAUTSCHNIG, Michael: Effective verification of low-level software with nested interrupts. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015* IEEE, 2015, S. 229–234

[Kom11] KOMMEGNE, Emile: *Effiziente Generierung des Exekution Graphs*, TU Kaiserslautern, Studienarbeit, 2011

[Kri63] KRIPKE, Saul A.: Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi. In: *Mathematical Logic Quarterly* 9 (1963), Nr. 5-6, S. 67–96. – ISSN 1521–3870

[KSS+19] KAESTNER, Daniel ; SCHMIDT, Bernard ; SCHLUND, Maximilian ; MAUBORGNE, Laurent ; WILHELM, Stephan ; FERDINAND, Christian: Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software / SAE Technical Paper. 2019. – Forschungsbericht

[Lig02] LIGGESMEYER, Peter: *Software Qualität*. Spektrum Akademischer Verlag, 2002 `www.liggesmeyer.de`

[LIN02] LIN ADMINISTRATION: *LIN Specification Package Rev. 1.3*. `http://www.lin-subbus.org/`. Version: 2002

[LIN03] LIN ADMINISTRATION: *LIN Specification Package Rev. 2.0*. `http://www.lin-subbus.org/`. Version: 2003

[Lio96] LIONS, J.L.: Ariane 5 Flight 501 Failure, Report by the Inquiry Board. Paris, 1996. – Forschungsbericht

[LWS+12] LOITZ, Sacha ; WEDLER, Markus ; STOFFEL, Dominik ; BREHM, Christian ; KUNZ, Wolfgang ; WEHN, Norbert: Formal Hardware/Software Co-verification of Application Specific Instruction Set Processors. In: KAZMIERSKI, Tom J. J. (Hrsg.) ; MORAWIEC, Adam (Hrsg.): *System Specification and Design Languages* Bd. 106. Springer New York, 2012. – ISBN 978–1–4614–1427–8, S. 1–20

[McM93] MCMILLAN, K. L.: *Symbolic Model Checking*. Boston : Kluwer Academic Publishers, 1993

[MPPK17] MUKHERJEE, Rajdeep ; PURANDARE, Mitra ; POLIG, Raphael ; KROENING, Daniel: Formal Techniques for Effective Co-verification of Hardware/Software Co-designs. In: *Proceedings of the 54th Annual Design Automation Conference 2017* ACM, 2017, S. 35

[Nit09] NITZSCHE, Manfred: *Graphen fuer Einsteiger*. 3.Auflage. Vieweg+Teubner, 2009

[NTW+08] NGUYEN, Minh D. ; THALMAIER, Max ; WEDLER, Markus ; BORMANN, Jörg ; STOFFEL, Dominik ; KUNZ, Wolfgang: Unbounded Protocol Compliance Verification using Interval Property Checking with Invariants. In: *IEEE Transactions on Computer-Aided Design* 27 (2008), November, Nr. 11, S. 2068–2082

[NWSK11] NGUYEN, Minh D. ; WEDLER, Markus ; STOFFEL, Dominik ; KUNZ, Wolfgang: Formal Hardware/Software Co-Verification by Interval Property Checking with Abstraction. In: *Proceedings of the 48th Design Automation Conference*. New York, NY, USA : ACM, 2011 (DAC '11). – ISBN 978–1–4503–0636–2, S. 510–515

[One] ONESPIN SOLUTIONS GMBH: *Germany. OneSpin 360MV*. http://www.onespin-solutions.com. `http://www.onespin-solutions.com`

[PH07] PATTERSON, David A. ; HENNESSY, John L.: *Computer Organization and Design*. Third Revised Printing. Morgan Kaufmann, 2007

[PR77] P.COUSOT ; R.COUSOT: Abstract Interpretation: A Unified Lattice Model For Static Analysis Of Programs By Construction Or Approximation Of Fix Point. In: *4th ACM Symposium on Principles of Programming Languages*, 1977, S. 238–252

[PV09] PĂSĂREANU, Corina S. ; VISSER, Willem: A survey of new trends in symbolic execution for software testing and analysis. In: *Int. J. Softw. Tools Technol. Transf.* 11 (2009), Oktober, Nr. 4, S. 339–353. `http://dx.doi.org/10.1007/s10009-009-0118-1`. – DOI 10.1007/s10009–009–0118–1. – ISSN 1433–2779

[Sch10] SCHLICH, Bastian: Model checking of software for microcontrollers. In: *ACM Trans. Embed. Comput. Syst.* 9 (2010), April, Nr. 4, S. 36:1–36:27. – ISSN 1539–9087

[Sed98] SEDGEWICK, Robert: *Algorithmen*. 4. Addison-Wesley, 1998

[SNBB11] SCHLICH, Bastian ; NOLL, Thomas ; BRAUER, Jörg ; BRUTSCHY, Lucas: Reduction of interrupt handler executions for model checking embedded software. In: *Proceedings of the 5th international Haifa verification conference on Hardware and software: verification and testing*. Berlin, Heidelberg : Springer-Verlag, 2011 (HVC'09). – ISBN 978–3–642–19236–4, 5–20

[Str00] STROUSTRUP, Bjarne: *Die C++ Programmiersprache*. 4., aktualisierte Auflage, Deutsche Ausgabe der Special Edition. Addison-Wesley, 2000

[SVF+13] SCHMIDT, Bernard ; VILLARRAGA, Carlos ; FEHMEL, Thomas ; BORMANN, Jörg ; WEDLER, Markus ; NGUYEN, Minh ; STOFFEL, Dominik ; KUNZ, Wolfgang: A New Formal Verification Approach for Hardware-dependent Embedded System Software. In: *IPSJ Transactions on System LSI Design Methodology (Special Issue on ASPDAC-2013)* 6 (2013), S. 135–145

[TCL88] *Tool command language*. http://www.tcl.tk. Version: 1988

[Tec13] TECH, UBM: *2013 Embedded Market Study*. http://e.ubmelectronics.com/2013EmbeddedStudy/index.html. Version: 2013

[The] THE MATHWORKS, INC. USA: *Polyspace - Static Analysis Tools*. http://www.mathworks.com/products/polyspace/. http://www.mathworks.com/products/polyspace/

[Udu11] UDUPI, Shrinidhi: *Generation of Abstract Cells for Aquarius Architecture via GapFree Verification*, TU Kaiserslautern, Diplomarbeit, 2011

[Uni05] UNIVERSITY OF BREMEN: *uDraw(Graph)*. http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html. Version: 2005

[VHHK15] VÖRTLER, Thilo ; HÖCKNER, Benny ; HOFSTEDT, Petra ; KLOTZ, Thomas: Formal Verification of Software for the Contiki Operating System Considering Interrupts. In: *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2015 IEEE 18th International Symposium on* IEEE, 2015, S. 295–298

[Vil16] VILLARRAGA, Carlos: *Formal Verification of Firmware-Based Designs*, University of Kaiserslautern, Diss., 2016

[VSB+13] VILLARRAGA, Carlos ; SCHMIDT, Bernard ; BARTSCH, Christian ; BORMANN, Joerg ; STOFFEL, Dominik ; KUNZ, Wolfgang: An Equivalence Checker for Hardware-Dependent Software. In: *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign*, 2013, S. 119–128

[VSB+14] VILLARRAGA, Carlos ; SCHMIDT, Bernard ; BAO, Binghao ; RAMAN, Rakesh ; BARTSCH, Christian ; FEHMEL, Thomas ; STOFFEL, Dominik ; KUNZ, Wolfgang: Software in a Hardware View: New Models for HW-dependent Software in SoC Verification and Test (Invited Paper). In: *Proc. International Test Conference (ITC'14)*, 2014

[VSK18] VILLARRAGA, Carlos ; STOFFEL, Dominik ; KUNZ, Wolfgang: Software in a Hardware View. In: *Formal System Verification*. Springer, 2018, S. 123–154

[Wil09] WILHELMS, Fabian: *Entwurf einer XML-Struktur fuer die Implementierung Abstrakter Zellen am Beispiel des OpenCore Aquarius*, TU Kaiserslautern, Studienarbeit, 2009

[XA07] XIE, Yichen ; AIKEN, Alex: Saturn: A scalable framework for error detection using Boolean satisfiability. In: *ACM Trans. Program. Lang. Syst.* 29 (2007), Mai, Nr. 3. http://dx.doi.org/10.1145/1232420.1232423. – DOI 10.1145/1232420.1232423. – ISSN 0164–0925

[ZAV04] ZIADE, Haissam ; AYOUBI, Rafic ; VELAZCO, Raoul: A Survey on Fault Injection Techniques. In: *The International Arab Journal of Information Technology* 1 (2004), July, Nr. 2

Lebenslauf

Name:	Bernard Schmidt

Ausbildung

1998-2001	Studium der Technoinformatik FB Informatik, Technische Universität Kaiserslautern
2001-2007	Studium der Informationstechnik mit dem Schwerpunkt Informationsverarbeitung FB EIT, Technischen Universität Kaiserslautern Abschluss: Diplom-Ingenieur der Informationstechnik

Berufstätigkeit

2007-2014	Wissenschaftlicher Mitarbeiter am Lehrstuhl Entwurf Informationstechnischer Systeme FB EIT, Technischen Universität Kaiserslautern
seit 2014	Foschungs Ingenieur Zentrale Forschung und Vorausentwicklung Robert Bosch GmbH