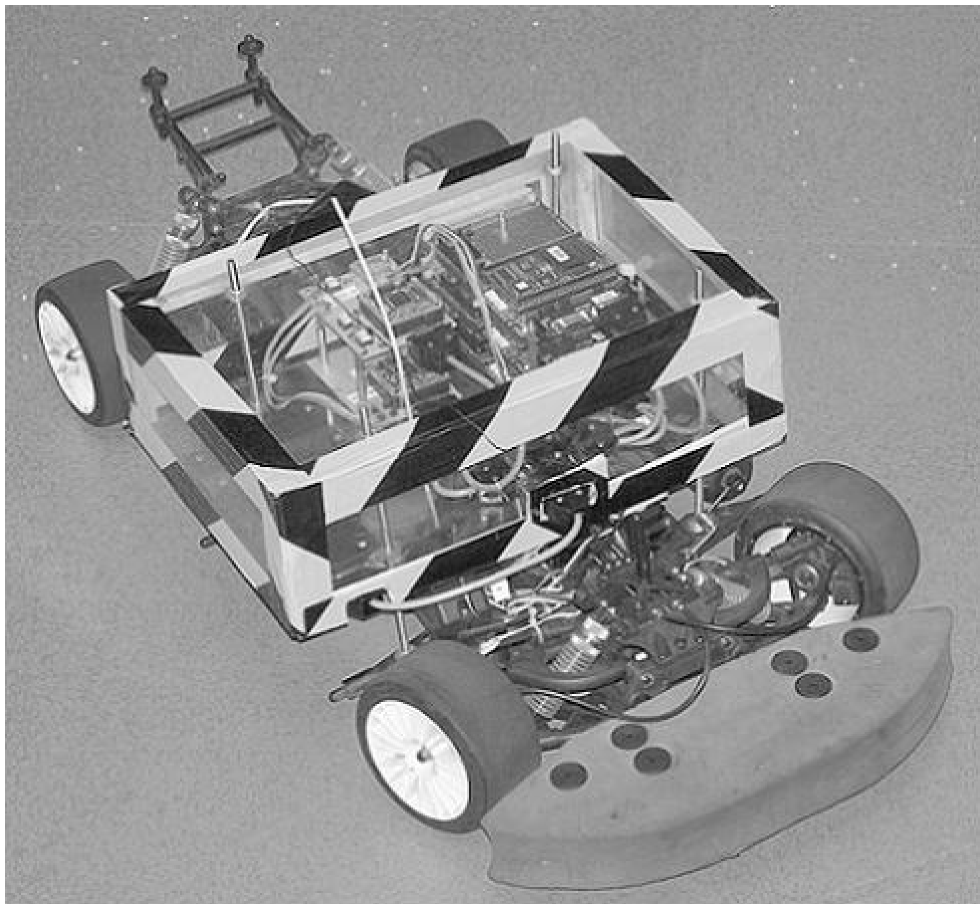
AG ROBOTERSYSTEME
FACHBEREICH INFORMATIK
AN DER UNIVERSITÄT KAISERSLAUTERN

# Diplomarbeit



# Anwendung und Evaluierung verhaltensbasierter Ansätze zur Entwicklung verlässlicher Automotivesysteme

# Frank Böhr

## 5. Juli 2006

# Anwendung und Evaluierung verhaltensbasierter Ansätze zur Entwicklung verlässlicher Automotivesysteme

Diplomarbeit

Arbeitsgruppe Robotersysteme
Fachbereich Informatik
Universität Kaiserslautern
und
Fraunhofer IESE
Department Security and Safety

Frank Böhr

**Tag der Ausgabe** : 7. Jan 2006
**Tag der Abgabe** : 5. Juli 2006

**Betreuer** : Dr. Mario Trapp
**Referent** : Prof. Dr. Karsten Berns

Ich erkläre hiermit, die vorliegende Diplomarbeit selbständig verfasst zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 5. Juli 2006

(Frank Böhr)

# Vorwort

An dieser Stelle möchte ich mich zuallererst bei Herrn Dr. Mario Trapp für die Unterstützung bei der Themenwahl, seine Themenstellung und die Betreuung dieser Diplomarbeit bedanken. Weiterhin möchte ich mich bei Prof. Dr. Karsten bedanken, der mir die externe Durchführung dieser Diplomarbeit ermöglicht hat.

Mein ganz besonderer Dank gilt aber nicht zuletzt meinen Eltern die mir das Studium überhaupt erst ermöglicht haben. Weiterhin möchte ich mich bei Hauke Kreft, Margit Cassel und Caroline Blind für das ausdauernde Korrekturlesen bedanken.

# Inhaltsverzeichnis


**1 Einleitung** **7**

1.1 Motivation . . . . . 8
1.2 Ziel der Arbeit . . . . . 8
1.3 Aufbau der Arbeit . . . . . 8

**2 Einführung in verhaltensbasierte Steuerungen** **11**

2.1 Steuerungsarten . . . . . 11
2.2 Beschreibung von Verhalten . . . . . 14
2.3 Koordination von Verhalten . . . . . 15
2.4 Architekturen . . . . . 18
2.5 Subsumptionsarchitektur . . . . . 18
2.6 State of the Art BBC . . . . . 22

**3 Dependability** **29**

3.1 Begriffsklärung . . . . . 29
3.2 Einführung in Softwarefehlertoleranz . . . . . 30
3.3 Klassische Ansätze . . . . . 30
3.4 Self-Checking-Software . . . . . 33

**4 Verhaltensbasierte Steuerungen und Software-Fault-Tolerance in Automotivesystemen** **39**

4.1 Graphische Darstellung . . . . . 39
4.2 Elemente der Architektur . . . . . 40
4.2.1 Arbiter . . . . . 41
4.2.2 Verhalten . . . . . 42
4.2.3 Safety Related System . . . . . 44
4.3 Aufbau der Architektur . . . . . 47

**5 Beispiel Architektur** **53**

**6 Aufbau des Versuchsträgers** **59**

6.1 PWM Steuersignale empfangen und senden . . . . . 59
6.2 CAN-Bus . . . . . 62

**7 Auswertung, Erfahrungen und Ergebnisse** **65**

**Literaturverzeichnis** **69**

# 1. Einleitung

Jimmy fährt mit seinem neuen PKW von der Arbeit nach Hause. Die Witterungsverhältnisse sind relativ schlecht, was ihm aber nicht bewusst ist, da die **Clima Tronic** für angenehme Temperatur sorgt und der **Regensensor** bereits die Scheibenwischer und die **automatische Scheibenreinigungsanlage** aktiviert hat und es ohnehin schon dunkel ist. Dank der **Night-Vison-Funktionalität** seines **Multi-Vision-Head-Up-Displays**, das **Nahbereichsradar** und **Infrarot-Laser-Nachtsicht** in einem System vereint, sind auch die Sichtverhältnisse kein Problem. Die Verkehrszeichen werden mittels **Verkehrszeichenanzeige** ohnehin im Cockpit angezeigt. Auch vom Fahrgefühl ist von schlechter Witterung nichts zu spüren, dank **Active Body Control** und **integralem Fahrdynamikregler**, die unter anderem **Elektronische Bremskraftverteilung, Dämpferregelung, Motormanagement (Klopfsensor, kontinuierliche Nockenwellenverstellung), ESP, Geschwindigkeitsabhängige Servolenkung, Antriebsschlupfregelung, ABS** und **Bremsassistent** ansteuern. Die **Road-Vision** meldet eine nicht vereiste Fahrbahn aus Vierkomponentenasphalt mit geringem Grip und Splittfaktor 3 an das **AdaptiveCruiseControl**, welches diese Information zusammen mit Querbeschleunigungssensor, Raddrehzahlsensor und **Reifendruck-Kontrollsystem** einsetzt, um die Parameter für **Mindestabstandsregelung (Ditronic)** und **ESP** zu regeln. Während dessen ertönen aus Jimmy's **Multimedia-MP3-Navi-Infotainmentgerät** die aktuellen Charts. Jimmy wird in seinem Musikgenuss unterbrochen, da die **AMK (Aufmerksamkeitskontrolle)** es für erforderlich hielt, ihn auf den drohenden Auffahrunfall vor sich hinzuweisen, für den das Risiko bei der gegenwärtigen Geschwindigkeit zu steigen droht. Jimmy's Versuch auf die Überholspur auszuweichen wird durch die Fanfare des in diesem Augenblick überholenden LKWs verhindert. Vor Schreck reißt er das Lenkrad nach rechts, um im nächsten Augenblick den vibrierenden **Force-Feedback-Signalen** des **Lane-departure-warning** ausgesetzt zu werden, welches ein Überfahren der durchgezogenen Straßenbegrenzungslinie zu verhindern versucht. Das **PRESAFE** System strafft in Erwartung des Aufpralls vorsorglich die Sicherheitsgurte, das **Notbremssystem** wird initiiert. Das **Abstandssystem** registriert die Gefahr des sich zu schnell nähernden Hintermanns und macht piepsend auf sich

aufmerksem. Trotz einer Vielzahl von akustischen, optischen und haptischen Warnsignalen kommt es zum Unfall.

## 1.1 Motivation

Das oben von [Kerkow 06] dargestellte Szenario zeigt, wie komplex das Zusammenspiel der Vielzahl von Komponenten eines Automobils mittlerweile ist, beziehungsweise bald sein wird. Ein typisches Mittelklassefahrzeug hat mittlerweile 100 Computerprozessoren, die miteinander kommunizieren, und bildet damit ein komplexes Softwaresystem [Rombach 06]. Die steigende Anzahl der miteinander kommunzierenden Funktionen trägt dazu bei, dass die Komplexität der Steurungssoftware immer schwerer zu beherrschen ist. Trotz eines enormen Kostenaufwands zur Entwicklung der entsprechenden Softwarekomponenten sind Softwarefehler bei Fahrzeugen der automobilen Oberklasse eine der am häufigsten angeführten Ursachen in der Pannenstatistik. Die Tatsache, dass die angeführten Funktionalitäten verteilt und parallel auf einer steigenden Anzahl unterschiedlicher Mikrokontroller abgearbeitet werden, birgt zusätzliche Probleme. Es ist jedoch Fakt, dass der Markt nach Funktionalitäten wie ABS, ASR und ESP verlangt, die zu einem wesentlichen Teil aus Software bestehen. Es ist also notwendig, einen alternativen Softwareansatz zu finden, der die Komplexität der Steuerungssoftware beherrschbar werden lässt und dennoch den Anforderungen des automobilen Umfelds, wie zum Beispiel der Verlässlichkeit gerecht wird.

## 1.2 Ziel der Arbeit

Ziel der Arbeit ist es, zu untersuchen, inwieweit sich der bisher hauptsächlich in der Robotik verwendete Ansatz der verhaltensbasierten Steuerungen (engl. Behavior-Based Control BBC) zur Lösung der angesprochenen Probleme einsetzen lässt. Aus der Vielzahl der BBC-Ansätze soll der vielversprechendste Ansatz augewählt werden. Dieser ist gegebenenfalls anzupassen und wenn notwendig, konzeptionell zu ergänzen und der Domäne entsprechend zu verfeinern. Um den dabei entstandenen Ansatz auf Schwächen, Probleme, Vorteile und Stärken untersuchen zu können, soll ein Beispielsystem implementiert werden, welches auf einem Versuchsträger eingesetzt werden kann.

## 1.3 Aufbau der Arbeit

Kapitel 2 gibt eine Einführung in die verhaltensbasierte Steuerung. Es wird auf die Entstehung der verhaltensbasierten Steuerung eingegangen. Anschließend wird auf Beschreibungs- und Koordinationsmöglichkeiten für Verhalten eingegangen und abgegrenzt, welche sich für die automobile Domäne eignen und welche nicht. Danach wird eine klassische verhaltensbasierte Architektur vorgestellt, die den Anforderungen der Domäne gerecht wird.

In Kapitel 3 werden zunächst die notwendigen Begriffe zum Thema Fault-Tolerance erklärt und eine Einführung in die herkömmlich verwendeten Mechanismen und die damit verbundenen Probleme gegeben. Als alternativer Ansatz wird Self-Checking-Software vorgestellt.

Kapitel 4 beschäftigt sich damit, wie verhaltensbasierte Steuerungen und Self-Checking-Software kombiniert werden können, gefolgt von Kapitel 5, in dem beispielhaft eine Architektur erklärt wird, die praktisch umgesetzt worden ist.

Kapiel 6 befasst sich mit dem praktischen Aufbau eines Versuchsträgers, der zum Testen der Implementierungen verwendet worden ist.

Abschließend werden die gewonnen Erfahrungen und Ergebnisse in Kapitel 7 erläutert.

# 2. Einführung in verhaltensbasierte Steuerungen

## 2.1 Steuerungsarten

Um die Bedeutung und Vorteile der verhaltensbasierten Steuerung besser verstehen zu können, werden im Folgenden zunächst einige klassische Ansätze zur Robotersteuerung kurz vorgestellt. Die klassischen Ansätze sind die *reaktive*, die *delebrative* und die *hybride Steuerung*.

Reaktive Systeme verbinden die Sensoreingaben direkt mit den Aktorausgaben. Sie erlauben so dem System eine schnelle Antwort auf eine sich ändernde unstrukturierte Umwelt. Die Inspiration und der Bezug zur Biologie entspringt dem „Stimulus - Response" Schema. Die Einschränkungen dieser Vorgehensweise schließen allerdings die Unfähigkeit des Systems ein, ein Gedächtnis oder eine interne Repräsentation der Umgebung zu haben. In rein reaktiven Modellen kann also kein Weltmodell erstellt werden und das System hat nicht die Möglichkeit, sich in einer lernenden Art und Weise auf seine Umgebung einzustellen. Reaktive Systeme sind somit für schnelle Reaktionen besser geeignet als für komplexes Schließen. In Umgebungen, die zur Entwurfszeit vollständig bekannt sind, können reaktive Systeme dennoch hoch effizient sein, und bei manchen speziellen Problemen sogar optimal. Sie kommen aber für den Einsatz im automobilen Umfeld nur eingeschränkt in Frage, da die Umgebung des Fahrzeugs zur Entwurfszeit nicht vollständig bekannt ist. Bei komplexeren Umgebungen oder Aufgaben, bei denen ein internes Weltmodell oder eine Erinnerung benötigt wird, sind die reaktiven Systeme unbefriedigend. Das Kredo dieser Steuerungsart lautet: „Denke nicht! Handele!" [Mataric b].

Im Gegensatz dazu stehen die delebrativen Systeme. In delebrativen Systemen nutzt der Roboter alle ihm zur Verfügung stehenden Sensordaten und alle intern gespeicherte Information um darauf zu schließen, welche Aktion er als nächstes ausführen wird. Schließen ist typischerweise eine Art von Planung, die sehr komplex ist. Es wird eine Sequenz von Zustand-Aktion-Folgen gesucht, wodurch sehr komplexe Berechnungen die Folge sein können, wenn der Zustandsraum groß ist. Hinzu kommt, dass

Planung ein internes Weltmodell voraussetzt, welches es dem Robotersystem erlaubt, die Ergebnisse von möglichen Aktionen vorherzusagen. Wenn genügend Information und genügend Zeit zur Verfügung stehen um einen guten Plan zu generieren, dann sind delebrative Systeme eine effiziente Möglichkeit, die gewünschte Funktionalität zu erzielen. Allerdings sind in Umgebungen mit zeit- und ortvarianten Hindernissen, wie sie für einen PKW sinnvollerweise anzunehmen sind, die Zeiten nicht immer groß genug um eine adäquate Planung garantieren zu können. Die delebrativen Steuerungsansätze sind also auf Grund ihrer eingeschränkten Echtzeitfähigkeit nur bedingt im Automotivesektor einsetzbar. Das Kredo dieser Steuerungsart lautet: „Erst genau nachdenken! Dann handeln!“ [Mataric b].

Die hybriden Kontrollsysteme versuchen die besten Eigenschaften der delebrativen und reaktiven Systeme zu vereinen. Sie versuchen den Echtzeitantworten der reaktiven Systeme gerecht zu werden, wollen aber gleichzeitig die Rationalität und Optimalität eines delebrativen Systems aufweisen. Daraus folgt, dass diese Systeme aus zwei Komponenten bestehen, nämlich der reaktiven und der delebrativen. Beide Komponenten müssen ihre Ausgaben koordinieren, damit das System ein vernünftiges Verhalten aufweisen kann. Diese Koordination ist allerdings nicht einfach, da beide Komponenten sehr unterschiedliche Aufgaben erfüllen. Die reaktive Komponente kümmert sich um die unmittelbar auftretenden „Bedürfnisse“ des Systems, wie zum Beispiel die Kollisionsvermeidung. Diese Aufgaben müssen in kurzer Zeit erfüllt werden. Im Gegensatz dazu arbeitet die delebrative Komponente mit abstrakten Modellen und löst Aufgaben, die längerfristig sein können. Daraus folgt, dass die beiden Teilsysteme miteinander kommunizieren müssen, um Synergieeffekte zu erzielen. Normalerweise wird dazu eine dritte Komponente verwendet. Das Erstellen eben dieser dritten Komponente ist in hybriden Systemen die schwierigste Aufgabe. Das Kredo dieser Steuerungsart lautet: „Denke und handele unabhängig und gleichzeitig!“ [Mataric b].

Eine mögliche Lösung für die oben genannten Probleme bieten die verhaltensbasierten Steuerungen. Verhaltensbasierte Steuerungen sind eine Erweiterung der reaktiven Ansätze. Sie fallen aber zwischen rein reaktive und rein delebrative Systeme. Reaktive und verhaltensbasierte Systeme unterscheiden sich fundamental. Obwohl verhaltensbasierte Systeme einige Eigenschaften der reaktiven Systeme aufweisen und normalerweise auch einige reaktive Komponenten haben, ist ihre Fähigkeit nicht auf das einfache direkte Verknüpfen von Eingangswerten mit Ausgangswerten beschränkt [Mataric a]. Der verhaltensbasierte Steuerungsansatz ist durch die Biologie inspiriert. Der Name dieser Steuerungsart ergibt sich aus der gewählten Repräsentationsgrundlage, dem „Verhalten“. Ein Verhalten ist ein überwachbares Aktionsmuster, welches aus der Interaktion des Roboters bzw. Automobils mit seiner Umwelt hervorgeht. Diese Systeme werden bottom-up konstruiert. Man beginnt mit einigen einfachen und grundlegenden Verhalten, bei einem Roboter zum Beispiel mit der Hindernisvermeidung, bei der ähnlich wie bei reaktiven Systemen die Sensoreingaben mit der Ausgabe verbunden werden. Im nächsten Schritt werden etwas komplexere Verhalten hinzugefügt, wie das Folgen einer Wand oder das Explorieren der Umgebung. Im Falle eines Autos könnte das erste Verhalten ein Drive-by-Wire sein und das Nächste ein ASR. Nach und nach werden mehr und mehr Verhalten hinzugefügt, bis deren Interaktion zum gewünschten Gesamtverhalten des Systems führt.

Wie in hybriden Systemen gibt es hier also mehrere Schichten. Allerdings unterscheiden sich diese Schichten nicht so sehr in der zeitlichen Skalierung und internen Repräsentation. Ein wichtiger Unterschied ist, dass die verhaltensbasierten Systeme die Repräsentation ihrer Umwelt in einer verteilten Art und Weise speichern. Falls ein solches System planen muss, dann tut es dies in einem Netzwerk von kommunizierenden Verhalten und nicht etwa in einem zentralisierten Planungsmodul. Dieser Repräsentationsunterschied hat einige Berechnungs- und Performancekonsequenzen [Mataric b]. Verhaltensbasierte Systeme können also beides, sowohl schnell reagieren wie reaktive Systeme als auch planend agieren, wie delebrative Systeme. Dazu benötigen sie keine zusätzliche Schicht wie die hybriden Systeme. Das heißt, mit der verhaltensbasierten Steuerung kann man die größten Nachteile der klassischen Ansätze umgehen. Deshalb scheinen sie besonders geeignet für den Einsatz im Automotivebereich. Auf Grund der Struktur und der Interaktion der einzelnen Verhalten entsteht aber ein emergentes Gesamtverhalten. Das bedeutet, dass alle einzelnen Verhalten zusammen mehr sind, als die Summe ihrer Teile. In der Domäne der Robotik ist dieses Phänomen durchaus wünschenswert. Im automobilen Umfled jedoch nicht, da sich das Gesamtverhalten des Systems durch die Emergenz nur schwer beziehungsweise garnicht vorhersagen lässt. Es stellt sich also die Frage wie der emergente Charakter der BBC mit den Anforderungen der automobilen Domäne in Einklang gebracht werden kann. Zusätzlich stellen sich einige weitere Fragen, wie zum Beispiel: „Wie wähle ich die Verhalten für mein System?“, „Wie kommunizieren die gewählten Verhalten miteinander?“ oder „Welche Hardwarestruktur unterstützt diesen Ansatz besonders gut?“. Das Kredo dieser Steuerungsart lautet: „Denke so, wie du handelst!“

## 2.2 Beschreibung von Verhalten

Da die Beschreibungsgrundlage für verhaltensbasierte Steuerungen, wie bereits erwähnt, die Verhalten sind, wird in diesem Kapitel erst kurz geklärt, was ein Verhalten ist. Im Anschluss werden einige Möglichkeiten zur Beschreibung von Verhalten vorgestellt.

Obwohl mittlerweile eine recht große Anzahl verhaltensbasierter Ansätze entstanden ist, gibt es keine einheitliche Definition für ein Verhalten. Das Verhalten ist unter anderem auch ein Zentralbegriff der Verhaltensbiologie. Es bezieht sich dort auf alle äußerlich wahrnehmbaren und daher auch mit technischen Hilfsmitteln erfassbaren, aktiven Veränderungen, Bewegungen, Stellungen, Körperhaltungen, Gesten und Lautäußerungen eines Menschen oder Tieres [Jones 04]. Die Definition kann ohne Probleme auf Roboter beziehungsweise technische Systeme übertragen werden. Die allgemeinste Definition besagt, dass ein Verhalten einfach eine Reaktion auf einen Stimulus ist [Arkin 98]. Ein Verhalten definiert sich also aus einem Stimulus beziehungsweise einer Sensoreingabe und einer entsprechend davon hervorgerufenen Reaktion. Ein Stimulus kann alles sein, was vom Roboter beziehungsweise Fahrzeug mit den ihm zur Verfügung stehenden Sensoren erfasst werden kann. Eine Reaktion ist eine Aktorausgabe mit oder ohne Veränderung des internen Zustandes des Systems. Um besser mit dem Begriff des Verhaltens arbeiten zu können benötigt, man eine Notation. Es gibt mehrere Möglichkeiten einer solchen Notation.

Das *Stimulus Response Diagram* (SRD) ist die einfachste und intuitivste Methode zur Beschreibung von Verhalten. Jedes Verhalten kann als eine Antwort auf einen Stimulus erzeugt werden. Abbildung 2.1 zeigt ein SRD für ein Wegfindungsproblem. Es besteht aus fünf Verhalten, deren Ausgaben in einen Koordinationsmechanismus geleitet werden. Dieser Mechanismus erzeugt dann aus den Ausgaben der einzelnen Verhalten ein angemessenes Gesamtverhalten [Arkin 98]. Diese Beschreibungsart ist jedoch sehr informell und ist dadurch in ihrer Benutzbarkeit eingeschränkt.

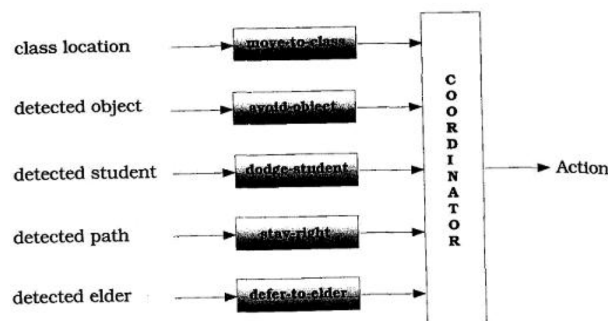



Abbildung 2.1: Darstellung eines Stimulus-response-diagramms

Die funktionale Beschreibung lässt sich im Grunde durch eine einzige Gleichung zusammenfassen: $b(s) = r$. Hierbei ist $b$ ein Verhalten und $s$ ein Stimulus, der $r$ auslöst. Man kann mit der funktionalen Beschreibung dieselben Verhalten spezifizieren wie mit einem SRD. Das Beispiel von Abbildung 2.1 würde in funktionaler Notation wie folgt aussehen.

$Coordinate-behaviors[move-to-classrom(detect-classroom-location), Avoid-objects(detect-objects), Dodge-students(detect-students), Stay-to-right-on-path(detect-path), Defer-to-elders(detect-elders)] = motor-response$

Jedes der Verhalten kann eine Ausgabe erzeugen, die dann von einer koordinierenden Funktion berücksichtigt wird. Der Vorteil der funktionalen Notation ist, dass sie relativ leicht auf eine Programmiersprache abgebildet werden kann. Der Nachteil ist, dass eine derartige Beschreibung schnell unübersichtlich wird. Daher eignet sich eine solche Beschreibung nicht für größere Systeme.

Die Verwendung endlicher Automaten hat nützliche Eigenschaften, wenn man Ansammlungen oder Sequenzen von Verhalten beschreiben möchte. Sie beschreiben explizit die benötigten Sensoreingaben und die Aktivität eines Verhaltens zu jeder Zeit. Endliche Automaten werden benutzt um komplexe Systeme zu spezifizieren, bei denen ganze Mengen von einfachen Verhalten ein- und ausgeschaltet werden, um ein Gesamtverhalten zu erzielen. Endliche Automaten stellen Mechanismen bereit, um Beziehungen zwischen Verhalten darzustellen. Das Wegfindungsproblem würde als Automat wie in Abbildung 2.2 aussehen. Der „Hauptzustand“ Journey besteht hier aus fünf Teilverhalten: move-to-classromm, Avoid-objects, Dodge-students, Stay-to-right-on-path und Defer-to-elders [Arkin 98]. Für die gewählte, und später erleuterte Architektur ist es aber nicht notwendig Mengen von Verhalten ein- bzw auszuschalten. Da die Standartbeschreibungen für Verhalten nicht geeigent erscheinen, wird in Kapitel 4.1 die verwendete Beschreibung eingeführt.

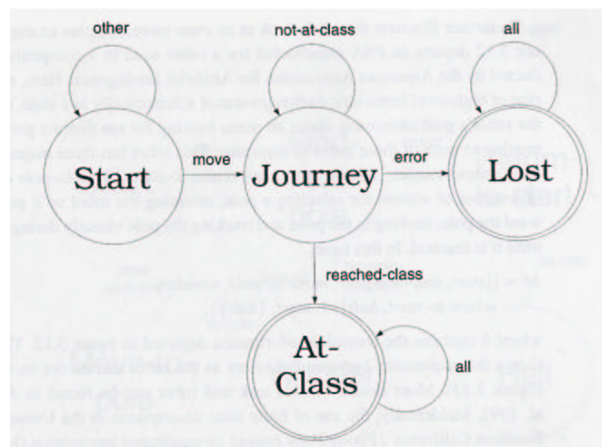



Abbildung 2.2: Ein endlicher Automat

## 2.3 Koordination von Verhalten

Zur Koordination von Verhalten gibt es zwei prinzipiell unterschiedliche Ansätze, nämlich den kompetitiven und den kooperativen [Arkin 98]. Die einfachste kompetitive Methode ist, aus der Menge der aktiven Verhalten ein beliebiges Verhalten auszuwählen, welches dann seine Ausgabe an die Aktoren senden darf. Die einfachste kooperative Methode ist die Addition aller Ausgaben, die von den Verhalten erzeugt werden. Der kompetitive Ansatz bietet eine Möglichkeit, um das Problem mehrerer konfliktionärer Verhaltensausgaben zu behandeln. Oft wird einfach das „stärkste“

Verhalten ausgewählt. Es gibt aber auch Ansätze, bei denen die Verhalten regelrecht gegeneinander kämpfen. Sie schwächen sich gegenseitig, bis nur noch ein Verhalten übrig ist, welches dann zur Ausführung kommt. Wie stark ein Verhalten das andere beeinflussen kann, hängt von der aktuellen Sensoreingabe ab. Dadurch wird die Wichtigkeit eines Verhaltens zur Laufzeit bestimmt. Das heißt, es gibt keine vordefinierte Hierarchie zwischen den Verhalten [Arkin 98]. Die kooperativen Ansätze hingegen versuchen die Verhaltensausgaben zu verschmelzen. Das Verschmelzen von Verhaltensausgaben erlaubt es, mehrere Verhalten nebenläufig zueinander zu benutzen. Das Hauptproblem dabei ist, eine Notation zu finden, die es erlaubt, die verschiedenen Ausgaben zu kombinieren. Eine nützliche Lösung für dieses Problem bieten die Potentialfelder. Wie ein durch Vektoraddition entstandenes Potentialfeld aussehen kann, zeigt Abblidung 2.3. Hier sind drei Hindernisse zu erkennen und ein Ziel. Das Verhalten, welches die Zielsuche durchführt, ist hier doppelt stark wie das Verhalten, das für das Ausweichen von Hindernissen, verantwortlich ist. Wenn man diese Gewichtung umkehrt, dann „fürchtet“ sich der Roboter mehr vor den Hindernissen und es ergibt sich ein Potentialfeld wie in 2.4. Aufgrund des stark emergenten Charakters der kooperativen Ansätze sind diese für die Anwendung im Automotiveumfeld nicht geeignet. Deshalb wurde für die hier vorgestellte Arbeit ein kompetitiver Ansatz gewählt.

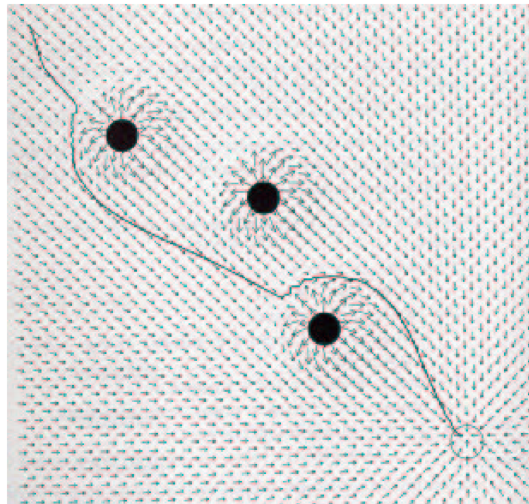


Abbildung 2.3: Potentialfeld mit starkem Zielsuchen

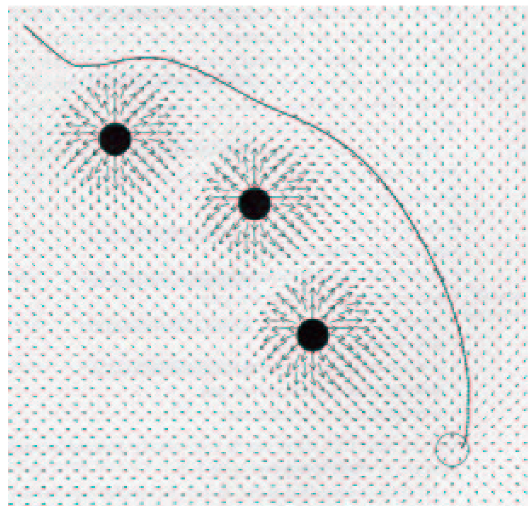


Abbildung 2.4: Potentialfeld mit starkem Hindernisausweichen

## 2.4 Architekturen

In diesem Kapitel werden zuerst die wichtigsten Anforderungen an eine Steuerungsarchitektur genannt und kurz erläutert. Danach, im folgenden Abschnitt, wird eine Architektur, die alle diese Punkte weitestgehen erfüllt, vorgestellt.

Damit eine Architektur im hier zu betrachtenden Umfeld gut einsetzbar ist, muss sie folgende Anforderungen erfüllen: Die Anforderungen sind Robustheit, Mehrsensorfähigkeit, Erweiterbarkeit und die Fähigkeit, mit mehreren, eventuell im Gegensatz zueinander stehenden Zielen arbeiten zu können. Sie muss auch damit fertig werden, dass die Wichtigkeit der Ziele von der aktuellen Situation abhängig sein kann. Es ist zum Beispiel viel wichtiger, einem bremsenden Vordermann auszuweichen als lediglich vom Wind getriebenem Laub. Konfliktionäre Verhalten könnten zum Beispiel sein, dass ein Fahrzeug in minimaler Zeit ein Ziel erreichen soll, aber dabei möglichst wenig Energie verbraucht [Brooks 85]. Die Forderung nach Mehrsensorfähigkeit ist von daher unabdingbar, das die meisten Fahrzeuge über weit mehr als nur einen Sensor verfügen. Sie müssen damit fertig werden, dass die Sensoren fehlerbehaftet sind. Oft ist es nicht möglich, aus den Sensordaten direkt auf die gewünschten physikalischen Größen zu schließen. Einige der Sensoren können sich gegenseitig überlappen und Aufschluss über die gleiche Größe geben. Da sie aber fehlerhafte Daten liefern, können sie sich gegenseitig widersprechen. Eine Systemarchitektur muss fähig sein, ausgehend von diesen Sensordaten, die richtige Entscheidung für ihr weiteres Handeln zu treffen [Brooks 85]. Eine einsetzbare Systemarchitektur muss außerdem robust sein. Robustheit bezeichnet die Fähigkeit, mit unerwarteten Eingaben zurechtzukommen. Wenn ein oder mehrere Sensoren ausfallen, sollte das System trotzdem weiter funktionieren und nicht komplett ausfallen. Unerwartete Sensoreingaben können auch durch unerwartete Umweltereignisse auftreten [Brooks 85]. Da die Umgebung, in der ein Fahrzeug später eingesetzt wird, nicht vollständig bekannt ist, kann dieser Fall relativ häufig eintreten. Eine weitere wichtige Eigenschaft einer Architektur ist ihre Erweiterbarkeit, was bedeutet, wie einfach neue Funktionalitäten in ein bereits bestehendes System eingebunden werden können. Ein System sollte leicht erweiterbar sein. Eine Architektur sollte es also erlauben, dem Fahrzeug neue Funktionalitäten hinzuzufügen, ohne es komplett neu aufbauen zu müssen. Die Architektur ist umso besser, je weniger Aufwand betrieben werden muss, um eine solche neue Funktion hinzuzufügen [Brooks 85].

## 2.5 Subsumptionsarchitektur

Eine Architektur, welche die oben genannten Anforderungen weitestgehend abdeckt, ist die Subsumptionsarchitektur. Der wesentliche Unterschied, aus dem die meisten Vorteile dieser Architektur hervorgehen, ist die Art und Weise, in der die Aufgaben des Systems zerlegt werden. Der herkömmliche Ansatz ist, das System in eine Reihe nacheinander auszuführender Funktionsblöck zu zerlegen(Abbildung 2.5 oben). Zuerst werden alle Schichten entwickelt und dann zu einem Gesamtsystem zusammengesetzt. Der bei der Subsumptionsarchitektur gewählte Ansatz ist, das System in parallel arbeitende, zielerfüllende Verhalten aufzuteilen, die nacheinander entwickelt und integriert werden (Abbildung 2.5 unten). Dies hat einige Auswirkungen auf Eigenschaften wie Robustheit und Testbarkeit [Brooks 85].

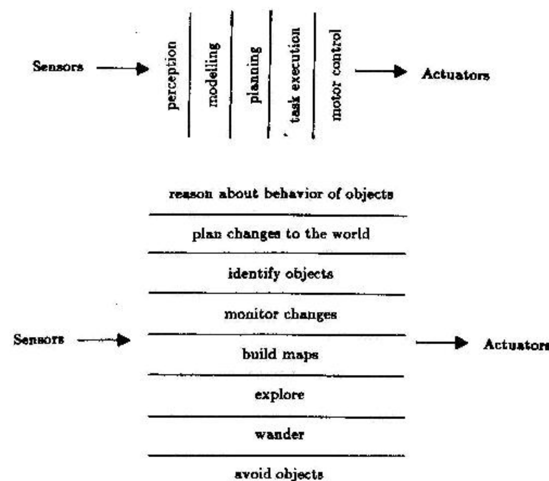



Abbildung 2.5: Oben: die Darstellung des herkömmlichen Datenflusses. Unten: der Datenfluss in der SUBSUMPTIONS-ARCHITEKTUR

Beim Design der Schichten sollte man bedenken, dass komplexes Verhalten nicht immer das Resultat eines komplexen Kontrollsystems sein muss. Man sollte versuchen, die Schichten so einfach wie möglich zu gestalten. Zum Bilden der Schichten wird das extern sichtbare Verhalten des Systems unterteilt. Man kann diese Schichten als informelle Spezifikation der zu implementierenden Verhalten sehen. Diese Schichten werden von einfacheren zu komplexeren Verhalten durchnummeriert. Jede der Schichten enthält ihre vorherige als einen Teil des von ihr dargestellten Verhaltenes. Man könnte die Schichten zum Beispiel wie folgt wählen und nummerieren:

1. Objektkontakte vermeiden
2. Ziellos umherwandern, ohne ein Hindernis zu berühren
3. Umgebung erforschen durch gezieltes Anfahren von ausgewählten Punkten
4. Landkartenbildung der Umgebung und Routenplanen um von einem Punkt zum nächsten zu gelangen
5. Änderungen in der statischen Umgebung feststellen
6. Erkenne Objekte und handele dementsprechend
7. Erstelle Pläne, welche die Umwelt in einer wünschenswerten Weise verändern, und führe diese durch.

Die Idee, die hinter dieser Vorgehensweise steckt, ist, dass man, wenn die vorherigen Schichten funktionieren, einfach eine neue Schicht hinzufügen kann. Man kann also auf der bisherigen Struktur aufbauen und das nächstkomplexere Verhalten davon ausgehend in Angriff nehmen. Wenn man ein System entwickelt hat, das die Funktionalität der ersten Schicht erfüllt (Objektkontakte vermeiden), dann kann man, ohne

dieses System zu verändern, darauf aufsetzen und eine neue Schicht hinzufügen. Diese neue Schicht kann dann Daten von Schicht 1 lesen und auch in deren normalen Datenfluss eingreifen. Die neue Schicht und die ursprüngliche erfüllen dann das nächst komplexere Verhalten. In diesem Fall das ziellose Umherwandern. Die untere Schicht arbeitet einfach weiter. Sie weiß sozusagen nichts von der ihr übergeordneten Schicht. Um komplexere Verhalten zu erreichen, wird dieser Vorgang solange wiederholt, bis das angestrebte Verhalten erreicht wird (Abbildung 2.6). Ein solche Architektur nennt man *Subsumptionsarchitektur*, da die unteren Verhalten von den oberen zusammengefasst werden.

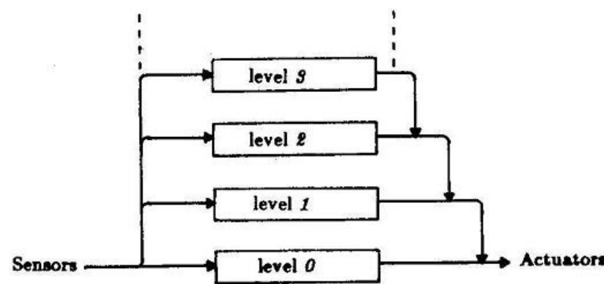



Abbildung 2.6: Mehrere Schichten eines Systems

Die Kommunikation zwischen den einzelnen Schichten stellt man sich am besten wie in Abbildung 2.7 vor. Jede Schicht hat Eingänge und Ausgänge. Eingänge müssen nicht unbedingt Sensoren sein, sondern können auch Ausgänge anderer, darunter liegender Module sein. Daraus folgt, dass die Ausgänge von Modulen auch an die Eingänge von darüber liegenden Schichten angeschlossen werden können. Eine besondere Funktionalität hat der Anschluss zum Unterdrücken der Ausgänge (Inhibitor). Wenn ein Modul über ihn eine Nachricht erhält, dann gehen alle Ausgaben für eine vorbestimmte Zeit verloren. Eine ähnliche Funktion gibt es auch für die Eingänge (Suppressor). Hier können allerdings nicht nur Werte unterdrückt werden, sondern auch ausgewählte Werte vorgegeben werden. Wie lange das Überschreiben der Werte andauert, ist unter anderem durch Zeitkonstanten bestimmt, die in den Kreisen in Abb. 2.7 zu sehen sind. Zusätzlich verfügt jedes Verhalten über einen Reseteingang [Brooks 85].

Da die Subsumption Architektur die in Kapitel 2.4 genannten Anforderungen erfüllt kann sie als Softwarearchitektur verwendet werden. Zusätzlich ist durch den schichtenartigen Aufbau das Emergente Verhalten leichter zu verstehen, da anders als beispielsweise in einem Netz nicht jedes Verhalten direkt mit jedem Anderen kommunizieren darf. Die direkte Kommunikation erfolgt nur von oben nach unten. Deshalb wurde im weiteren Verlauf der Arbeit eine Abwandelung der Subsumption Architektur verwendet.

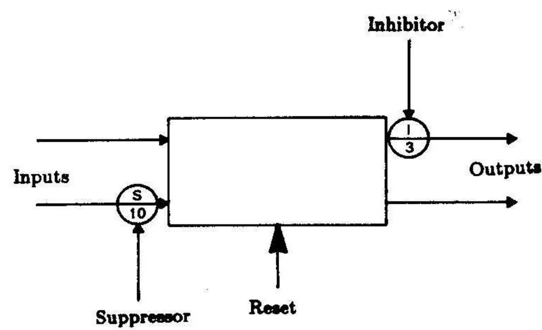



Abbildung 2.7: Darstellung einer SCHICHT, der von Rodney A. Brooks entwickelten Subsumption Architektur.

## 2.6 State of the Art BBC

Um einen besseren Überblick über das mittlerweile weite Feld der BBC Ansätze zu geben, werden im Folgenden einige weitere, sich im Ansatz stark von der Subsumption Architektur unterscheidende, Ansätze aus dem Bereich der BBC vorgestellt.

Wie bereits erwähnt, lassen sich die verhaltensbasierten Ansätze in zwei Hauptgruppen unterteilen: In die kooperative und die kompetitive. Auf Grund der Sicherheitsanforderungen im Automobilbereich können die kooperativen Ansätze bereits im Vorfeld für diesen Anwendungsbereich ausgeschlossen werden, da sie eine zu starke Emergenz ihrer Verhalten aufweisen. Deshalb wird im Folgenden nur auf die kompetitiven Ansätze eingegangen. Diese lassen sich wiederum in drei Gruppen unterteilen: In „Priotity-based“- „State-based“- und „Winner-take-all-Verfahren“.

Prioritätenbasierte Mechanismen zur Auswahl von Verhalten sind am einfachsten zu verstehen. Sie legen auf Grund bereits vor der Laufzeit festgelegter Prioritäten die Wichtigkeit der einzelnen Verhalten fest. Oft werden diese Prioritäten je nach Situation mit Gewichtungen verrechnet um eine bessere Anpassung an die aktuelle Umgebung des Roboters zu gewährleisten. Das Verhalten, welches nach Gewichtung der Prioritäten in der aktuellen Situation am wichtigsten erscheint, erhält dann die Kontrolle über die Aktoren [Arkin 98]. Das bekannteste Beispiel für einen solchen Ansatz ist die Subsumptionsarchitektur [Brooks 85], die bereits in Abschnitt 2.5 behandelt wurde. Dieser Ansatz ist bei prioritätenbasierten Mechanismen immer noch der häufigste. An der Purdue Universität wurde die Architektur auf einem Outdoorfahrzeug eingesetzt, das in Abbildung 2.8 zu sehen ist. Auf Grund des Aufbaus der Subsumptionsarchitektur und des prioritätenbasierten Charakters ist dieser Ansatz für den weiteren Verlauf der Arbeit gewählt worden.

Abbildung 2.8: Ein mit der Subsumptionsarchitektur ausgestattetes Fahrzeug der Purdue Universität

Zustandsbasierte Systeme hingegen wählen ein Verhalten oder eine Menge von Verhalten aus. Welches Verhalten aktiv ist, wird mit Hilfe eines endlichen Automaten (engl. Finite State Automata FSA) entschieden. Ein Zustand entspricht dabei einem Verhalten beziehungsweise einer Menge von Verhalten. Die Zustandsübergänge werden durch Events ausgelöst. Ein Event kann entweder eine Sensoreingabe oder eine Ausgabe eines bestimmten Verhaltens sein. Ein Framework zum Erstellen eines

solchen Automaten heißt „Discrete Event System“ [Kosecka 93]. Dieses Framework arbeitet mit zwei FSA's. Der erste endliche Automat wird „plant“ genannt und seine Aufgabe ist es, die Interaktion des Roboters mit der Umgebung zu spezifizieren. Der zweite Automat wird „supervisor“ genannt. Er überwacht den Ersten, um korrektes Verhalten mit Hilfe von „constraints“ zu erzielen. Diese Aufgabe erfüllt der Supervisor, indem er die Events, die für die Zustandsübergänge verantwortlich sind, einschränkt oder erweitert. Eine Implementierung wurde auf den Fahrzeugen in Abb. 2.9 umgesetzt. Wie schon in Kapitel 2.2 gesagt eignen sich Automaten vorallem zur Behandlung ganzer Verhaltensgruppen. Da in dieser Arbeit aber keine Verhaltensgruppen entstehen, werden keine Automaten, und somit auch der Ansatz nicht verwendet.

Abbildung 2.9: Zwei Fahrzeuge der Universität Berkley, auf denen das „Discrete Event System“ getestet wurde

Bei Winner-take-all Mechanismen resultiert die Verhaltensauswahl aus der Interaktion einer Menge von Verhalten. Die Verhalten stehen miteinander im Wettstreit bis ein Verhalten als Gewinner feststeht. Dadurch erhält es die Kontrolle über die Aktoren. Der mit Abstand am meisten verwendete Winner-take-all Ansatz wurde an der Carnegie Mellon University entwickelt. Er heißt „Distributed Architecture for Mobile Navigation“ (DAMN). Jedes Verhalten gibt hier eine Stimme für oder gegen jede mögliche Aktion des Roboters ab. Die Menge der möglichen Aktionen muss also vor der Laufzeit festgelegt werden. Die Stimmen geben wieder, welche Aktion von welchem Verhalten im aktuellen Zustand bevorzugt wird. Die Stimmen werden jedoch noch von einem sogenannten „Modemanger“ gewichtet. Über die Gewichte kann der Einfluss eines Verhaltens, wenn nötig, angepasst werden. Welches Verhalten letztendlich der Gewinner ist, wird dann vom Arbiter entschieden. Er erklärt das Verhalten mit der größten gewichteten Summe zum Sieger. Einen Überblick über die Architektur bietet Abbildung 2.10. Auf Grund des starken Verteilungscharakters ist DAMN robust gegen viele Ausfälle. Allein der Arbiter stellt eine kritische Komponente dar [Rosenblatt 97]. DAMN wurde erfolgreich auf In- und Outdoorfahrzeugen des CMU Navlab Abbildung 2.11 und des Hughes Research Labs eingesetzt. Da die

möglichen Aktionen vor der Laufzeit festgelegt werden müssen ist dieser Ansatz für das Anwednungsfeld dieser Arbeit nicht einsetzbar.

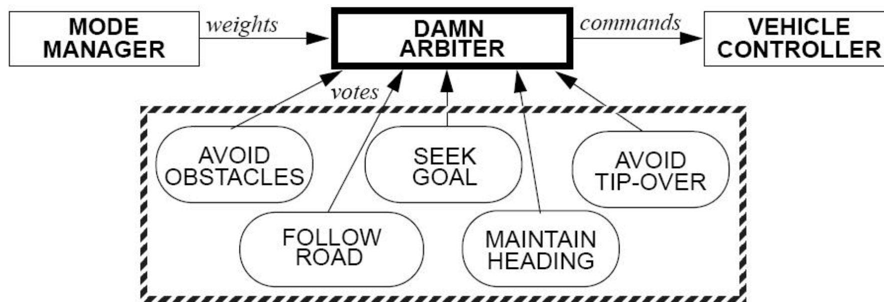



Abbildung 2.10: mehrere Verhalten, von denen das „beste“ mit Hilfe des Modemanager und des Arbiter ausgewählt wird

Abbildung 2.11: Fahrzeug das mit DAMN erfolgreich getstet wurde

Ein wesentlich stärker biologisch motivierter Ansatz wird von [Correia 95] vorgeschlagen. Bei diesem Ansatz wurde jedes Verhalten in zwei Submodule unterteilt. Das eine Modul berechnet die vom Verhalten gewünschte Aktion, also die Aktorausgabe. Das andere Modul berechnet die Aktivität des Verhaltens, also die momentane Wichtigkeit eines Verhaltens aus seiner eigenen Sicht. Beide Werte werden an den Arbitermechanismus weitergegeben. Um die relative Wichtigkeit eines Verhaltens bereits im Vorfeld beeinfussen zu können, wird jede Aktivitätsausgabe eines Verhaltens mit einem festen Wert (fixed Priority) multipliziert. Durch diese Faktoren kann der Charakter des Fahrzeugs beeinflusst werden. Der Einfluss aus der Biologie zeigt sich beim Einsatz eines „Fatigue signals“, um das „Fatigue phenomen“ nachzubilden. Dieses Phänomen drückt sich dadurch aus, dass bei einem dauerhaft vorhandenen Reiz ein bestimmtes Verhalten eines Tiers immer weniger wird, bis es schließlich verschwindet. Damit das entsprechende Verhalten jetzt wieder aktiv werden kann, muss eine definierte Zeit vergehen. Im hier beschriebenen Modell wird sie als „Recovery

Time“ bezeichnet. Eine graphische Ansicht eines Verhaltensmodul zeigt Abbildung 2.12.

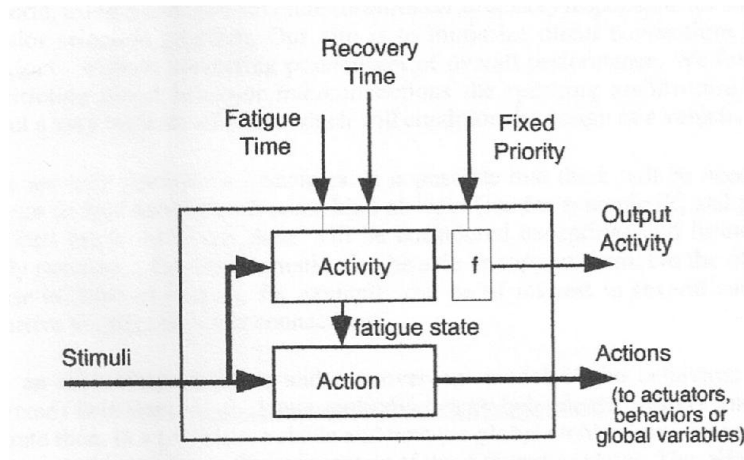



Abbildung 2.12: Verhaltensschema mit zwei Submodulen

Der Arbiter ist in diesem Modell verteilt und setzt das aus der Biologie bekannte Phänomen der Übersprungshandlung um. Jede „Zelle“ des Arbiters erhält als Eingabe die Ausgaben eines Verhaltens (Activity und Action) und die Ausgabe einer vorherigen Arbiterzelle (Ausnahme ist die erste Zelle, die als Eingabe die Ausgaben von zwei Verhalten erhält). Eine Arbiterzelle erhält also zwei Paare aus je einem Activity- und einem Action-Signal. Das Signalpaar mit der größeren Aktivität wird dann von der Arbiterzelle weitergereicht, entweder zur nächsten Arbiterzelle oder zu einem Aktor. Falls die Aktivitätssignale jedoch nur um einen bestimmten Wert H, der hier Hysterese genannt wird, variiert, wird keines der Signale weitergeleitet, was dazu führt, dass das nächstwichtigere Verhalten zur Ausführng kommt. Dabei gibt es für jeden Aktor eine eigene Arbiterstruktur. Das heißt, dass ein Verhalten an einem Aktor die Kontrolle gewinnen kann und zur gleichen Zeit an einem anderen nicht! Diese Tatsache schließt den Ansatz für die Anwendung im Kfz aus. Eine graphische Darstellung für eine Arbiterzelle zeigt Abbildung 2.13 und einen Überblick über die Gesamtstruktur zeigt Abbildung 2.14. Bisher wurde die Architektur jedoch nur auf kleinen Indoorfahrzeugen getestet. Ein Prototyp, auf dem ein solches System umgesetzt wurde, ist in Abbildung 2.15 zu sehen.

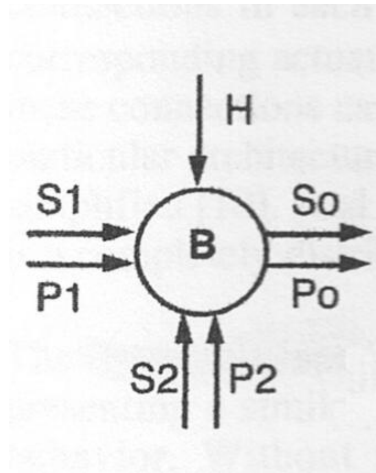



Abbildung 2.13: schematische Dartellung einer Arbiterzelle

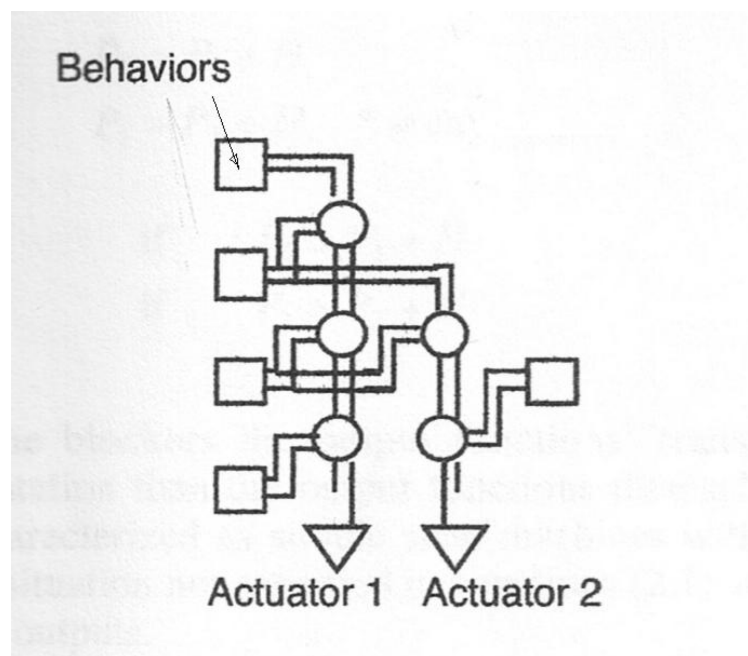



Abbildung 2.14: Übersicht über die Gesamtstruktur

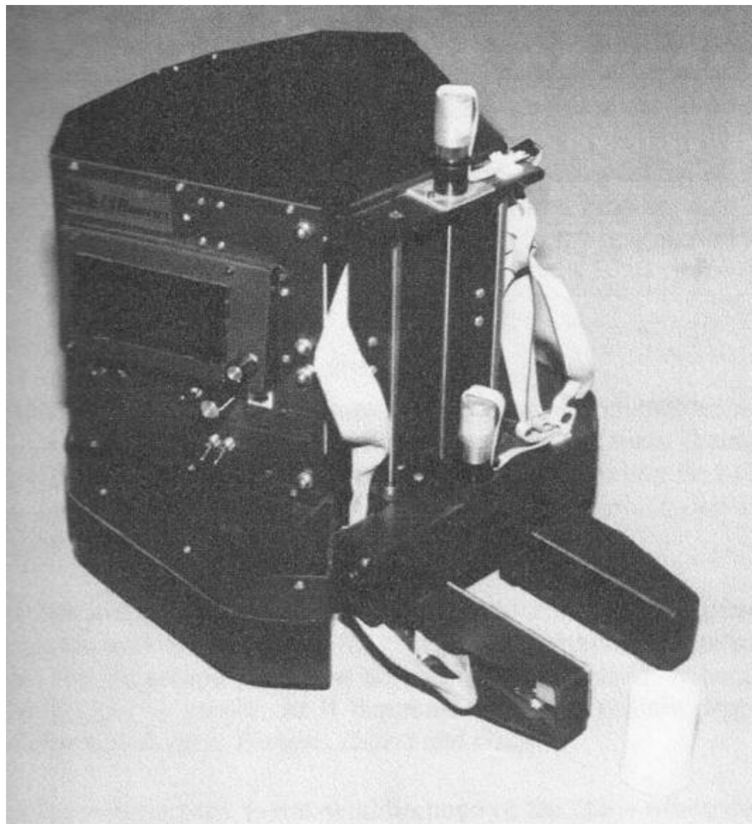


Abbildung 2.15: Prototyp, auf dem die Architektur eingesetzt wurde

# 3. Dependability

Ein wichtiger Entwicklungsaspekt für Software die in einem Automobil eingesetzt werden soll, ist *Dependability*. Da dieser Aspekt aber bei den meißten der BBC Ansätze, für den Einsatz im automotive Sektor nicht die notwendige Ausprägung zeigt, wird er im Folgenden gesondert behandelt, um einen BBC Ansatz mit den notwendigen Konstrukten ergänzen zu können.

Dependability besteht aus mehreren Aspekten und bedeutet sinngemäß übersetzt „Verlässlichkeit". Man kann dependability aufteilen in *availability*, *reliability*, *confidentiality*, *integrity*, *maintainability*, *security*, *safety* [N. Schneidewind 98], *robustness* und *survivability*. Für das automobile Umfeld ist der Teilaspekt der Safety von großer Wichtigkeit. Deshalb wird dieser Teilaspekt der Dependability ausgewählt und im weiteren Verlauf genauer behandelt. Safety beschreibt die Fähigkeit des Systems, alle bekannten und als gefährlich eingestuften Situationen zu vermeiden [Barbacci95 ]. Um eine Verbesserung der Safetyeigenschaften eines Systems und damit auch der Dependability zu erreichen, können auch Softwarefehlertoleranz-Mechanismen zum Einsatz kommen. Komplexe safety-kritische Systeme, wie zum Beispiel ein Mittelklasse KFZ, sind häufig multi-disziplinär. Ein Teil dieser Systeme ist meist ein Computerkontrollsystem. Um sicherzustellen, dass diese Gesamtsysteme mit hoher Dependability arbeiten, ist es also nötig, eben dieses Kontrollsystem fehlertolerant zu gestalten. Das gilt sowohl für Hard- als auch für Software. Klassische Ansätze um Softwarefehlertoleranz zu erreichen, sind zum Beispiel *Recovery Blocks* und *N-Version-Programming* [Software Fault Tolerance 98]. In den folgenden Abschnitten wird auf die verwendeten Begriffe, dann auf Softwarefehlertoleranz, sowie die damit verbundenen Konzepte eingegangen.

## 3.1 Begriffsklärung

Um die Methoden und Ansätze, die gewählt wurden um Softwarefehlertoleranz zu erreichen, zu verstehen, wird im Folgenden kurz auf die Entstehung, die Eigenschaften und die möglichen Folgen eines Faults eingegangen.

Den Lebenszyklus einer Software kann man im Wesentlichen in zwei Teile aufteilen. Während der ersten Phase, der Entwicklungsphase, können Änderungen am System

vorgenommen werden. Die zweite Phase ist die Runtimephase, in der das System ausgeführt wird und keine weiteren Änderungen vorgenommen werden können. Die Grundlage für das Entstehen eines *Errors* ist ein *Fault*. Ein Fault ist ein Fehler im Code, der auf den Programmierer zurückzuführen ist. Ein Beispiel für einen Fault ist die falsche Interpretation eines Requirements oder ein falscher Arrayindex. Faults entstehen in der Entwicklungsphase der Software. Ein großes System kann nicht als faultfrei angesehen werden, egal wie sorgfältig es implementiert und desigend wurde [s. s. Yau ]. Wenn dieser Fault während des Testens nicht gefunden wird, dann kann er später (zur Laufzeit) die Ursache für ein Error sein. Dies passiert immer dann, wenn die Systemkomponente, in welcher der Fault enthalten ist, zur Ausführung kommt und dadurch diese Komponente nicht wie spezifiziert funktioniert. Wenn jetzt keine Maßnahmen ergriffen werden um zu verhindern, dass das Fehlverhalten dieser Komponente sich auf das Verhalten des Gesamtsystems negativ auswirkt, wird das Verhalten des Gesamtsystems nicht mehr der Spezifikation entsprechen. In diesem Fall spricht man von einem *Failer*. Software, die trotz eines Faults keinen Failer aufweist, nennt man fault-tolerant. [Safety and relyability engineering 06] Die deutschen Begriffe für Fault, Error und Failer sind Fehler, Fehlzustand bzw. Fehlverhalten.

## 3.2 Einführung in Softwarefehlertoleranz

Softwarefehlertoleranz ist die Fähigkeit einer Software, Errors, die entstehen oder die bereits entstanden sind, zu entdecken und mit ihnen so umzugehen, dass das System weiterhin wie spezifiziert arbeitet. Softwarefehlertoleranz ist eine notwendige Designkomponente, um einem hohen Anspruch an Safety und Reliability in Embedded Systems gerecht zu werden. Dabei ist Fehlertoleranz alleine aber nicht ausreichend, sondern sie ist ein weiterer Schritt hin zu besseren Systemen. Um Probleme, die beim Design von fehlertoleranten Systemen auftreten, besser verstehen zu können, ist es nötig, die Fehler, die damit behoben werden sollen, genauer zu betrachten. Software Faults sind Design-Faults. Die Softwareherstellung, also deren Produktion, ist nahezu fehlerfrei. Da die auftretenden Fehler alle durch Design Faults entstehen, unterscheidet sich Software signifikant von allen anderen Bereichen (z.B. Hardware), in denen Fehlertoleranz bisher erwünscht war und ist. Die Tatsache, dass Software Faults das Ergebnis von menschlichen Fehlern beim Implementieren von Algorithmen oder dem Interpretieren einer Spezifikation sind, muss berücksichtigt werden. Bisherige Softwarefehlertoleranzmethoden basieren aber auf traditionellen Softwarefehlertoleranzansätzen. Das Problem dabei ist, dass diese Ansätze hauptsächlich dazu entwickelt wurden, um Produktionsfehler abzufangen. Design Diversity war kein Konzept um Hardwarefehlertoleranz zu erreichen, deshalb konnten N-way redundante Systeme zwar viele, aber nicht alle Faults abfangen. Software-Fault-Tolerace Systeme versuchen, sich die Erfahrungen, die mit Hardwarefehlertoleranz gesammelt wurden, zu Nutze zu machen. Dadurch entsteht allerdings die Notwendigkeit nach Design Diversity, da die Produktion von Software ja nahezu fehlerfrei ist. [Software Fault Tolerance 98]

## 3.3 Klassische Ansätze

Es gibt zwei klassische Softwarefehlertoleranz Ansätze. Zum einen N-Version-Software und zum Anderen Recovery-Blocks. Die Recovery-Block Methode ist eine konzeptio-

nell relativ einfache Methode. Dieser Ansatz arbeitet mit einem Adjudicator, der die Ergebnisse mehrerer Blöcke, die semantisch dieselbe Funktionalität implementiern, verarbeitet. In einem System mit Recovery-Blocks wird das System in Blöcke zerteilt, bei deren Ausfall es möglich ist, das System zu retten. Das ganze System ist aus solchen fehlertoleranten Blöcken aufgebaut. Jeder dieser Blöcke enthält wenigstens einen *primary-case-*, *secondary-case-* und *exceptional-case-Code* und den *Adjunctor*. Die primary-case-, secondary-case- und excepional-case-Blöcke können wieder nach demselben Schema aufgeteilt sein. Der Adjunctor stellt fest, welcher Block das korrekte Ergebnis liefert. Es sollte versucht werden, den Adjunctor möglichst einfach zu halten um eine korrekte Implementierung zu erleichtern. Beim ersten Eintreten in einen Block ruft der Adjunctor den primary-case-code auf. Falls der Adjunctor nun feststellt, dass das primary-case-code Fragment einen Fault aufweist, versucht er, alle Auswirkungen des primary-case rückgängig zu machen und ruft den secondary-case code auf. Falls der Adjunctor keine der Alternativen zulässt, ruft er den exceptional-case code auf, der dann mitteilt, dass der Block seine angeforderte Aufgabe nicht erfüllen kann. Eine schematische Darstellung ist in Abbildung 3.1 zu sehen. Die Recovery-Block-Methode verlässt sich auf das für Softwarefehlertoleranz notwendiges Prinzip der Design diversity. Daher muss es, um diese Methode anwenden zu können, möglich sein, die Anforderungen spezifisch und gleichzeitig „ungenau“ genug zu gestalten, dass am Ende möglichst verschiedene Codefragmente entstehen, die aber semantisch gleich sind. Dies ist keine triviale Aufgabe. Die Recovery Block Methode wird dadurch kompliziert, dass sie die Möglichkeit für Rollbacks benötigt. Die Fähigkeit von Try and Rollback hat den Effekt, dass solche Systeme sehr stark vom Transaktionskonzept beeinflusst sind. Bei Systemen, die das Transaktionskonzept verwirklichen, wird eine Aktion nur an das System weitergegeben, wenn eine komplette Transaktion erfolgreich war. Dies hat Vor- und auch Nachteile. Der größte Vorteil ist es wohl, dass ein System, das mit Transaktionen arbeitet, nur schwer in einen inkorrekten oder instabilen Zustand gebracht werden kann. [Lyu 95]

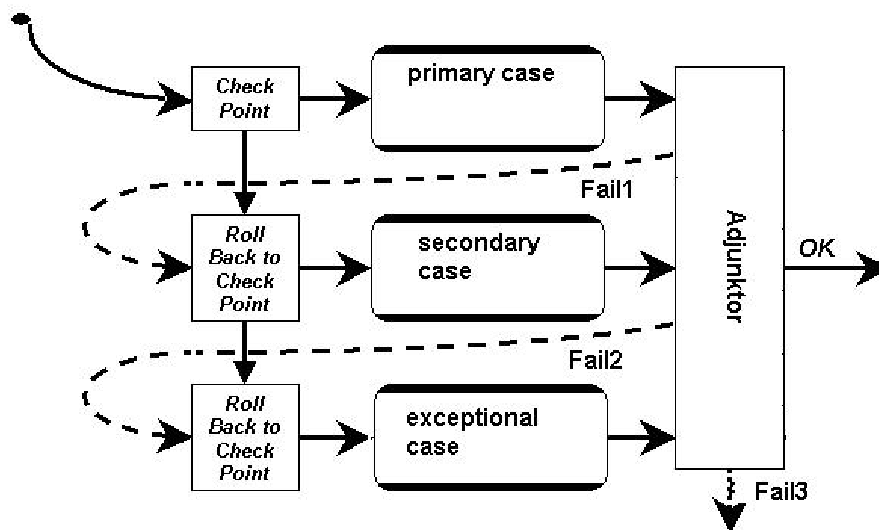



Abbildung 3.1: Das Recovery-Block Model

Das N-version-Software-Konzept hingegen versucht Fehlertoleranz über denselben Weg zu erreichen wie N-way redundance bei Hardware. Bei N-Version Softwaresystemen wird jede Komponente N-Mal mit einer unterschiedlichen Implementierung hergestellt. Jede Komponente erfüllt dieselbe Aufgabe. Man hofft dabei, dass diese Aufgabe auf verschiedenen Wegen erfüllt wird. Jede Komponente gibt ihre Ausgabe an einen Voter weiter, der dann entscheidet, welches die richtige Ausgabe ist. Diese Ausgabe gilt dann als Ausgabe des gesamten Moduls. Eine schematische Darstellung ist in Abbildung 3.2 zu sehen. Dieser Ansatz kann je nach Größe von $N$ viele Software-Faults abfangen, wobei er sich auf das Konzept der Design Diversity stützt. Ein Vorteil bei N-version Software ist, dass die Teilsysteme auf unteschiedlicher Hardware laufen können. Das Ziel, das damit verfolgt wird, ist die Design-diversity weiter zu erhöhen. Um N-version Software zu benutzen, ist es notwendig, jedes Modul mit der größtmöglichen Design-diversity zu versehen. Das heißt, die Entwickler sollten unterschiedliche Tools, Programmiersprachen und Hardware benutzen. Außerdem ist der Kontakt zwischen Entwicklern verschiedener Teilsysteme zu vermeiden. Diese Maßnahmen sind wichtig, denn N-version-Software kann nur ausreichend fehlertolerant sein, wenn die Design-diversity ausreichend ausgeprägt ist. Bei der Implementierung von N-version-Software liegt eine Schwierigkeit, genau wie bei Recovery-Blocks, bei der Spezifikation der Komponenten. Es muss die notwendige Balance gefunden werden, die Spezifikation einerseits so genau zu gestalten, dass die Komponenten zusammen so eingesetzt werden können, dass der Voter unter ihnen gleich entscheiden kann, und andererseits muss die Spezifikation ausreichend „ungenau“ sein, damit die Programmierer Software diversity hervorbringen können. [Software Fault Tolerance 98]

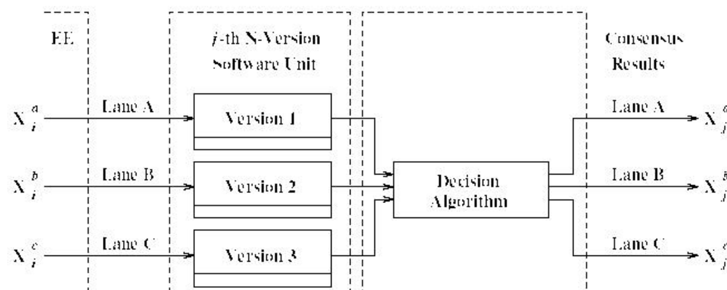



Abbildung 3.2: Das N-version Softwaremodell für N=3 [LYU]

Die Unterschiede zwischen Recovery Blocks und N-version Programming scheinen auf den ersten Blick nicht allzu groß. Sie sind aber trotzdem von Bedeutung. Beim Recovery-Block-Ansatz wird jede Alternative nacheinander getestet, bis der Adjunctor eine akzeptiert. Der N-version Ansatz ist darauf ausgelegt, N-way Hardware zu benutzen und damit alle Alternativen nebenläufig auszuführen. Durch die serielle Ausführung der Alternativen bei Recovery-Blocks entsteht eine längere Ausführungszeit. Dieser erhöhte Zeitaufwand zur Laufzeit kann bei real-time Systemen zum Problem werden. Im Gegensatz dazu benötigt der N-version Software Ansatz aber N-way

Hardware und ein Kommuniktionsnetzwerk, um diese zu verbinden. Dieser erhöhte Hardwareaufwand führt zu erhöhten Kosten [Software Fault Tolerance 98]. Beide Ansätze sind also für den Einsatz im Automobil nur bedingt geeignet: Recovery-Blocks wegen ihrer mangelnden Echtzeitfähigkeit und N-version-Software auf Grund der erhöhten Kosten.

## 3.4 Self-Checking-Software

Eine mögliche Alternative zu den eben vorgestellten Ansätzen ist Self-Checking-Software. Laut [s. s. Yau ] wird Self-checking-Software hauptsächlich für Systeme designed, in denen ultra-reliability oder Fehlertoleranz gefordert wird. Diese Systeme sind meist real-Time Systeme, wie zum Beispiel Flugkontrollsysteme oder Raketenleitsysteme. Deshalb liegt die Vermutung nahe, dass Self-Checking-Software Ansätze auch im automobilen Umfeld einsetzbar sind. Für große Systeme ist das erschöpfende Testen aller Fälle nicht möglich, da die Anzahl der Möglichkeiten zu groß ist. Es wird immer Fälle geben, die unberücksichtigt bleiben. Selbst wenn es gelingen würde, das Softwaresystem zu verifizieren, wäre es nur möglich, das korrekte Verhalten des Gesamtsystems zu garantieren, wenn man seine Integrität garantieren kann. Diese kann aber durch Hardwarefehler verletzt werden, wie zum Beispiel defektem Speicher. Einige, wenn auch nicht alle solcher Integritätsfehler und „normale" Faults können von Self-Checking-Software erkannt werden. Self-Checking-Software kann dazu verwendet werden, das korrekte Verhalten des Systems während der Laufzeit zu verifizieren [s. s. Yau ]. Ein integraler Bestandteil dieses Ansatzes ist, Softwareredundanz in das System einzufügen. Dabei werden sowohl zusätzliche Instruktionen als auch zusätzliche Daten verwendet. Self-Checking-Software kann die folgenden Aufgaben erfüllen:

1. Erkennen von Softwarefaults
2. Lokalisieren von Softwarefaults
3. Verifizieren der Systemintegrität
4. Recovery des Systems nach Faults

Die Fähigkeit, Faults während der Laufzeit zu erkennen, ist wichtig für viele real-time Systeme. In manchen Fällen würde man lieber den Prozess stoppen als eine falsche Aktion auszuführen. Man will ein Fahrzeug zum Beispiel lieber zum Stillstand bringen als fälschlicherweise den Motor mit einer zu hohen oder sogar maximalen Gasstellung zu betreiben. Im Gegensatz zu Hardware-Faults gibt es bei Software-Faults keine Statistiken über die Verteilung von Faults. Es ist auch nicht möglich, alle Fehlerzustände aufzuzählen. Deshalb kann das Erkennen von Software-Faults nur durch das Erkennen eines abnormalen Systemverhaltens geleistet werden. Dies ruft die Notwendigkeit hervor, ein normales Verhalten zu definieren. Die Self-Checking-Software kann dann eine Abweichung von diesem normalen Verhalten feststellen. Durch Überprüfung der Zumutbarkeit des Systemverhaltens an verschiedenen Programmpunkten kann die Reliayability erhöht werden und die Ausbreitung von Faults eingeschränkt werden. Die Einschränkung von Faults hilft dabei, den Schaden, den

ein solcher verursacht, zu verringern und somit das System vor einem eventuell katastrophalen Versagen, wie zum Beispiel dem Ausfallen der Bremsanlage zu bewahren. Nachdem ein Fault entdeckt wurde, kann eine Recovery Prozedur gestartet werden, um das abnormale Verhalten zu korrigieren. Ein schnelles Vorgehen der Recoverymethode ist wichtig für Systeme, die eine hohe Verfügbarkeit benötigen. Solche Systeme können so desigend werden, dass der Recoveryvorgang parallel zum normalen Verhalten abläuft. Dabei kann es aber durchaus zu einer Einschränkung der Funktionalität kommen. Grundsätzlich überprüfen Self-Checking-Software Ansätze aber immer einen oder meherere der folgenden drei Punkte:

1. Kontrollsequenz
2. Daten
3. Funktionalität des Prozesses

Deshalb wird auf jeden dieser Punkte im Folgenden kurz eingegangen.

Die Kontrollsequenz eines Programms ist die Ausführungsreihenfolge der abgearbeiteten Befehle. Jede Abweichung von der gewünschten Kontrollsequenz birgt das Potential, einen Fehlerzustand hervorzurufen. Ein Fehlerzustand, der auf diese Art entsteht, wird „controllfault“ genannt und kann in eine der drei folgenden Gruppen eingeteilt werden:

1. Eine Endlosschleife wird ausgeführt (z.B. fehlerhafte Abbruchbedingung)
2. Eine illegale Verzweigung im Code wird ausgeführt (z.B. falsche Interruptadresse)
3. Eine falsche Verzweigung im Code wird ausgeführt (z.B. fehlerhafter Vergleich)

Endlosschleifen können dadurch erkannt werden, dass im Vorfeld eine Abschätzung vorgenommen wird, wie oft eine Schleife im Maximalfall durchlaufen werden soll. Dies kann je nach Komplexität des Codes zu mehr oder weniger Aufwand führen. Eine illegale Verzweigung kann durch ein Schema entdeckt werden, welches relay-runner genannt wird. Bei diesem Vorgehen wird ein „Staffelstab“, der die Funktion eines Codewortes übernimmt, von Checkpoint zu Checkpoint weitergereicht. Wenn eine illegale Verzweigung vorgenommen wird, kann dies am nächsten Checkpoint erkannt werden, da nicht der richtige Staffelstab übergeben wird. Auf die gleiche Weise erkennt man falsche Verzweigungen. Anhand des falschen Staffelstabs kann erkannt werden, von wo aus die falsche Verzweigung gestartet wurde. Ein Beispiel für das Überprüfen einer If-Verzweigung mit relay-runner ist in Abbildung 3.3 zu sehen.

Mit Self-Checking-Software können aber auch Daten überprüft werden. Normalerweise versteht man unter Daten den Teil der Software, der von einem System verarbeitet wird. Man kann den Begriff aber auch etwas weiter fassen und jegliche Information die in einem System gespeichert ist, als Daten auffassen. Dazu gehört dann auch der Code, aus dem das System selbst besteht. Diese Erweiterung ist von

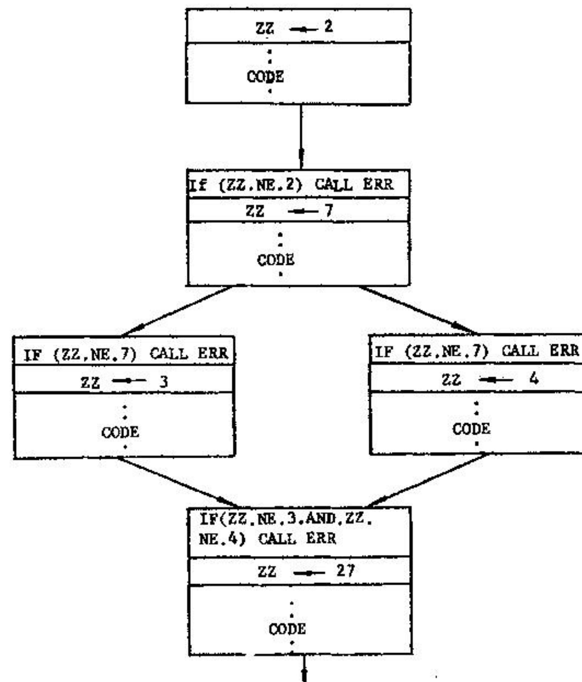



Abbildung 3.3: Ein Beispiel für die Verwendung von relay-runner bei einer If-verzweigung

daher sinnvoll, dass eine fehlerhafte Änderung sowohl von zu verarbeitenden Daten als auch des Programmcodes zu einem Fault führen kann. Eine solche fehlerhafte Veränderung kann sowohl durch Hardwarefehler als auch durch Softwarefehler entstehen. Das Überprüfen der Daten kann anhand der folgenden drei Aspekte erfolgen:

1. Integrität der Datenwerte
2. Integrität der Datenstruktur
3. Werte der Daten

Das Prüfen der Integrität der Datenwerte kann mit Hilfe einer Prüfsumme erreicht werden. In diesem Berreich gibt es mehrere etablierte Ansätze, wie zum Beispiel das Einfügen von Paritätsbits oder einer CRC-Checksum. Die Prüfbits oder Prüfsummen können entweder periodisch oder direkt vor einer Nutzung des Datensatzes eingesetzt werden. Es ist auch hilfreich eine Programmiersprache zu verwenden, die den Programmierer davor schützt, Daten zu zerstören, indem sie zum Beispiel verhindert, dass ein zu großer Arrayindex benutzt wird. Das Vorgehen, das gewählt wird, um die Integrität einer Datenstruktur zu überprüfen, hängt stark von der Datenstruktur ab. Es können auch Faults anhand der Werte von Variablen erkannt werden. Manche Werte haben eine natürliche Ober- bzw Untergenze; außerhalb dieses Bereiches hat ein Variablenwert keine Gültigkeit. Wenn eine solche Variable einen Wert außerhalb der Grenzen aufweist, kann von einem Fault ausgegangen werden. Es kann auch der Durchschnitt und die Standardabweichung wichtiger Variablen eines System überwacht werden um ein abnormales Verhalten des Systems festzustellen. Eine andere

Methode ist die Überwachung des Gradienten einer Folge von Werten. Drittens kann der funktionale Aspekt eines Programms an Hand der Angemessenheit eines Paares aus Eingabe- und Ausgabewert überprüft werden. Dies kann oft überprüft werden, wenn die Berechnung mathematisch genau definiert ist. Wenn die Ausgabe zum Beispiel die Lösung eines Gleichungssystems ist, kann durch Einsetzten die Korrektheit der Lösung überprüft werden. Wenn die Berechnung keine mathematische Funktion ist, gibt es trotzdem häufig Zusammenhänge, mit deren Hilfe sich die Korrektheit eines Resultats festellen lässt [s. s. Yau ]. Laut [Hal Wasserman ] ist es eine theoretisch erwiesene Kuriosität, dass für bestimmte Berechnungen die Zeit, die benötigt wird, um ein Ergebnis zu bestimmen, größer ist als die Zeit, die notwendig ist, um festzustellen, ob das errechnete Resultat richtig oder falsch ist. Diese Tatsache führt zu einem Ansatz, der Result-checking genannt wird. Bei diesem Ansatz wird das Paar $\prec x, y \succ$ aus Eingabe $x$ und Ausgabe $y$ an einen Checker übergeben, der anhand bestimmter Zusmmenhänge die Zusammengehörigkeit der Werte bestätigt oder widerlegt. Formalisiert wird diese Idee durch das von [Blum ] eingeführte Konzept der „Simple Checkers“. Dort wird die zu berechnende Funktionalität allgemein mit $f$ bezeichnet. Die kleinste bekannte Berechnungszeit für $f$ wird als $T(n)$ bezeichnet, wobei $n$ die Länge der Eingabe darstellt. Damit ein Algorithmus als Simple Checker bezeichnet wird, muss er die folgenden I/O Eigenschaften erfüllen:

- **Input:** I/O Paar $\prec x, y \succ$ des zu prüfenden Codes
- **Korrekte Ausgabe:** If $y = f(x) \Rightarrow$ ACCEPT, else REJECT
- **Zuverlässigkeit:** Für alle Paare $\prec x, y \succ$ muss der Checker mit einer Wahrscheinlichkeit $p_c$ das richtige Ergebnis liefern. $p_c$ soll dabei nahe bei 1 liegen.
- **„Little-o Regel“:** Der Checker darf die Komplexität von $o(T(n))$ nicht überschreiten.

Die „Little-o Regel“ führt zu Checkern mit einem gewissen Grad an Effizienz. Außerdem verhindert sie die Überprüfung durch ein erneutes Berechnen von $f$. Weiterhin hat die „Little-o Regel“ zur Folge, dass der Checker sich von der ursprünglichen Implementierung von $f$ unterscheiden muss. Dies gibt laut [Blum ] begründeten Anlass zur Hoffnung, dass die Checker wirklich unabghängig von der Software sind, die sie überprüfen sollen, und so Faults zuverlässiger erkennen können. Ein von [s. s. Yau ] und [Hal Wasserman ] vorgestelltes Beispiel für einen einfachen Resultchecker überprüft einen Sortieralgorithmus. Dieser Sortieralgorithmus erhält als Eingabe einen Integerarray $\vec{x} = (x_1, ..., x_n)$ und liefert die Ausgabe $\vec{y} = (y_1, ..., y_n)$, wobei $\vec{y}$ die Elemente aus $\vec{x}$ in sortierter Reihenfolge enthält. Es ist bekannt, dass diese Aufgabe eine Zeit von $\Omega(nlogn)$ benötigt. Wenn der Checker also auf $O(n)$ begrenzt wird, ist die „Little-o rule“ erfüllt. Ein Checker, der überprüft, ob die Elemente in $\vec{y}$ aufsteigend sind und ob alle Elemente noch vorhanden sind, kann die Anforderung der „Little-o rule“ erfüllen [Hal Wasserman ]. Der Aufbau und die im Checker enthaltene Logik sind stark abhängig von der zu überprüfenden Software. Deshalb gibt es kein allgemeines Vorgehen zum Erstellen eines Checkers. Es muss für jedes Probelem ein passender Checker erarbeitet werden. Ein ähnlich formaler Ansatz wird von [Paolo Traverso ] vorgestellt. Dort wird die Überprüfung in zwei Teile zerlegt,

in einen Online- und einen Offlineteil. Der Onlineteil läuft zeitgleich mit der zu überprüfenden Software. Jedes Berechnungsergebnis wird während der Laufzeit auf bestimmte Kriterien getestet. Der Offlineteil wird vor der eigentlichen Laufzeit des Systems ausgeführt. Der Offlineteil beweist, dass die Kriterien, die der Onlineteil überprüft, genügen, um die Korrektheit zu beweisen. Dieser Ansatz wird „mechanized result verification“ genannt. Der Nach- und zugleich Vorteil beider Ansätze, „Result Checker“ und „Mechanized Result Verification“, liegt im hohen Grad der Formalisierung. Zum einen ist dies ein Vorteil, da sie bei entsprechender Anwendung zu mathematisch beweisbaren Aussagen führen, andereseits aber auch ein Nachteil, da sie einen erheblichen mit Kosten verbundenen Mehraufwand bedeuten. Laut [Balkhis Abu Bakar ] muss für eine entsprechende formale Betrachtung zum einen die Implementierung in einer für die Analyse zugänglichen Darstellung vorliegen und zum anderen muss genügend Erfahrung vorhanden sein, um einen entsprechenden theoretischen Formalismus anzuwenden. Des Weiteren darf die Implementierung nicht zu groß werden, da ansonsten die formalen Methoden nur noch sehr schwer anwendbar sind. Diese Bedingungen sind jedoch meist nicht erfüllt. Weniger formal und deshalb anwendbarer ist der „Automated Result Verification“ genannte Ansatz von [Balkhis Abu Bakar ]. Das Ziel dieses Ansatzes ist zu zeigen, dass eine Ausführung des zu überwachenden Programms der Spezifikation seiner Anforderungen entspricht. Bei diesem Ansatz werden also zur Laufzeit die vorher festgehaltenen Anforderungen an ein System bzw. eine Softwarekomponente überprüft. Eine Berechnung einer Softwarekomponente wird genau dann als faultfree akzeptiert, wenn sie den Anforderungen genügt. Genau wie bei Result-Checkern und Mechanized-Result-Verification bedeutet auch hier ein korrektes Ergebnis für einen Programmdurchlauf nicht, dass das Programm faultfree ist, sondern „nur“, dass der aktuelle Programmablauf keinen Fault aufweist. Die Anwendung von „Automated Result Verification“ hat den weiteren Vorteil, dass dieser Ansatz auch auf off-the-shelf Komponenten angewendet werden kann, da die Spezifikationen zugänglich sind. Da das Vorgehen nur auf der Betrachtung von Eingabe- Ausgabepaaren beruht, ist seine Komplexität größtenteils unabhängig von der Größe des zu testenden Programms und seiner Implementierungssprache. Dadurch ist der Ansatz geeignet für komplexe und heterogene Systeme [Balkhis Abu Bakar ]. Dieser Ansatz bietet neben den eben genannten Vorteilen weiterhin, wie die Anderen Self-Checking-Software-Ansätze auch, Echtzeitfähigkeit. Außerdem ist der zusätzliche Aufwand zur Entwicklung der zusätlichen Software vertretbar. Es wird auch kaum zusätzliche Hardware benötigt. Auf Grund dieser Vorteile wurde aus den hier vorgestellten Self-Checking-Software-Ansätzen der „Automated Result Verification“ genannte Ansatz für den weiteren Verlauf der Arbeit ausgewählt.

# 4. Verhaltensbasierte Steuerungen und Software-Fault-Tolerance in Automotivesystemen

Im Folgenden Abschnitt wird aufgeführt, welche der vorgestellten Ansätze im weiteren Verlauf verwendet werden und warum.

Auf Grund der Anforderungen im Automotive Bereich ist als Software-Fault-Tolerance Mechanismus, von den hier vorgestellten, Self-Checking Software am vorteilhaftesten. Die Anwendung von N-Version Programming ist wegen des hohen Kostendrucks in der Automotive Branche nachteilhaft, da das erhöhte Aufkommen an Softwareentwicklungs- und Hardwarekosten groß ist. Da die meisten Anwendungen im Automotive Sektor zeitkritisch sind, sind auch Recovery-Blocks kritisch zu betrachten, da sie mit unter Umständen zeitaufwändigen Rollbacks arbeiten. Dadurch wird das Einhalten von Realtime Anforderungen sehr erschwert. Self-Checking Software hingegen benötigt keinen derartig intensiven Hardwareeinsatz wie N-Version Software und arbeitet nicht notwendigerweise mit Rollbacks. Dazu kommt eine im Vergleich zu den beiden anderen Ansätzen hohe Designdiversity. Deshalb wird im weitern Verlauf der Arbeit zur verbesserung der Safetyeigenschaften Self-Checking Software verwerndet.

Unter den verhaltensbasierten Ansätzen ist für das automobile Umfeld ein prioritätenbasierter Ansatz, wie zum Beispiel die Subsumptionsarchitektur, am besten geeignet. Bei dieser Art von Ansätzen lässt sich verhältnismäßig leicht abschätzen, welche Folgen Software- bzw. Hardwareausfälle haben.

Deshalb wird im folgenden Abschnitt ein Ansatz vorgestellt, der eine etwas veränderte Subsumptionsarchitektur mit einem Self-Checking-Software Ansatz kombiniert.

## 4.1 Graphische Darstellung

Um den prinzipiellen Aufbau der Architektur und das Zusammenspiel der einzelnen Architekturelemente darstellen zu können, wird im Folgenden eine Notation eingeführt um Zusammenhänge graphisch darzustellen.

Eingaben, die das System erhält, werden durch das Symbol in Abbildung 4.1 dargestellt. Ausgaben, die an andere Software gehen, die auf einem anderen Controller läuft, werden mit einem ähnlichen Symbol dargestellt (Abbildung 4.2). Ausgaben, die nicht an Softwarekomponenten gehen, sondern an Hardware und damit das Softwaresystem verlassen, werden wie in Abbildung 4.3 repräsentiert. Verhalten und SRS teilen sich ein Symbol und werden anhand der Beschriftung unterschieden (Abbildung 4.4). Ein Arbiter wird wie in Abbildung 4.5 dargestellt. Der unterdrückende Eingang wird mit einem roten Pfeil markiert. Um den Informationsfluss zwischen den bisher eingeführten Elementen zu modellieren, werden Pfeile verwendet (Abbildung 4.6).

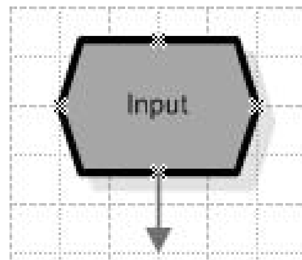



Abbildung 4.1: Darstellung von Eingaben in das System

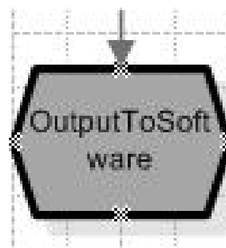



Abbildung 4.2: Darstellung von Ausgaben, die an Software auf anderen Controllern gehen

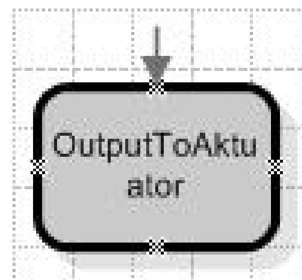



Abbildung 4.3: Darstellung von Ausgaben die an Hardware gehen.

## 4.2 Elemente der Architektur

In den folgenden Abschnitten werden die oben bereits als graphische Darstellung eingeführten Architekturelemente Arbiter, Verhalten und SRS genauer erklärt.

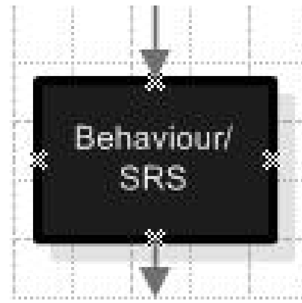



Abbildung 4.4: Darstellung von Verhalten und SRS

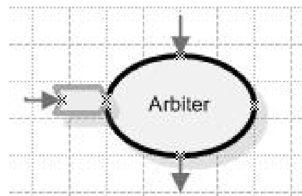



Abbildung 4.5: Darstellung eines Arbiters

### 4.2.1 Arbiter

Der Aufbau der Architektur lehnt sich wie bereits erwähnt an die von [Brooks 85] vorgestellte Subsumptionsarchitektur an. Die in dieser Arbeit verwendete Arbiterstruktur ist ähnlich. Es werden allerdings keine Inhibitoren verwendet, sondern nur vereinfachte Suppresser. Anders als die von [Brooks 85] vorgestellten Suppresser arbeiten die hier benutzten Arbiter nicht mit einer Zeitkonstanten. Die eingehenden Werte werden genau so lange überschrieben, wie das unterdrückende Verhalten aktiv ist. Der prinzipielle Aufbau eines Arbiters ist im Folgenden in einer Pseudocodedarstellung zu sehen.

```
//Input                         //output
supressedInput;                 arbiterResult;
supresseingInput;
supresserActive;

arbiter (){

  if(supresserActive>0)
    arbiterResult=supresseingInput;
  }else{
    arbiterResult=supressedInput;
  }

}
```

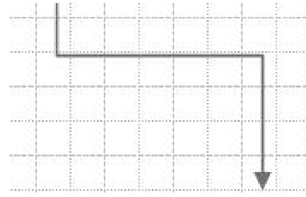


Abbildung 4.6: Der informationsfluß wird mit Hilfe von Pfeilen daregstellt

### 4.2.2 Verhalten

Das zentrale Gebilde der von [Brooks 85] vorgestellten Subsumptionsarchitektur ist das Verhalten. Durch das Zusammenspiel mehrer Einzelverhalten ergibt sich das Gesamtverhalten des Systems. Die in dieser Arbeit verwendeten Verhalten unterscheiden sich ein wenig im Aufbau.

Ein Verhalten besteht aus einem Eingabevektor $I(i_1, ..., i_n)$, der die aktuellen Eingaben des Verhaltens darstellt, und einem Ausgabevektor $O(o_1, ..., o_m)$, der die aktuellen Ausgaben des Verhaltens repräsentiert. Eingaben können Sensorwerte, Ausgaben anderer Verhalten oder Benutzereingaben sein. Die Ausgabe eines Verhaltens enthält zum einen seine Berechnungsergebnisse und zum anderen seinen Status, das heißt, ob es aktiv ist oder nicht. Der Status des Verhaltens wird durch das Aktivsignal an die nachfolgenden Arbiter geschickt. Falls Verhalten und Arbiter auf dem selben Controller laufen, ist das senden des Aktivsignals lediglich das setzen einer Variablen. Wenn Verhalten und Arbiter aber auf unterschiedlichen Controllern abgearbeitet werden, dann wird eine CAN-Nachricht geschickt, die den Status des Verhaltens übermittelt. Ob ein Verhalten aktiv ist oder nicht, ist von seiner Aktivierungsfunktion $A(i_1, ..., i_n)$ abhängig. Das Verhalten ist aktiv, wenn $A(i_1, ..., i_n)$ einen Wert ungleich Null annimmt. In diesem Fall führt das Verhalten seine Berechungen aus und setzt seine Ausgabevariablen. Die Ausgabevariablen werden dann, je nach Position des Verhaltens im System, an die ihm nachfolgenden Verhalten bezihungsweise Arbiter geschickt. Wenn $A(i_1, ..., i_n) = 0$ gilt, dann ist das Verhalten inaktiv. Da abgesehen von den Sensorwerten die Eingaben der hier verwendeten Verhalten dieselbe semantische Bedeutung für das System haben wie die Ausgaben und sich nur in ihrer Ausprägung unterscheiden ist es im Fall der Inaktivtät eines Verhaltens praktikabel die entsprechenden Datenwerte einfach ohne Veränderung durch das Verhalten hindurchzureichen und das Aktivsignal anzupassen, d.h. es auf null zu setzen.

Semantisch äquivalent bedeutet, die Ausgabewerte repräsentieren denselben Stellwert wie die Eingaben, d.h. die Eingabe ist zum Beispiel eine Gasstellung und die Ausgabe auch, jedoch der Stellwert kann ein anderer sein. Die sematische äquivalenz von Eingabe und Ausgabe gilt nicht für alle Eingabewerte. Sensorwerte werden z.B. nicht ausgegeben, obwohl sie Eingaben für ein Verhalten darstellen. Bei den hier aufgebauten Verhalten ist es aber immer so, dass jede Ausgabe ein semantisches Pendant im Eingabevektor besitzt (außnahme ist das Aktivsignal).

Der Wert des Aktivsignals ist das Ergebnis der Aktivierungsfunktion. Daher ist der Wert des Aktivsignals im Falle der Inaktivität null. Das Aktivsignal wird verwendet um den nachfolgenden Arbitern mitzuteilen ob ein Verhalten aktiv ist oder nicht.

Anders als bei den von [Brooks 85] vorgestellten Verhalten ist hier kein Reset-Eingang für ein Verhalten vorgesehen. Da ein Verhalten, wenn es einen Fehler aufweist, von seinem *Safety Related System (SRS)* durch das Überschreiben seiner Ausgänge sozusagen abgeschaltet wird und danach nicht wieder eingeschaltet wird, kann auf den Reset-Eingang verzichtet werden. Warum ein Verhalten nach seiner Abschaltung nicht wieder aktiviert wird ist in Kapitel 4.2.3 eleutert. Funktion und Aufbau des SRS wird im folgenden Abschnitt erklärt. Der prinzipielle Aufbau eines Verhaltens ist im Folgenden als Pseudocode zu sehen.

```
//Inputvector   Outputvector
Input_1;       Output_1;
Input_2;       Output_2;
   .              .
   .                 .
   .              .
Input_n           Output_m;

void Behaviour1(){

//activation function
activ = a(Input_1, Input_2, ... , Input_n);

if(active>0){
// behaviour specific computaions
Output_1 = computaion_1(Input_1, Input_2, ... , Input_n) ;
Output_2 = computaion_2(Input_1, Input_2, ... , Input_n) ;
          .
          .
          .
Output_m = computaion_m(Input_1, Input_2, ... , Input_n) ;
}else{
    Output_1=Input_1;
    Output_2=Input_2;
          .
        .
        .
    Output_m=Input_m;
}

//send Outputs to other Behaviours
sendOutputToSystem();
}
```

### 4.2.3 Safety Related System

Es ist eine weitgehend anerkannte Tatsache, dass Software nicht fehlerfrei ist. Da ein Softwarefehler in einem eingebetteten System aber fatale Folgen haben kann, müssen sie nicht nur erkannt, es muss auch in einer geeigneten Weise darauf reagiert werden. In Anbetracht dieser Tatsache wird jedem Verhalten ein Softwaremodul zur Seite gestellt, das „Safety Related System" (SRS) genannt wird. Dieses Softwaremodul ist dafür verantwortlich, ein von der Spezifikation des Verhaltens abweichendes Berechnungsergebnis zu erkennen und darauf zu reagieren. Ein Ergebnis eines Verhaltens wird also genau dann als fehlerhaft eingestuft, wenn es nicht den zuvor festgelegten Spezifikationen entspricht. Es ist also wichtig, vor der Implementierung alle Anforderungen, die an ein Verhalten gestellt werden, zu erfassen und in einer überprüfbaren Form festzuhalten. Dies gilt sowohl für Einschränkungen, die die Werte einer Ausgabe betreffen, als auch für zeitliche Beschränkungen, an die ein Verhalten zwingend gebunden ist. Da das SRS allein von der Spezifikation des zugehörigen Verhaltens abhängig ist, kann es unabhängig und ohne Kenntnis von Implementierungsdetails des zugehörigen Verhaltens erstellt werden. Ein weiterer Vorteil ist, dass ein Verhalten ohne Probleme ausgetauscht werden kann, da sich jedes neue Verhalten auch an die Spezifikationen halten muss. Jedes in dieser Arbeit verwendete SRS erhält, zusätzlich zu den Eingaben und Ausgaben des zugehörigen Verhaltens, also $I(i_1, ..., i_n)$ bzw. $O(o_1, ..., o_m)$, als Eingabe die unverarbeiteten, vom Benutzer vorgegeben Eingabewerte $u_1, .., u_v$. Der Eingabevektor eines SRS ergibt sich dadurch als $I_{SRS}(i_1, ..., i_n, o_1, ..., o_m, u_1, .., u_v)$. Die Reaktion des SRS auf das Erkennen eines Fehlers im Verhalten besteht darin, die Ausgabe des Verhaltens mit den vom Benutzer vorgegebenen, zur eigentlichen Ausgabe semantisch äquivalenten Werte zu ersetzen.

Wenn zum Beispiel das Verhalten ein ABS wäre, dann ist eine der Eingaben des Verhaltens die gewünschte Bremsstellung und eine Ausgabe die vom ABS erzeugte Bremsstellung. Dieses Ein- Ausgabepaar ist semantisch äquivalent, da beide Werte Bremsstellungen sind. Die Ausprägung des Wertes ist nach der Verarbeitung durch das ABS zwar eine andere, aber die Semantik ist die Selbe. Dabei muß die Vorgabe der Bremsstellung an das Verhalten, nicht direkt vom Benutzer kommen, sondern kann von beliebig vielen, in weiter oben liegenden Schichten arbeitenden Verhalten, vorverarbeitet worden sein. Das Berechnungsergebnis des ABS wird im Fehlerfall dann also vom SRS durch die vom Fahrer gewünschte Bremsstellung ersetzt und so das ABS abgeschaltet. Dieses Vorgehen macht Sinn, da dadurch alle Verarbeitungen die vor dem ABS stattgefunden haben verloren gehen, und somit die dem ABS folgenden Verhaltensschichten zur obersten Schicht in der Subsumption Architektur werden. Es kommt also nicht dazu, dass nur eine Schicht in der Mitte abgeschaltet wird, sondern immer alle Schichten die über einem fehlerhaften Verhalten liegen. Diser Vorgang ist aufgrund des Aufbaus der Subsumption sinnvoll. Bei dieser Architektur werden die Schichten nacheinander aufgebaut immer unter der Vorraussetzung, dass alle darunter liegeden Schichten arbeiten. Beim Abschalten einer Shicht ist das aber nichtmehr der Fall. Deshalb müssen alle Schichten die über einer fehlerhaften Schicht liegen abgeschaltetn werden. Die Benutzereingaben als Backup zu nutzen macht von daher Sinn, dass im schlimmsten Fall alle Verhalten abgeschaltet werden und so der Benutzerwunsch direkt an die Aktoren weiter geleitet wird. Bei einem Ausfall das ganzen Systems wäre das Auto also normal zu fahren. Um die Benut-

zereingaben so als Backup nutzen zu können muß aber jedes Verhalten als Eingabe Werte benutzen, die der Anwender vorgeben kann. Dieses Vorgehen hat sich während der Arbeit als praktikabel erweisen, da durch den schichtenartigen Aufbau der Architektur jedes Verhalten während der Entwicklung einmal die oberste Schicht war und somit die Eingaben direkt vom Benutzer erhalten hat.

Durch das eben dargestelle Vorgehen ergibt sich der Ausgabevektor des SRS als $O_{SRS}(u_1, .., u_v, a)$, wobei $a$ angibt, ob das SRS aktiv ist oder nicht. Ein als fehlerhaft erkanntes Verhalten wird so „abgeschaltet“. Ein Verhalten, das einmal als fehlerhaft klassifiziert wurde, wird vom SRS nicht wieder aktiviert, auch wenn es wieder Ergbnisse liefert, die der Spezifikation entsprechen.

Ein Verhalten wieder zu aktivieren in dem bereits ein Fehler erkannt wurde ist unangebracht. Man weiß nicht in welcher Situation der bereits erkannte Fehler wieder auftritt und wie er dann zum tragen kommt. Eventuell könnte so ein bereits in einer ungefährlichen Situation erkannter Fehler erneut auftreten. Das erneute Auftreten kann prinzipiell zu jedem Zeitpunkt passieren auch in Situationen in denen der Fehler kritischer ist. Weiterhin kann es durch das erneute Anschalten eines Verhaltens zum zyklischen An- und Abschalten eines desselben kommen. Wenn ein Fehler beispielsweise jede Sekunde erkannt würde, dann würde das Verhalten Ständig an- und abgeschaltet. Bei einer Kurvenfahrt mit Eingriff eines ESP wäre das ständige an- un abschalten des ESP mehr als fragwürdig. Weiterhin sollte man nicht vergessen, dass das Abschalten einer Systemkomponente bzw. eines Verhaltens nicht der Normalfall ist sondern die Ausnahme. Dadurch ist die gewählte Vorgehensweise weit weniger restriktiv also sie auf den ersten Blick erscheint. Das ein Verhalten beim erkennen eines Softwarefehlers dauerhaft abgeschaltet wird, bedeutet nicht, dass es bei Verwendung des dargestellten Ansatzes unmöglich ist ein Verhalten nur für eine bestimmte Zeitspanne zu deaktivieren. Ein Verhalten kann auch nur für die Dauer einer transiente Störungen unterdrückt werden. Das erkennen und behandeln von transiente Störungen wurde jedoch während des Aufbaus der Arbeit nicht weiter verfolgt und deshalb auch nie ein Verhalten wieder aktiviert. Der prinzipielle Aufbau eines SRS ist im Folgenden als Pseudocode dargestellt.

Die Variable $Behaviour1OK$ ist mit $true$ initialisiert und wird im Fehlerfall auf $false$ gesetzt. Da ein Verhalten bei erkanntem Fehler nicht wieder aktiviert werden soll wird $Behaviour1OK$ nicht wieder auf $true$ gesetzt.

```
//Inputvector                Outputvector

User_Input;                  User_Input;
Behaviour_In_1;              Behaviour1OK;
Behaviour_In_2;
      .
      .
      .
Behaviour_In_n ;
Behaviour_Out_1;
Behaviour_Out_2;
      .
```

```
      .
      .
Behaviour_Out_m;


void SRS_Behaviour1(){

Requirement1OK=checkRequirement1(Inputvector, Outputvector);
Requirement2OK=checkRequirement2(Inputvector, Outputvector);
                        .
                        .
                        .
RequirementkOK=checkRequirementk(Inputvector, Outputvector);


if(!( Requirement1OK && Requirement2OK && ... && RequirementkOK)){
        Behaviour1OK = false;
}

  sendOutputToSystem();
}
```

Das Konzept des SRS ist aber nicht nur auf einzelne Verhalten anwendbar. Das ist von Vorteil, da nicht alle Anforderungen einem einzigen Verhalten zuzuordnen sind. Es können auch globale, das ganze System betreffende Spezifikationen überprüft werden. Die Eingaben eines globalen SRS sind die Benutzereingaben und die Systemausgaben. Wie in diesem Fall ein entsprechendes Eingreifen eines globalen SRS realisiert werden kann, wird in Abschnitt 4.3 näher erläutert. Ein global eingesetztes SRS kann bei Erkennen eines Fehlers sicherstellen, dass absolut notwendige Grundfunktionalitäten für den Anwender benutzbar bleiben, oder das System, sozusagen durch einen Reflex, in einen sicheren Zustand versetzen. Die Analogie zu einem Reflex besteht darin, dass die Systemantwort nicht durch das eigentliche, kompliziertere Softwaresystem beziehungsweise Gehirn erzeugt wird, sondern durch eine Art Bypass, dem SRS bzw. Rückenmark. Ein globales SRS unterscheidet sich dabei von einem normalen SRS. Ein globales SRS kann beim Erkennen eines Fehlers nicht nur die Benutzereingaben weiterleiten, sondern selbstständig Stellwerte generieren. So kann eine gloables SRS das Fahrzeug zum Stillstand, und somit in einen sicheren Zustand, bringen in dem es die Bremsen schließt. Während ein SRS das Fahrzeug abbremst, also in einen sicheren Zustand bringt, sind seine Ausgaben nicht von Sensorwerten oder anderen Verhalten abhängig. Wie bei einem Reflex erfolgt die Reaktion auf den Reiz nach einem fest vorgegeben Ablaufmuster, sozusagen „unwillkürlich“. Das System hat keine Möglichkeit sich dem Bremsen zu wiedersetzen, sobald das gloable SRS ausgelöst wurde.

## 4.3 Aufbau der Architektur

Im folgenden Abschnitt wird erklärt wie die in den vorrangegangenen Kapiteln erklärten Architekturelemente zusammengefügt werden.

Wie bereits in Abschnitt 4.2.3 erwähnt, ist es die Aufgabe eines SRS, ein Verhalten zu überwachen und gegebenenfalls die Ausgaben des Verhaltens zu überschreiben. Da das Wirken des SRS aber unabhängig von der Implementierung des Verhaltens sein soll, darf die Verarbeitung eines erkannten Fehlers nicht vom Verhalten selbst vorgenommen werden. Es wäre bedenkenswert, diese Verarbeitung einem Softwaremodul zu überlassen, in dem gerade ein Fehler erkannt wurde. Deshalb werden Verhalten und zugehöriges SRS, wie in Abbildung 4.9 zu sehen, über einen Arbiter gekoppelt. Das SRS hat die Kontrolle über den unterdrückenden Eingang. Da das SRS genau dann aktiv wird, wenn es einen Fehler erkennt, werden in diesem Fall die Verhaltensausgaben, die am unterdrückten Eingang des Arbiters anliegen, überschrieben. Das Zusammenspiel des Arbiters mit Verhalten und SRS ist in Abbildung 4.7 bzw. Abbildung 4.8 als Sequenzdiagramm dargestellt.

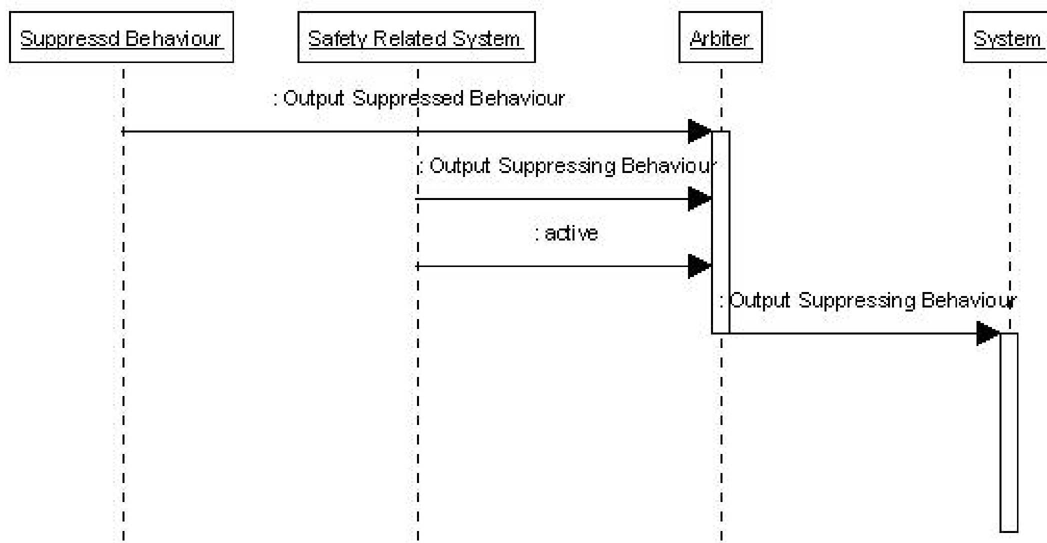



Abbildung 4.7: Arbiter, bei dem das unterdrückende Verhalten aktiv ist

Dieses Vorgehen hat mehrere Folgen. Die erste und offensichtlichste Folge ist, dass sich so erkannte Fehler in einem derart überwachten Verhalten nicht über das gesamte System ausbreiten können. Die zweite, weniger offensichtliche Folge ist „graceful degradation“ mit Determiniertheit bei erkannten Fehlern beziehungsweise Ausfällen. Dieser Effekt entsteht durch die Schichtung der von [Brooks 85] vorgestellten Architektur. Jede der Architekturschichten realisiert ein von außen sichtbares Verhalten. Die Schichten sind dabei so angeordnet, dass eine Schicht die Funktionalität der vorherigen Schichten ergänzt. Nach Fertigstellung einer Schicht wird diese intensiv getestet, wobei alle darunterliegenden Schichten arbeiten. Jede Schicht ist also nur getestet unter der Voraussetzung, dass all ihre darunterliegenden Schichten aktiv sind. Wenn ein Verhalten bzw. eine Schicht jetzt vom SRS abgeschaltet wird, werden nur die darüber liegenden Verhalten und das fehlerhafte Verhalten selbst deaktiviert,

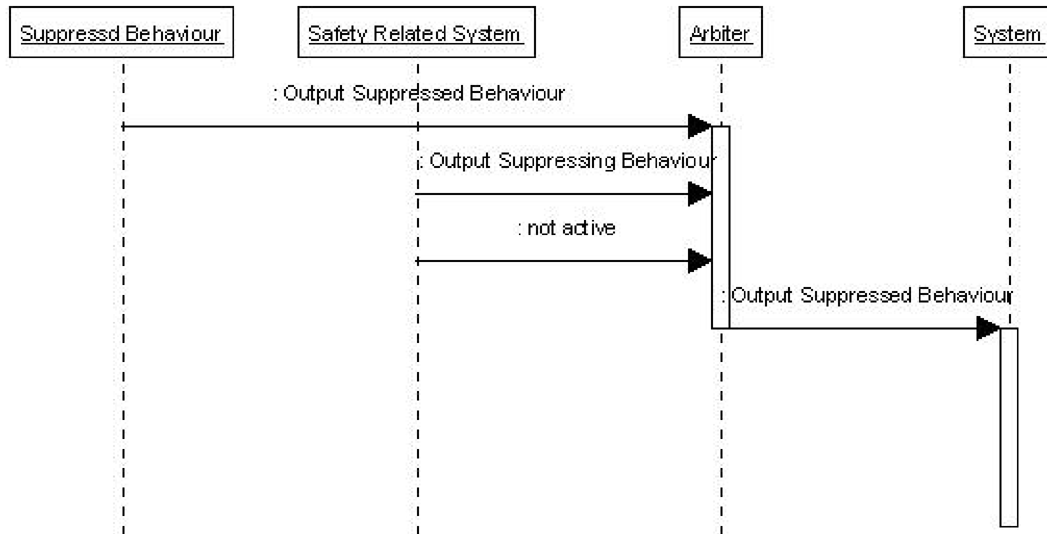



Abbildung 4.8: Arbiter, bei dem das unterdrückende Verhalten nicht aktiv ist

wodurch der Effekt von „graceful degradation“ entsteht. Die Determiniertheit tritt auf, da beim Aktivwerden eines SRS nicht nur eine Schicht abgeschaltet wird. Es werden implizit auch alle darüberliegenden Schichten abgeschaltet. Das ist der Fall, da das SRS die unverarbeiteten Benutzereingaben in den Arbiter einspeist und diese an das System weitergereicht werden. Es gehen also alle Verarbeitungen auf den Eingabedaten, die vor dem vom SRS kontrollierten Arbiter stattgefunden haben, verloren. Das „Abschalten“ aller darüberliegender Schichten macht Sinn, da diese Schichten während der Entwicklung des Systems nur für die Vorraussetzung getestet und implementiert wurden, dass alle darunterliegenden Schichten arbeiten. Wenn ein SRS eine unten liegende Schicht deaktiviert, ist das aber nicht mehr für alle Verhalten der Fall. Wenn die oben liegenden Schichten jetzt nicht abgeschaltet werden würden, käme es zur Ausführung einer vorher nicht getesteten Zusammestellung von Schichten. In sicherheitskritischen Umgebungen ist das jedoch nicht akzeptabel. In Abbildung 4.10 wird die Arbeitsweise graphisch dargestellt und erläutert.

In einer ähnlichen Art und Weise kann das Konzept des SRS auf Bedingungen angewendet werden, die das gesamte System betreffen. Dabei wird das gesamte System als ein einziges Verhalten betrachtet. Ein solches globales SRS erhält als Eingaben die Benutzereingaben und die Ausgaben des Gesamtsystems, die das Softwaresystem an Aktoren gibt. Verbunden werden System und SRS dann wieder über einen Arbiter. Dieser Arbiter muss als einziges Softwareelement zwischen dem System und den Aktoren stehen. So wird gewährleistet, dass nicht doch andere Ausgaben an die Aktoren gelangen. Ein globales SRS kann entweder genau wie ein normales SRS die Benutzereingaben einspeisen oder versuchen, das System in einen sicheren Zustand zu bringen. Es wäre für ein Automobil zum Beispiel denkbar, bei einem Fehler, der vom globalen SRS erkannt wurde, das Fahrzeug abzubremsen, wobei der Fahrer aber die volle Kontrolle über die Lenkung behält. Bezüglich der Lenkung würden also die unverarbeiteten Benutzereingaben an die Aktoren weitergeleitet und bezüglich Gas und Bremse die vom globalen SRS erzeugten Ausgaben um das Fahrzeug anzuhal-

ten. Es sind aber auch Funktionen wie Begrenzen der maximalen Geschwindigkeit oder Begrenzen der maximalen Gasstellung als Eingriffe denkbar und wurden auch in der Arbeit implementiert. Ein Unterschied zwischen SRS und globalem SRS ist jedoch, dass es bei einem globalen SRS sinnvoll sein kann nicht nur die Ausgaben des Gesamtsystems zu überschreiben, sondern die Information einen betimmten Fehler erkannt zu haben an bestimmte Teile des Systems weiterzugeben. Ein derartiger Fehler kann das Erkennen eines Sensorausfalls sein. Ein solcher erkannter Fehler wird aber nie an ein Verhalten weitergeleitet, sondern immer nur an ein oder mehrere SRS. Ein benachrichtigtes SRS kann dann entscheiden, ob es notwenig ist, ein Verhalten abzuschalten oder nicht. Ein weiterer Unterschied besteht darin, dass eine globale SRS nicht nur die Benutzereingaben zum überschreiben verwendet, sondern selbst generierte Werte. Wenn das globale SRS dazu verwendet wird, dass Fahrzeug in einen sichern Zustand zu bringen, dann erfolgt die Ausgabe der Werte nach der Aktivierung des gloablen SRS unabhängig von anderen Verhalten oder Sensoren. Die Ausgabe der Werte folgt wie bei einem Reflex einem festen Muster. Das ist sinnvoll, da ein vom globalen SRS erkannter Fehler durch andere Verhalten oder einen Sensorfehler ausglöst worden sein kann. Durch den festen Ablauf können diese Fehler das gloable SRS nicht beeinflussen. Eine entsprechende Erweiterung des Systems aus Abbildung 4.10 um ein globales SRS ist in Abbildung 4.11 zu sehen. Alle in diesem Abschnitt vorgestellten Idden und Ansätze wurden in einer Beispielarchitektur, die im nächsten Kapitel vorgestellt wird, implementiert und ausprobiert. Die Anordnung der Verhalten erfolgt wie in der von [Brooks 85] vorgestellten Subsumption Architektur.

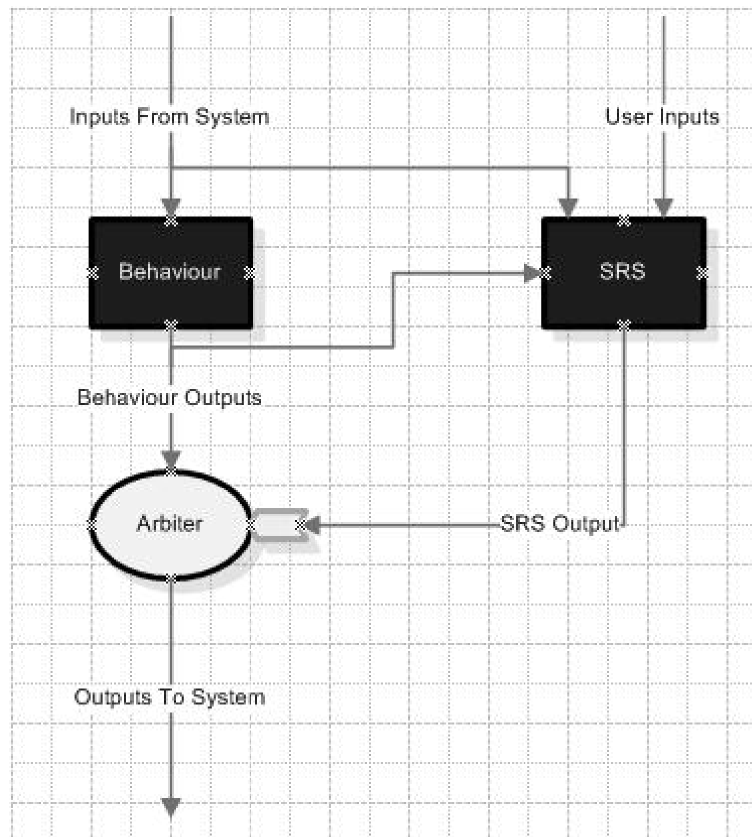



Abbildung 4.9: Zusammenspiel von Verhalten, Arbiter und SRS

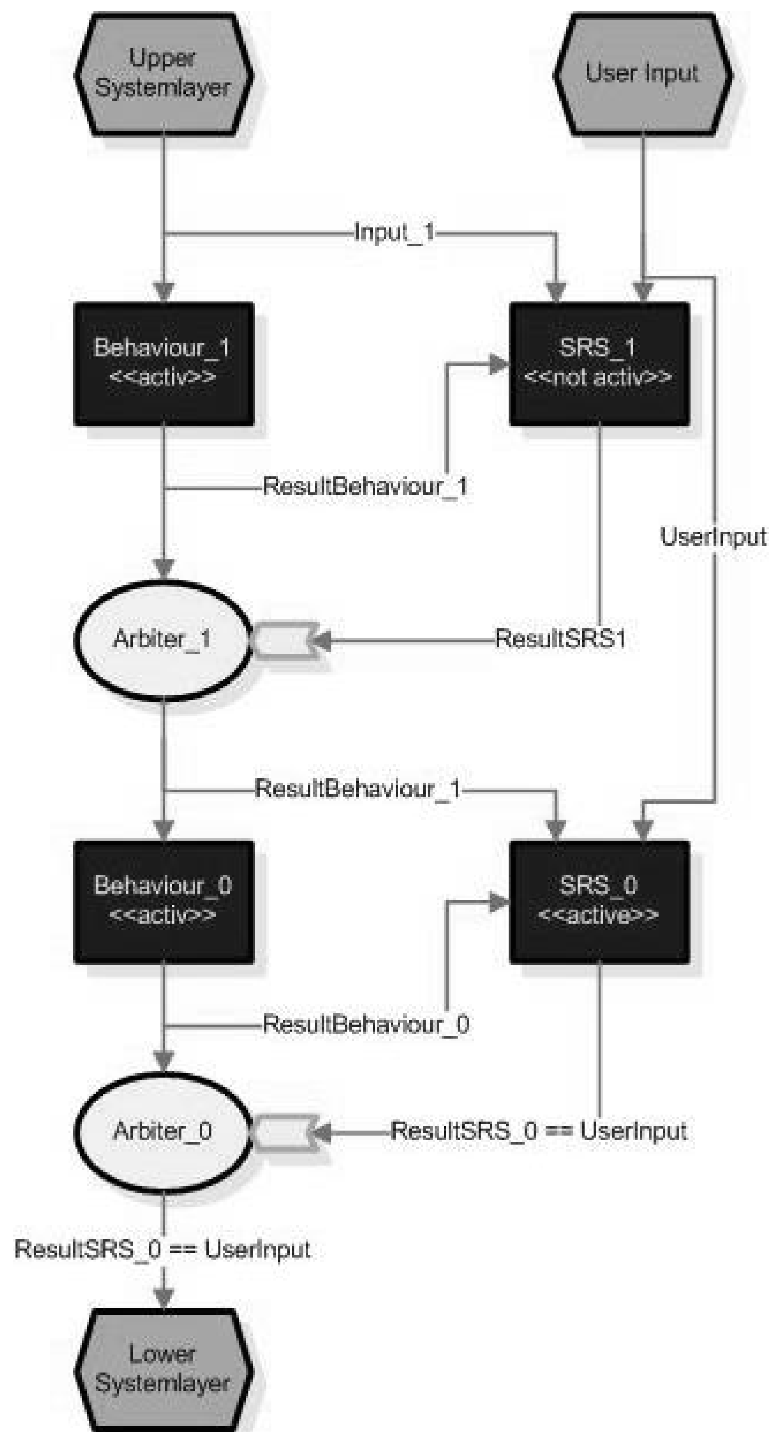



Abbildung 4.10: Verhalten1 ist aktiv und SRS1 nicht. Deshalb ist das Ergebnis von Arbiter1 die Ausgabe von Verhalten1. Verhalten0 ist aktiv, SRS0 jedoch auch das heißt es wurde ein Fehler erkannt. Deshalb sind die Ausgaben von Arbiter0 die Ausgaben von SRS0. SRS0 gibt die unverarbeiteten Benutzervorgaben weiter. Die Verarbeitungsschritte von Verhalten1 gehen verloren.

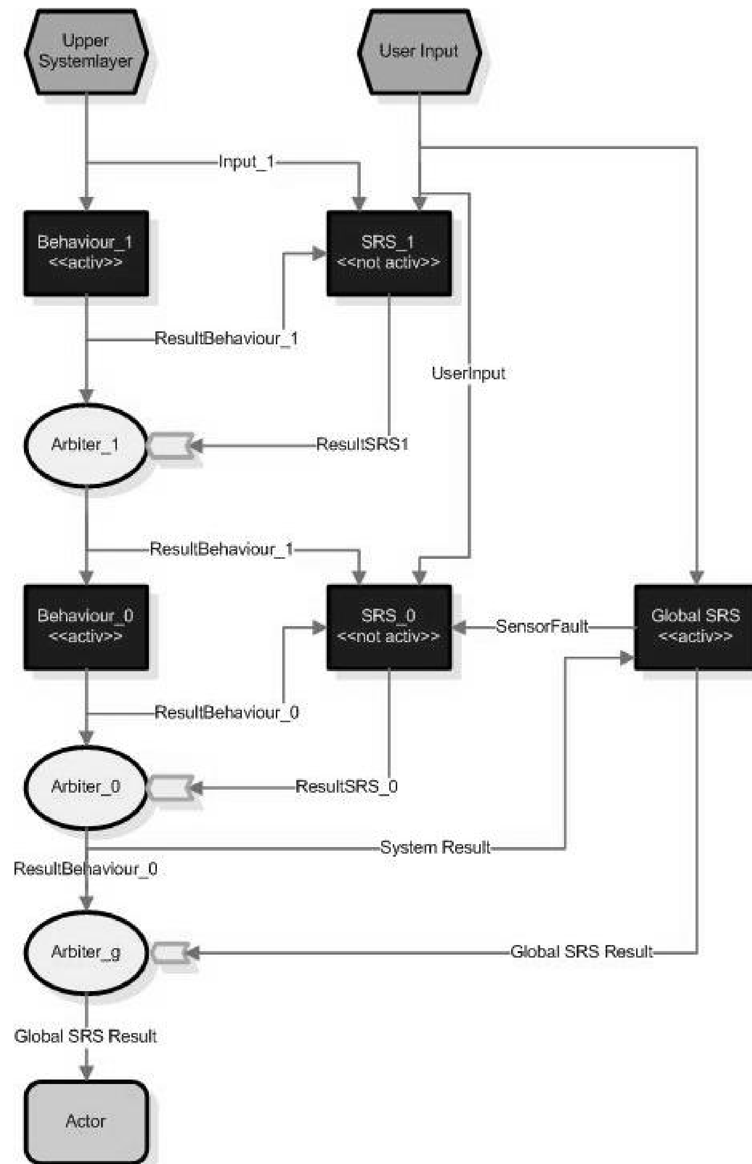



Abbildung 4.11: Das globale SRS meldet einen Sensorfehler an SRS0. SRS0 wird aber nicht aktiv, da dieser Fehler für Behaviour0 akzeptabel ist. Das globale SRS wird jedoch aktiv und überschreibt die Ausgaben des gesamten Systems.

# 5. Beispiel Architektur

Im Rahmen der vorliegenden Arbeit wurde ein Framework implementiert, mit dessen Hilfe es möglich ist, den oben beschriebenen Ansatz auf einem Microkontroller (AT90CAN128) umzusetzen. Die damit entstandene Umsetzung einer Steuerungssoftware soll dazu herangezogen werden, Vor- und Nachteile des Ansatzes an einem praktischen Beispiel testen zu können. Ziel der Beispielimplementierung ist, möglichst alle Aspekte des vorgestellten Ansatzes testen zu können. Einen Überblick über die Implementierung gibt Abbildung 5.1.

Als Kernfunktionalität wurden drei Verhalten implementiert. Schicht Nummer zwei bildet eine Antriebs Schlupf Regelung (ASR). Sie soll verhindern, dass die Antriebsräder des Versuchsträgers bei starkem Beschleunigen stark durchdrehen können. Es wird versucht, den Antriebsschlupf nicht kleiner als einen bestimmten Wert werden zu lassen. Als Eingabewerte erhält die ASR eine Wunschvorgabe für die Stellung des „Gaspedals“ und für jedes Rad eine Drehgeschwindigkeit. Falls der entstandene Schlupf kleiner als ein bestimmter Wert sein sollte, kann er durch Anpassen der Gasstellung wieder auf eine gewünschte Größe angehoben werden. Dabei wurden folgende Anforderungen gestellt:

1. Das Einregeln darf nicht länger als eine bestimmte Zeit dauern.
2. Die vom ASR eingestellte Gasposition darf nie mehr Gas geben als die Vorgabe.
3. Die ASR darf nicht bremsen.
4. Die ASR darf nur aktiv sein, wenn der Antriebsschlupf einen bestimmten Wert unterschreitet.
5. Wenn ein Antriebsschlupf vorliegt der kleiner als ein bestimmter Wert ist und die Vorgabe verlangt mehr Gas als im vorherigen Regelschritt, dann darf die ASR die Gasstellung trotzdem nicht erhöhen.

Die fünf oben aufgeführten Punkte werden vom zur ASR gehörenden SRS während der Laufzeit nach jedem Regelschritt überprüft. Falls eine der Bedingungen nicht

erfüllt sein sollte, wird die ASR durch ihr SRS wie bereits oben beschrieben mit Hilfe eines Arbiters abgeschaltet, noch bevor die fehlerhaften Werte zu den Aktoren gelangen können. Durch Überprüfung von Punkt 2 wird gewährleistet, dass die ASR das Fahrzeug nicht unkontrolliert beschleunigt. Punkt 3 stellt sicher, dass die ASR das Fahrzeug nicht unkontrolliert bremst. Die Überprüfung von Punkt 1 bewirkt, dass die Aktivität der ASR nicht viel zu klein ausfallen darf, was ja ein Fehler wäre. Durch die Überprüfung von Punkt 4 wird sichergestellt, dass die ASR nur eingreifen darf, wenn es auch notwendig ist. Punkt 5 wurde eingeführt, um ein gutes Funktionieren der ASR sicherzustellen. Die Beispielimplementierung wurde so aufgebaut, dass beim Ausfall der ASR bzw. beim Erkennen eines Fehlers die darüberliegenden Softwareschichten mit abgeschaltet werden. Danach lässt sich das Fahrzeug genauso steuern, als ob keine den Fahrer unterstützenden Softwaresysteme vorhanden wären. Der Aufbau wurde so gewählt, um den Effekt der „Graceful Degradation“ zeigen zu können. Es ist für die restlichen, dann abgeschalteten Systeme nicht zwingend notwendig, eine ASR als Unterstützung zu haben.

Die über der ASR liegende Schicht Nummer drei enthält zwei der drei Kernverhalten. Diese beiden Verhalten sind im Architekturgebilde so eingebaut, dass beim Abschalten eines der beiden Verhalten das andere unabhängig weiterarbeiten kann. Das eine Verhalten stellt eine Kennlinie, das andere einen Kurvenassistenten dar. Der Kurvenassisitent legt in Abhängigkeit vom momentanen Lenkeinschlag eine maximale Gasstellung fest. Falls die Wunschvorgabe diesen Wert überschreitet, wird nur der errechnete Maximalwert eingestellt. Dies führt dazu, dass eine Kurve nicht mit beliebiger Geschwindigkeit durchfahren werden kann. Dabei wurden die folgenden Anforderungen gestellt:

1. Der Kurvenassistent darf nur bei Lenkeinschlägen eingreifen, die mehr als 80 Prozent des maximalen Lenkeinschlages betragen.
2. Die reduzierte Gasstellung darf höchstens um 20 Prozent von der Vorgabe abweichen.
3. Das Berechnungsergebnis muss kleiner sein als die Vorgabe.

Das zum Kurvenassistenten gehörige SRS überprüft diese drei Punkte. Sobald eine Verletzung einer dieser drei Anforderungen erkannt wird, wird der Kurvenassistent abgeschaltet. Die Kennline arbeitet danach anders als beim Abschalten der ASR, trotzdem unabhängig weiter und wird davon nicht beeinflusst. Das ebenfalls in Schicht Nummer drei enthaltene Kennlinienverhalten sorgt dafür, dass bei einem schnellen Gasgeben die Vorgabe nicht sofort an die zugehörige Aktuatorik weitergereicht wird. Es wird beginnend bei der momentanen Gasstellung eine über der Zeit aufgetragene Kennlinie durchlaufen, bis die gewünschte Gasstellung erreicht ist. Dabei werden die folgenden Anforderungen gestellt:

1. Es darf ein maximaler Gradient beim Durchlaufen der Kurve nicht überschritten werden.
2. Der Abstand der Punkte, mit denen die Kennlinie durchlaufen wird, darf einen vorgegebenen Wert nicht übeschreiten.

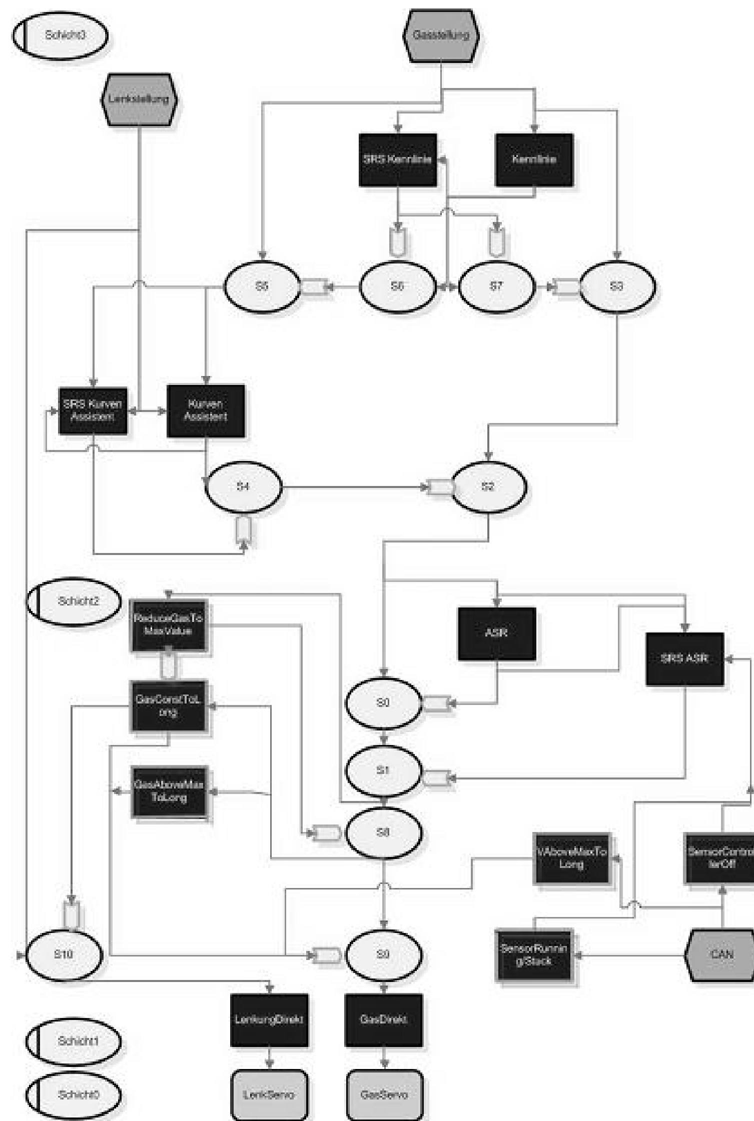



Abbildung 5.1: Übersicht über eine mit Hilfe des Frameworks erstellten Softwarearchitektur

3. Die Zeit, die benötigt wird, um die Kennlinie bis zur Vorgabe zu durchlaufen, darf einen Maximalwert nicht überschreiten.

4. Die Kennlinie darf nur aktiv sein, wenn die Vorgabe mehr Gas vorgibt, als momentan eingestellt ist.

Das SRS, das zur Überwachung der Kennlinie eingesetzt wird, greift ein, sobald eine der Anforderungen nicht berücksichtigt wird. Die Beschränkung des Gradienten verhindert, dass die Kennlinie beliebig große Sprünge enthalten kann. Durch die Überprüfung der Punktabstände wird gewährleistet, dass die Kennlinie auch wirklich durchlaufen wird, und dabei keine Zwischenschritte ausgelassen werden. Falls jedoch zu viele Zwischenschritte eingefügt werden sollten, greift Bedingung vier, da in diesem Fall mehr Zeit zum Durchlaufen der Kennlinie benötigt wird, wie zugelas-

sen ist. Der letzte Punkt dient dazu, zu verhindern, dass die Kennlinie beim Bremsen aktiv sein könnte.

Unterhalb der beiden beschriebenen Schichten liegt Schicht Nummer eins. Die Funktionalität dieser Schicht ist verhältnismäßig einfach. Sie trägt dafür Sorge, dass die vom System generierten Aktuatorvorgaben in die entsprechenden physikalischen Signale umgesetzt werden, die benötigt werden um die Aktuatorik anzusteuern. Eine wichtige Eigenschaft dieser Schicht ist ihre Einfachheit, da es wichtig ist, dass sie in jedem Fall ihre Aufgabe erfüllt und fehlerfrei ist.

Zwischen Schicht Nummer eins und Nummer zwei wurde eine Arbiterstruktur eingefügt. Diese Arbiterstruktur erlaubt es mehreren auf einem anderen Mikrocontroller laufenden SRS im Fehlerfall einzugreifen. Die auf einem anderen Mikrocontroller ausgelagerten SRS sind globale SRS (siehe Abschnitt 4.3). Sie überprüfen das Gesamtsystem auf Fehler oder schränken es ein. Es werden mehrere globale Fehler erkannt. Wenn die Aktuatorvorgabe für eine festgelegte Zeitspanne konstant ist, wird von einem Fehler ausgegangen, da es beim verwendeten Demonstrator auf Grund der Bauweise nicht möglich ist eine Gasvorgabe von Hand exakt beizubehalten. Es wird im Falle einer konstanten Vorgabe (außer Nullstellung oder Maximalausschlag) also davon ausgegangen, dass die Vorgabe fälschlicherweise von einer Softwarekomponente erzeugt wurde. Ebenso wird es als Fehler eingestuft, wenn für einen eingestellten Zeitraum ein Grenzwert für die Gasstellung oder Fahrzeuggschwindigkeit überschritten wird. Weitere erkannte Fehler sind Ausfälle der Mikrocontroller, die für die Erfassung der Sensordaten erforderlich sind, und Ausfälle der einzelnen Sensoren. Bei erkannten Fehlern der Sensorik werden die SRS der Verhalten informiert, die die Sensoren benötigen. Die jeweiligen SRS entscheiden dann darüber, ob das zugehörige Verhalten abgeschaltet wird oder nicht. Beim Erkennen eines der andern Fehler sind mehrere Reaktionen denkbar. Zum einen können die vom Benutzer gewünschten unveränderten Eingabewerte direkt an die Aktuatoren weitergegeben werden, um so einen Komplettausfall des Fahrzeuges zu verhindern und die Grundfunktionalität zu erhalten. Eine andere Möglichkeit ist, im Fehlerfall zu versuchen, das Fahrzeug in einen sicheren Zustand zu bringen. Dies kann zum Beispiel dadurch erfolgen, dass das Fahrzeug bis zum Stillstand abgebremst wird während die Lenkung unter der Kontrolle des Benutzers bleibt. Um das Eingreifen der globalen SRS besser demonstrieren zu können, wurde in dieser Arbeit die zweite Variante gewählt. Zusätzlich zu den bereits genannten globalen SRS wurde ein globales SRS eingeführt, dass beim Überschreiten einer vorgegebenen Gasstellung die Ausgabe des Gesamtsystems mit einem Maximalwert überschreibt. Dieser, die maximale Gasstellung bestimmende Wert, ist so angepasst, dass damit die maximale Geschwindigkeit, die von einem anderen Teil des globalen SRS überprüft wird, nicht überschritten werden kann. Zusätzlich ist der die Gasstellung bestimmende Wert, durch den die Systemausgabe ersetzt wird, geringer als die vom gloablen SRS erlaubte Maximalstellung. Dadurch, dass ab einem Grenzwert am Gas nur noch ein Maximalwert eingestellt wird, ist es möglich, dass eine konstante Gasstellung erzeugt wird. Da dies von einem Teil des globalen SRS aber als Fehler erkannt werden würde, wird der entsprechende Teil des globalen SRS darüber in Kenntnis gesetzt, dass dieser konstante Wert auf das Eingreifen eines globalen SRS zurückzuführen und somit in Ordung ist. Die einzelnen Komponenten des globalen SRS kommunizieren also miteinander und überprüfen

zum Teil auch andere Komponenten des globalen SRS. Die Komponenten des globalen SRS sind in Abb. 5.1 durch eine farblich geänderte Umrandung gekennzeichnet.

Bei einer normalen Fahrt sind keine den Fahrer unterstützenden Eingriffe notwendig. Aus diesem Grund ist der normale Zustand eines Verhaltens die Inaktivität. Weiterhin treten im Normalfall auch keine Fehler in Verhalten auf, weshalb normalerweise auch die SRS nicht aktiv sind. Wenn Verhalten und SRS nicht aktiv sind erfolgt auch keine Unterdrückung an den Eingängen der Arbiter. Der normale Weg des am Aktor eintreffenden Stellwertes ist also wie folgt: Die Eingabe der Gasstellung erfolgt durch den Benutzer und wird an die Kennlinie, das zur Kennlinie gehörende SRS, Arbiter drei und Arbiter fünf weitergeleitet (Numerierung siehe Abbildung 5.1). Da weder Kennlinie noch das zugehörige SRS aktiv sind erfolgt keine Unterdrückung an Arbiter drei und fünf. Das Ergebnis von Arbiter fünf, also die Benutzereingaben, werden an den Kurvenassistenten und die dazugehörige SRS geleitet. Da aber auch Kurvensassistent und sein SRS nicht aktiv sind erfolgt keine Unterdrückung an Arbiter vier und Arbiter zwei. Das Ergebnis von Arbiter zwei ist somit die Ausgabe von Arbiter drei, was die Benutzereingaben sind. Die Ausgabe von Arbiter zwei ist die unterdrückbare Eingabe von Arbiter null. Der unterdrückende Eingang von Arbiter null wird durch das ASR kontrolliert. Auch das ASR ist im Normalfall nicht aktiviert und die Ausgabe von arbiter null ist die Eingabe des nicht unterdrückenden Eingangs von Arbiter eins, an dem die Benutzereingaben anliegen. Das Ergebnis von Arbiter null ist der unterdrückbare Eingabe von Arbiter eins. Der unterdrückte Eingang von Arbiter eins steht unter der Kontrolle des zum ASR gehörenden SRS. Auch dieses SRS ist normalerweise nicht aktiviert. Die Ausgabe von Arbiter eins ist somit wieder die Benutzereingabe und wird an den unterdrückbaren Eingang von Arbiter acht weitergegeben. Arbiter acht gibt sein Ergebnis an Arbiter neun. Die unterdrückenden Eingänge von Arbiter acht und neun stehen unter dem Einfluß der globalen SRS die bei normaler Fahrt auch nicht aktiv sind. Die Ausgabe von Arbiter neun wird an den Aktoren eingestellt. Das heißt normalerweise, wenn kein den Fahrer unterstützendes System eingreift werden die Benutzereingaben ohne Verarbeitung an die Aktoren durch das System hindurchgeleitet.

Es kann aber auch passieren, dass alle Verhalten aktiv werden und kein SRS. Ist das der Fall ergibt sich der folgende Ablauf: Die Eingabe der Gasstellung erfolgt durch den Benutzer und wird an die Kennlinie, das zur Kennlinie gehörende SRS, Arbiter drei und Arbiter fünf weitergeleitet. Da die Kennlinie aktiv ist, und das dazugehörige SRS nicht, ist die Ausgabe von Arbiter sieben das Berechnungsergebnis der Kennlinie. Die Ausgabe von Arbiter sieben wird an Arbiter drei geleitet. Arbiter drei reicht das Ergebnis von Arbiter sieben weiter, da die Kennlinie aktiv ist und das Ergebnis von Arbiter sieben am unterdrückenden Eingenag von Arbiter drei anliegt. Aus den selben Gründen ist das Ergebnis von Arbiter fünf das Resultat der Kennlinie. Die Ausgabe von Arbiter fünf wird an den Kurvenassistenten geleitet und das dazugehörige SRS. Der Kurvenassistent erhält jetzt also nicht die Benutzervorgaben als Eingabe, sondern das Berechnungsergebnis der Kennline, das genau wie die Benutzervorgabe eine Gasstellung repräsentiert. Der Kurvenassistenten hat davon aber keine Kenntnis und verhält sich genau so als ob die Vorgabe direkt vom Benutzer kommen würde. Da der Kurvenassistent den Wert nur reduzieren und nicht erhöhen kann repräsentiert das von ihm berechnete Ergebnis auf jeden Fall eine geringere

Gasstellung wie die ihm vorgegebene. Da das SRS des Kurvenassistenten nicht aktiviert ist, ist seine Ausgabe auch das Ergebnis von Arbiter vier. An Arbiter zwei liegen jetzt zwei Berechnungsergebnisse von Verhalten an. Am unterdrückbaren Eingang das Ergebnis der Kennlinie und am unterdrückenden das Ergebnis des Kurvenassistenten. Der Aufbau der Architektur ist hier so gewählt, dass der Kurvenassistent eine höhere Priorität hat als die Kennlinie, weshalb hier das Ergebnis von Arbiter zwei das Resultat des Kurvenassistenten ist. Es wäre auch umgekehrt möglich gewesen. Die Ausgabe von Arbiter zwei, also das Ergebnis des Kurvenassistenten, ist die Eingabe von ASR und dem dazugehörigen SRS. Da in diesem Beispiel alle Verhalten aktiv sein sollen ist es auch das ASR. Das ASR berechnet die von ihm maximal zugelassene Gasstellung, die wieder nur kleiner sein kann als die ihm vorgegebene. Das ASR gibt sein Ergebnis über Arbiter null an das System weiter. Da das ASR an diesem Arbiter die Kontrolle über den unterdrückenden Eingang hat wird dort das Ergebnis des Kurvenassistenten, das am unterdrückbaren Eingang von Arbiter null anliegt, überschrieben. Da die folgenden Arbiter eins, acht und neun von SRSen kontrolliert werden, die in diesem Beispiel nicht aktiv sind, wird das Ergebnis des ASR an die Aktoren weitergeleitet.

Wäre im letzt genannten Szenario das SRS der Kennlinie aktivert worden, dann hätte dieses SRS die Ausgaben der Kennlinie an Arbiter sechs und sieben mit den Benutzervorgaben überschrieben. Dadurch wären die Eingaben von Kurvenassistent, dazugehörigem SRS und unterdrücktem Eingang von Arbiter zwei die Benutzervorgaben. Falls Der Kurvenassistent aktiv ist, ist der weitere Verlauf an Arbiter zwei, derselbe wie im zweiten Szenario. Wenn der Kurvenassistent nicht aktiv ist, dann erfolgt die Verarbeitung an Arbiter zwei wie im ersten Szenario.

Beim Eingriff des zum Kurvenassistenten gehörenden SRS würden die Ausgaben des Kurvenassistenten bei Arbiter vier überschrieben und der unterdrückende Eingang an Arbiter zwei ist nicht aktiv. Das bedeutet, dass die Eingabe die am unterdrückbaren Eingang von Arbiter zwei anliegen an das System weitergegeben werden. Das sind entweder die Ausgaben der Kennlinie oder die Benutzervorgaben. Bei einem erkannten Softwarefehler im Kurvenassistenten kann die Kennlinie also weiter funktionieren.

Anders verhält es sich wenn das zum ASR gehörende SRS eingreift. Es überschreibt die Ausgaben des ASR bei Arbiter eins mit den Benutzervorgaben. Dadurch gehen die Berechnungen des ASR und die davor durchgeführten Berechnungen verloren. Es werden also alle Verhalten abgeschaltet. Bei den hier vorgestellten Verhalten ist das nicht zwingend notwendig, wurde aber trotzdem so gemacht um zu zeigen, dass alle darüber liegenden Schichten abgeschaltet werden können.

Um Ausfälle von Controllern oder Schwiedrigkeiten mit der CAN-Kommunikation zu erkennen, sendet jeder Controller ein Heartbeatsignal. Bleibt dieses Signal aus, wird von einem Ausfall des Controllers ausgegangen.

# 6. Aufbau des Versuchsträgers

Grundlage für den Aufbau des Versuchsträgers ist ein „Carson On-Road-Chassis C-5 RtR“ im Maßstab 1:5. Dieses Fahreug verfügt über eine Standard-Empfängereinheit mit zwei Kanälen. Diese Empfängereinheit generiert die für Modellbauservomotoren üblichen Signale in Abhängigkeit der Stellung der Fernbedienung. Normalerweise werden die Signale der Empfängereinheit direkt an die Servomotoren weitergeleitet. Beim gegenwärtigen Aufbau des Versuchsträgers wurde diese direkte Verbindung von Empfängermodul zu den Servomotoren unterbrochen. In die entstandene Lücke wurde ein Mikrocontroller (AT90CAN128) eingefügt. Dieser Mikrocontroller empfängt die Signale der Empfangseinheit, verarbeitet sie wie in den vorangestellen Kapiteln erklärt und generiert danach die entsprechenden Signale für die Servomotoren. Wie bereits erwähnt, werden zwei Mikrocontroller verwendet. Controller zwei nimmt keine Signale des Empfängermoduls auf, sondern kommuniziert via CAN-Bus mit Controller eins. Dieser wird im Folgenden als Behaviourcontroller bezeichnet und Controller zwei als Constraintcontroller. In den folgenden Abschnitten wird auf die Art der Signale eingegangen, wie sie empfangen beziehungsweise generiert werden und auf die verwendete Elektronik.

## 6.1 PWM Steuersignale empfangen und senden

Die Ansteuerung eines Modellbauservomotors erfolgt durch die Verwendung von PWM Signalen, die jedoch ein bestimmmtes „Datenformat“ erfüllen müssen. Die Ansteuerung erfolgt immer in einem 20ms Intervall. Die erste Millisekunde dieses Intervalls muß immer mit „logisch 1“ belegt werden. Die zweite Millisekunde ist die entscheidende und gibt die Stellung des Servomotors an. Ist der Wert des PWM Signals während der kompletten zweiten Millisekunde auf „logisch 1“, dreht sich der Servomotor in Richtung des linken Vollausschlags. Ist der Wert jedoch während der gesamten zweiten Millsekunde des PWM Intervalls auf „logisch 0“ so dreht sich der Servomotor zum rechten Vollauschlag. Die Mittelstellung wird dadurch erreicht, dass die zweite Millisekunde in der ersten Hälfte mit einer „1“ belegt wird. Durch das Variieren der Belegung der zweiten Millisekunde kann so theoretisch jede beliebige Position eingestellt werden. Die restlichen 18ms des Intervalls sind mit „logisch 0“ zu

belegen. Der zeitliche Verlauf eines solchen Signals ist in Abbildung 6.1 dargestellt. Das Erfassen der PWM-Signale wird im Mikrocontroller mit Hilfe von Interrupts bewerkstelligt.

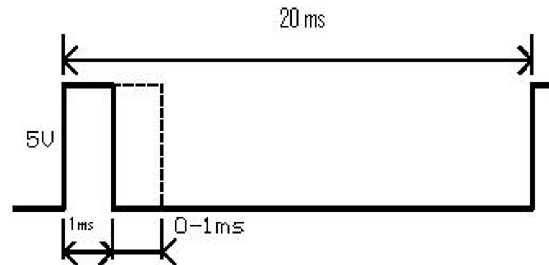



Abbildung 6.1: Graphische Darstellung eines PWM-Signals, das zur Ansteuerung von Servomotoren verwendet wird.

Da zu Beginn jedes Intervalls eine steigende Taktflanke steht, ist der Mikrocontroller so konfiguriert, dass er bei einer steigenden Taktflanke am entsprechenden Pin eine ISR (Interrupt Service Routine) aufruft. In dieser ISR wird zuerst ein Timer gestartet. Danach wird der Mikrocontroller so konfiguriert, dass er die ISR aufruft, sobald eine fallende Taktflanke erkannt wird. Beim Erkennen der fallenden Taktflanke wird dann der Timer gestoppt. Der Wert des Timers gibt Aufschluß über die Länge des vom Empfänger generierten PWM-Signals und damit über die vom Benutzer gewünschte Stellung des Servomotors. Anschließend wird der Timer mit 0 initialisiert. Um das nächste PWM-Signal erfassen zu können, wird am Ende der ISR der Mikrocontroller wieder so konfiguriert, dass er auf eine steigende Taktflanke wartet. Um das beschriebene Vorgehen bewerkstelligen zu können ist es nötig „ExternalInterruptPins“ zu verwenden. Die Lenkung wurde deshalb an PD0 (INT0) und das Gas an PD1 (INT1) angeschlossen. Das Senden beziehungsweise Erzeugen eines PWM-Siganls gestaltet sich einfacher, da vom Mikrocontroller entsprechende Hardware bereitgestellt wird. Nach Durchführung einiger Konfigurationen genügt es, die gewünschte Pulslänge in ein Register zu schreiben und der Controller übernimmt die Erzeugung des Signals. Jedoch müssen auch hier bestimmte Controllerpins verwendet werden. Die Ausgabe der Lenkstellung erfolgt über PB5 (OC1A), die der Gasstellung über PE3 (OC3A). Mit der dargestellten Vorgehensweise zum Senden und Empfangen konnte die Servostellung in 1000 Positionen aufgeteilt werden, das heißt es kann beim Senden und Empfangen die zweite Millisekunde in Abstufungen von bis zu einer Mikrosekunde unterschieden werden.

Da die Servomotoren und die verwendeten Mikrocontroller getrennte Stromversorgungen haben, wurden die Stromkreise voneinander getrennt. Zwischen Empfängermodul und Mikrocontroller beziehungsweise Mikrocontroller und Servomotoren wurden deshalb Optokoppler eingebaut. Da diese jedoch die Eigenschaft besitzen, Signale zu invertieren, muss die Software entsprechend angepasst werden. Die aufgebaute Schaltung ist in Abbildung 6.2 zu sehen. Der dazugehörige Schaltplan in Abbildung 6.3. Die Schaltung, die zum Einlesen der PWM-Signale verwendet wurde, besteht aus einem Optokoppler (6N137), einem Transistor (2N3904) und zwei Widerständen. Dabei ist das vom Empfänger kommende PWM-Signal an der Basis eines

Transistors angeschlossen und sorgt dafür, dass bei entsprechendem Spannungspegel die Kathode der LED im Optokoppler auf GND des Empfängerstromkreises liegt. Um die LED zu schützen, wurde ein Vorwiderstand eingebaut. Auf der Controllerseite des Optokopplers ist der Emitter des im Optokoppler enthaltenen Transistors mit GND des Controllers verbunden. Der Kollektor ist mit dem Eingang des Controllers verbunden und über einen 10kOhm Widerstand mit Vcc des Controllers. Wenn der im Optokoppler enthaltene Transistor nun schaltet, liegt am Microcontrollerpin „logisch 0“ ansonsten „logisch 1“ an. Der Aufbau der Schaltung ist schematisch in Abbildung 6.4 dargestellt. Die Ausgabe des PWM-Signals erfolgt wie bereits erwähnt auch über einen Optokoppler. Die Kathode der Optokoppler-LED wird über einen Vorwiderstand vom 470 Ω an den Controllerpin angeschlossen, der das PWM-Signal erzeugt. Die Kathode wird mit GND des Controllers verbunden. Je nach Pegel des PWM-Signals leuchtet die LED. Auf der Seite des Fahrzeugs ist der Transistor des Optokopplers am Emitter mit GND des Fahrzeugs verbunden. Der Collektor ist mit dem Servomotor und über einen Widerstand von 10 kΩ mit Vcc des Fahrzeugs verbunden. Je nachdem, ob der Transistor schaltet, liegt am Signaleingang des Servos also „logisch 0“ oder „logisch 1“ an. Der Aufbau der Schaltung ist in Abbildung 6.5 zu sehen.

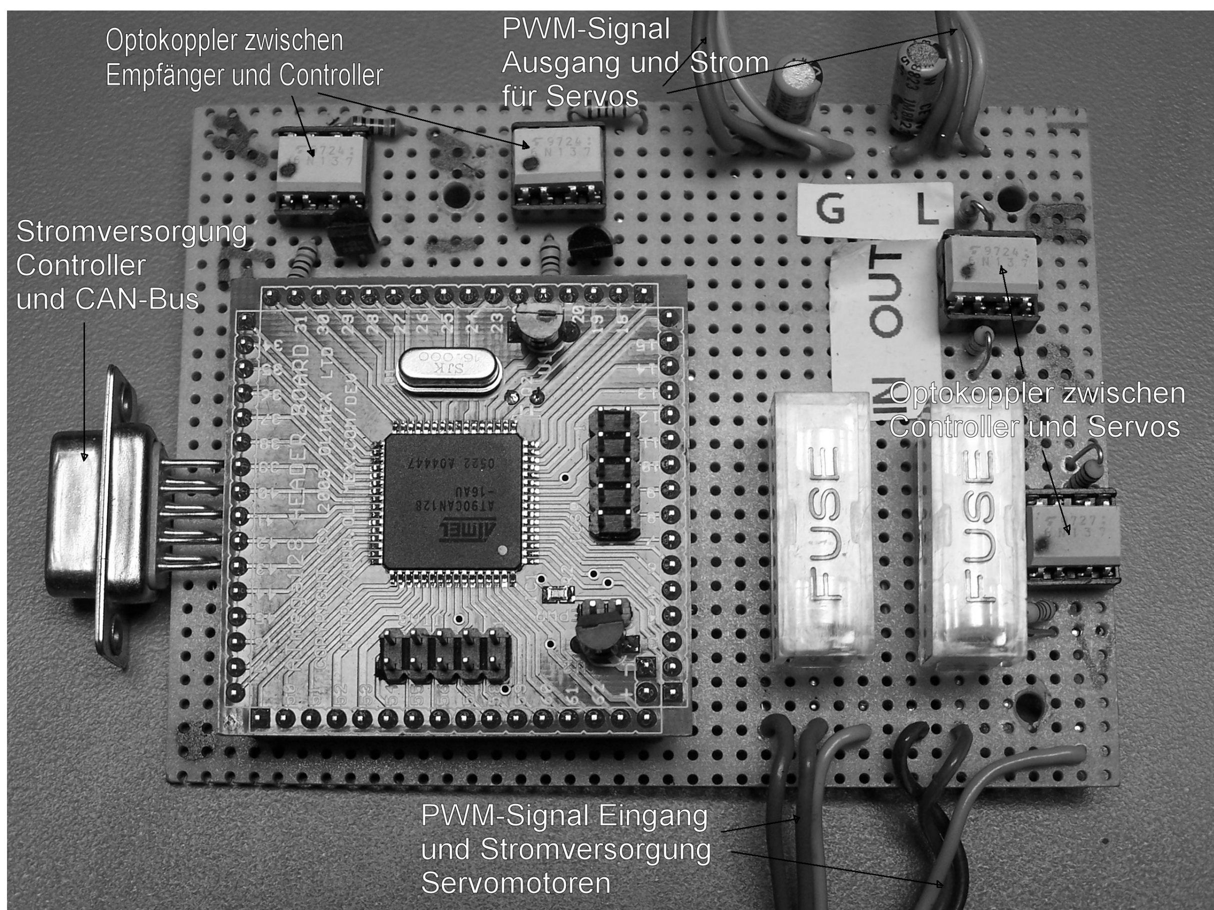



Abbildung 6.2: Aufgebaute Schaltung für das Einlesen und Ausgeben von PWM-Signalen und die Kommunikation via CAN-Bus

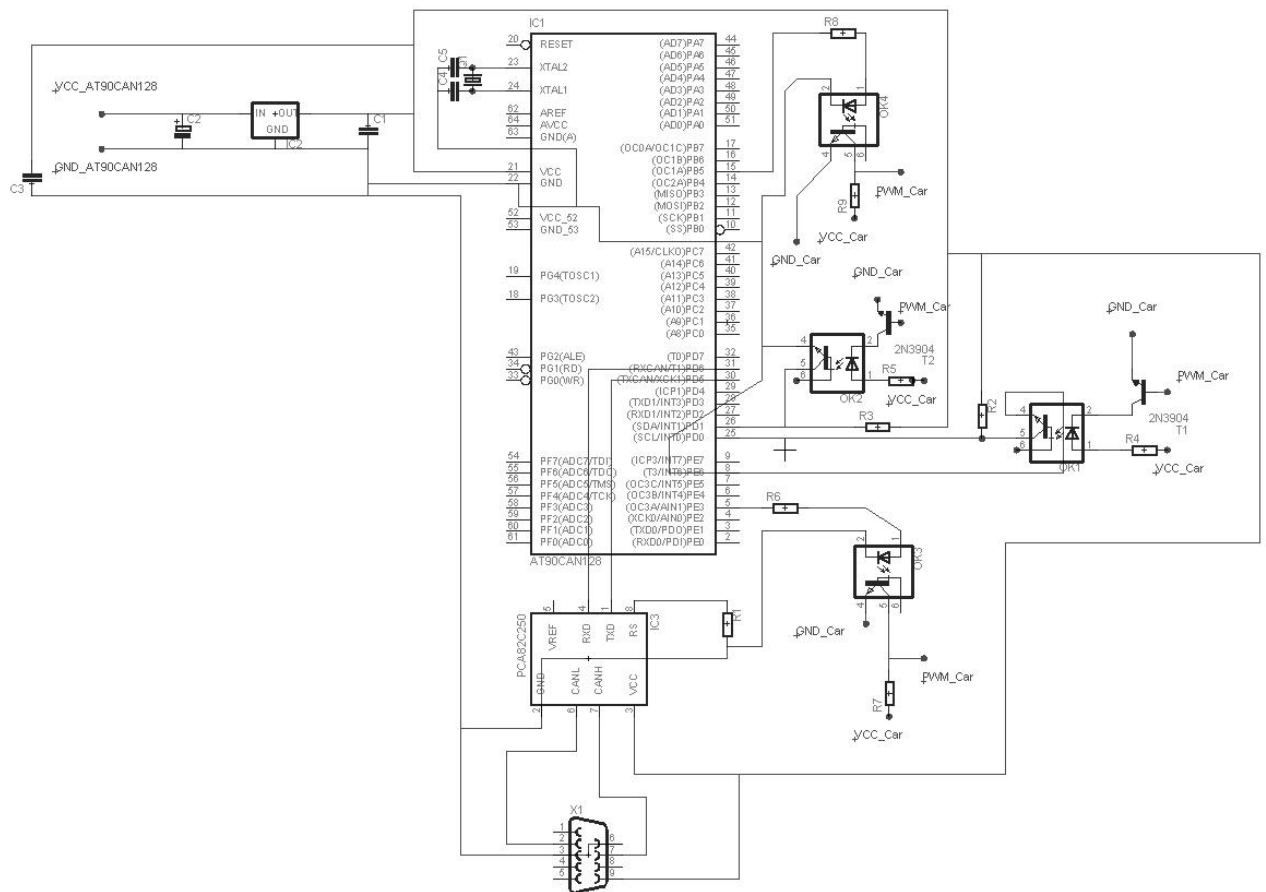



Abbildung 6.3: Schematische Darstellung der Schaltung des Behaviourcontrollers

## 6.2 CAN-Bus

Anders, als es der Name des verwendeten Controllers (AT90CAN128) vermuten lässt, kann er nicht direkt an den CAN-Bus angeschlossen werden. Es wird zusätzlich ein CAN-Bustreiber (PCA82C250) benötigt (Abbildung. 6.6). Der Controller, beziehungsweise der CAN-Bustreiber, wurden mit einem handelsüblichen D-SUB Stecker an den CAN-Bus angeschlossen. Zusätzlich zur Datenübertragung dient dieser D-SUB Stecker auch zur Stromversorgung. Der CAN-Bus wurde zum einen für die Kommunikation zwischen dem Behaviour- und Constraintcontroller verwendet und zum anderen zur Übertragung der Sensorwerte. Als Sensoren standen vier Radgeschwindigkeitssensoren zur Verfügung, die parallel im Rahmen einer Projektarbeit [Fabrice 06] aufgebaut wurden.

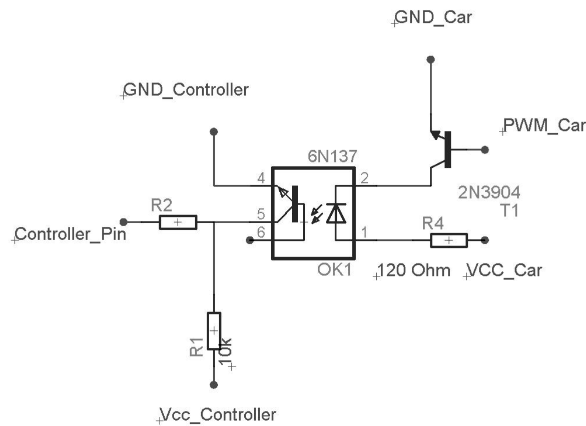



Abbildung 6.4: Schematische Darstellung der Schaltung zum Einlesen eines PWM-Signals mit Hilfe eines Optokopplers

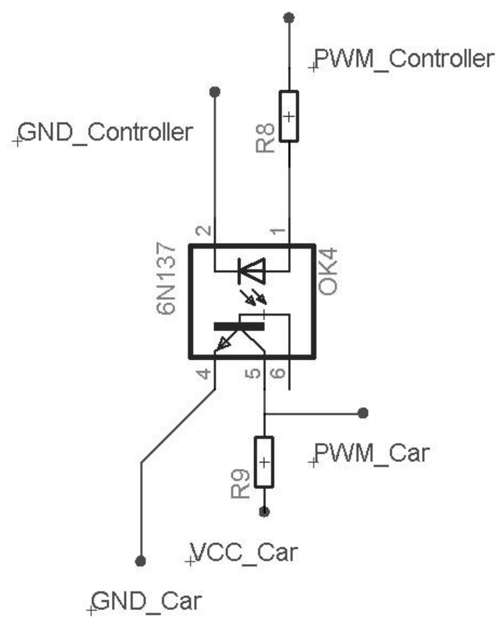



Abbildung 6.5: Schematische Darstellung der Schaltung zum Ausgeben eines PWM-Signals mit Hilfe eines Optokopplers

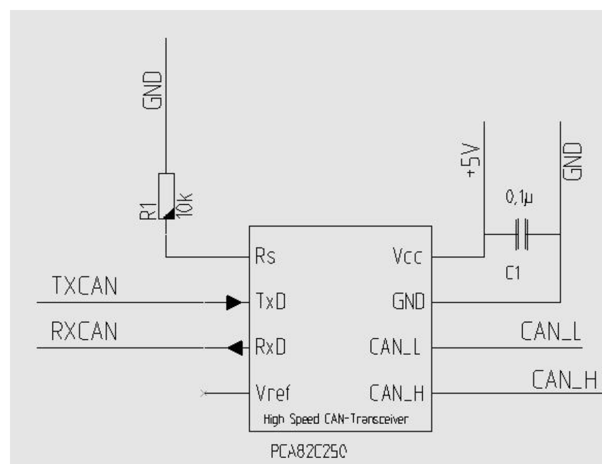



Abbildung 6.6: Schematische Darstellung der Schaltung zum Betreiben eines CAN-Bustreibers

# 7. Auswertung, Erfahrungen und Ergebnisse

Um die vorgestellte Beispielarchitektur bewerten zu können, wurde eine Reihe von Tests durchgeführt. Während dieser Tests wurden an verschiedenen Punkten Fehler injiziert. Die Fehler bestanden zum einen aus Softwarefehlern und zum anderen aus Hardwarefehlern. Als Hardwarefehler wurden Controllerausfälle simuliert, indem die entsprechenden Controller im Labor resettet oder vom CAN-Bus abgekoppelt wurden. Einerseits wurden so Ausfälle der Controller festgestellt, die für die Generierung der Sensorwerte verantwortlich sind, und andererseits Ausfälle des Constraintcontrollers. Alle getesteten Hardwarefehler wurden erkannt.

Während einer Testreihe auf der Versuchsstrecke wurde beobachtet, dass der Behaviourcontroller immer wieder, scheinbar unmotiviert, die Eingriffe des Constraintcontrollers ignoriert. Im Labor wurde das zuvor nie beobachtet. Wie sich nach einer genaueren Untersuchung des Phänomens herausstellte, gab es einen Wackelkontakt im CAN-Busstecker des Constraintcontrollers. Durch die Vibrationen des Motors kam dieser Fehler erst während der Testfahrten zum Tragen. Durch diesen Wackelkontakt wurden die Signale des Constraintcontrollers verfälscht und/oder gingen verloren. Nachdem die Software des Behaviourcontrollers dies durch Fehlen des Heartbeatsignals erkannt hat, wurden, wie gewünscht, die Vorgaben des Constraintcontrollers ignoriert, da er als fehlerhaft eingestuft wurde. Das System arbeitete danach ohne globale SRS weiter und erfüllte seine Aufgaben. Ein weiterer erst bei den Testfahrten aufgetretener Fehler ist, dass auf Grund mechanischer Spannungen es während einer Testfahrt zum Ausfall eines Sensorcontrollers kam. Auch diesen Ausfall erkannte das System und reagierte mit dem gewünschten Verhalten. Der für die Sensorüberwachung verantwortliche Teil des globalen SRS hat den Ausfall erkannt und an das SRS des ASR weitergeleitet. Das SRSASR reagierte darauf, wie gewünscht, mit der Abschaltung des ASR. Das Abschalten des ASR hatte, wie spezifiziert, zur Folge, dass alle Fahrer unterstützenden Systeme der darüberliegenden Schichten ebenfalls abgeschaltet wurden. Während einer anderen Testfahrt lieferte ein Radsensor auf Grund einer Beschädigung falsche Werte. Dieser Fehler wurde

wie erwartet erkannt, und es wurde genau wie beim Ausfall eines Sensorcontrollers reagiert.

Das Injizieren von Softwarefehlern erfolgte immer unmittelbar vor einem SRS, wie in Abbildung 7.1 dargestellt. Dabei wurden Fehler injiziert, welche die in Kapitel 5 angeführten Spezifikationen verletzen. Das Einfügen der Fehler erfolgte im Labor manuell. Es wurde zu einem beliebigen Zeitpunkt eine CAN-Nachricht generiert, die einen Fehler auslöst. Während der Testfahrten bestand diese Möglichkeit jedoch nicht, da das Einstreuen der CAN-Nachrichten bisher nur per Kabel möglich ist. Aus diesem Grund wurden während der Testfahrten die Fehler automatisch nach einer vordefinierten Zeitspanne generiert. Die erzeugten Fehler wurden ausnahmslos erkannt. Zuerst wurden Fehler erzeugt, die vom globalen SRS erkannt werden sollen. Der erste erzuegte Fehler war eine dauerhaft zu hohe Gasstellung des Gesamtsystems. Der Fehler wurde erkannt und das Fahrzeug automatisch angehalten. Als nächstes wurde ein Geschwindigkeit Simuliert, die höher ist als ein zuvor festgelegter Schwellwert. Nach einer vor definierten Zeitspanne reagierte das globale SRS und bremste auch in diesem Fall das Fahrzeug ab. Der Fehler wurde sowohl mit unterschiedlichen Geschwindigkeiten als auch mit unterschiedlichen Zeiten erkannt. Ein anderer Test simulierte eine dauerhaft gleiche Gasstellung, was als Fehler angesehen wird, sofern die Gasstellung keinem zulässigen Maximalausschlag entspricht. Das Auto reagierte auch in diesm Fall wie gewünscht mit der Schließung der Bremsen. Ein Teil des globalen SRS soll die Gasstellung auf einen maximalen Wert beschränken. Im Labor wurden alle Überschreitungen dieses Schwellwertes bemerkt, und die Systemausgabe auf das vorgegebene Maximum reduziert.

Die SRS der einzelnen Verhalten wurden im Anschluß getestet. Es wurde mit dem SRS der ASR begonnen. Es wurde simuliert, dass das ASR mehr Gas vorgibt als die Eingabe. Weiterhin wurde simuliert, dass die ASR länger für das Einregeln benötigt als spezifiziert. Ein dritter injizierter Fehler bestand darin, dass noch Schlupf vorhanden ist, der Benutzer die Gasstellung erhöht, und die ASR dann mehr Gas gibt. Dieses Verhalten wurde vorher als Fehler spezifiziert. Die Fehler wurden alle vom SRS des ASR erkannt. Danach wurde das SRS des Kurvenassistenten getestet. Es wurde zuerst ein Eingreifen des Kurvenassistenten simuliert, obwohl die Lenkstellung das noch nicht rechtfertigt. Als nächstes wurde erkannt, dass der Kurvenassistent mehr Gas gibt als seine Vorgabe. Es wurde auch ein zu strakes Eingreifen des Kurvenassisten erkannt. Fehler die beim durchlaufen der Kennlinie erkannt wurden sind Überschreiten eines maximalen Gradienten, zu weite Sprünge beim durchlaufen der Kennlinie, zu lange Dauer beim durchlaufen und eingreifen der Kennlinie obwohl sie nicht aktiv sein dürfte.

Hilfreich war der Einsatz der SRS beim Entwickeln des Systems. Es wurden häufig Fehler dadurch bemerkt, dass ein SRS aktiviert wurde. Dabei war der Fehler nur selten im SRS zu finden, sondern meist im dazugehörigen Verhaltensmodul. Mit Hilfe von globalen SRS konnte z.B. ein Fehler beim Einlesen der PWM-Signale bemerkt werden. Wenn die Stellung der Fernbedienung schlagartig auf Maximum verändert wurde, dann wurde gelegentlich, jedoch nicht immer, nicht der Maximalwert eingelesen. Dadurch kam es zu einer konstanten Gasstellung, die jedoch nicht (absichtlich) von Systemkomponenten erzeugt worden war. Wie in Kapitel 5 angeführt, wird das als Fehler erkannt und das Fahrzeug infolgedessen abgebremst. Auf Grund der SRS

wurde aber auch eine Vielzahl kleinerer Fehler, die nur gelegentlich auftreten, wie zum Beispiel ein Vergleich auf „kleiner", anstatt „kleiner gleich", entdeckt. In diesen Fällen kann man die Verwendung der SRS als eine Art „extended debugging"-Mechanismus bezeichnen, bei dem die ersten Tests der Softwarekomponenten schon während der Entwicklungszeit vorgenommen werden. Da ein SRS über CAN-Bus mitteilen kann, warum es aktiviert wurde, kann dann auch direkt überprüft werden, ob es sich tatsächlich um ein fehlerhaftes Verhalten handelt oder ob ein Fehler im SRS vorliegt. Falls im SRS kein Fehler vorhanden ist, wird das Verhalten auf den entsprechenden Fehler untersucht und gegebenenfalls verbessert.

Zu bemerken ist, dass der Aufwand, der für die Verwendung der SRS notwendig ist, größer ausgefallen ist, als erwartet. So sind zum Beispiel sieben der insgesamt elf Arbiterknoten lediglich auf Grund der SRS eingefügt worden. Auf die Codegröße wirkt sich jedoch nicht die Verwendung der Arbiterknoten aus, sondern die verwendeteten SRS selbst. Das aufgebaute Framework benötigt alleine 8180 Byte. Nach Implementierung der Verhalten benötigt die Software 13098 Byte, das heißt, die Verhalten benötigen 4918 Byte. Nach Einfügen der SRS wächst die Codegröße weiter auf 23737 Byte an. Das heißt, die SRS machen ca. 69 Prozent des Gesamtcodes aus (ohne Berücksichtigung des Frameworks). Im Vergleich dazu fielen etwa 40 Prozent der Entwicklungszeit, nach Fertigstellung des Frameworks, auf die Implementierung der SRS ab. Zusätzlich wurde Zeit benötigt, um die SRS in die Architektur einzubinden und zu konzeptionieren. Alles in allem verteilte sich so die Entwicklungszeit zur einen Hälfte auf die eigentlichen Verhalten und zur anderen Hälfte auf die SRS. Unbedenklich ist die Verwendung der SRS jedoch bezüglich der Echtzeitfähigkeit, da alle SRS auf einen eigenen Controller ausgelagert werden können. Dadurch fällt die durch die SRS erzeugte Extralaufzeit auf dem Behaviourcontroller nicht ins Gewicht. Die Komplexität konnte durch die Benutzung der Subsumptionsarchitektur auf einem überschaubaren Maß gehalten werden. Ein Teil dieses Effektes ging jedoch durch die Verwendung der Selfchecking-Software wieder verloren. Das war jedoch notwendig, um den automobilen Anforderungen gerecht werden zu können.

Abschließend kann festgehalten werden, dass der vorgestellte Softwareansatz in der getesteten Architektur wie gewünscht funktioniert hat und auf Grundlage der gewonnenen Erfahrungen prinzipiell für das Anwendungsgebiet geeignet ist.

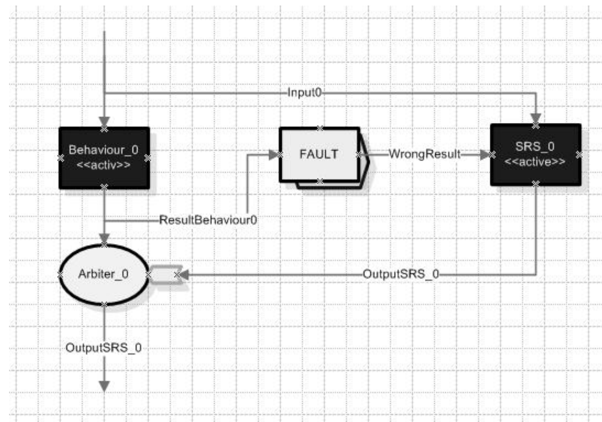



Abbildung 7.1: Darstellung des Faultinjectionmechanismusses

# Literaturverzeichnis


[Arkin 98] R. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[Balkhis Abu Bakar ] Tomasz Janowski Balkhis Abu Bakar. Automated Result Verification with AWK. In *??*

[Barbacci95 ] Barbacci95. Quality Attributes (CMU/SEI-95-TR-021). In *??*, ?? 95.

[Blum ] Manuel Blum. Designing programms to check their work. In *??*

[Brooks 85] Rodney A Brooks. A Robust Layered Control System For A Mobils Robot. In *??*, September 1985.

[Correia 95] Luìs Correia, A. Steiger-Garcao. A Useful Autonomous Vehicle with a Hierarchical Behavior Control. In *??*, ?? 1995.

[Fabrice 06] Aufbau und Implementierung von Radgeschwindigkeitssensoren, 2006.

[Hal Wasserman ] Manuel Blum Hal Wasserman. Software Reliability via Run-Time Reslt-Checking. In *??*

[Jones 04] Joseph L. Jones. *Robot Programming A Practical Guide to Behaviour-Based Robotics*. McGraw-Hill, 2004.

[Kerkow 06] Usability Challenges im Automotive: ein Szenario. Persönliches Gespräch, 2006.

[Kosecka 93] Jana Kosecka, Ruzena. Discrete Event Systems for Autonomous Mobile Agents. In *Proceedings Intelligent Robotic Systems 1993 Zakopane*, July 1993.

[Lyu 95] Michael R. Lyu. *Software Fault Tolerance*. Wiley, 1995.

[Mataric a] Maja J Mataric. Behavoir-Based Control: Examples from Navigation, Learning, and Group Behavior. In *??*, ?? ??

[Mataric b] Maja J Mataric. Learning in Behavior-Based Multi-Robot Systems: Policies, Models, and Other Agents. In *??*, ?? ??

[N. Schneidewind 98] A. Nikore M.R. Lyu J. Musa N. Schneidewind, J-C. Laprie, W. Everett. Issues in the Next Generation of dependability Standards. In *Proceedings Ninth International Symposium on Software Reliability Engineering (ISSRE'98) pp.101-104*, November 1998.

[Paolo Traverso ] Piergiorgio Bertoli Paolo Traverso. Mechanized result verification: an industrial application. In *??*

[Rombach 06] Dieses Potential ist den meisten noch gar nicht bewusst. Rheinpfalz 2006 Ausgabe: ??, 2006.

[Rosenblatt 97] J. Rosenblatt. DAMN: A Distributed Architecture for Mobile Navigation. In *Journal of Experimental and Theoretical Artificial Intelligence Vol.9*, ?? 1997.

[s. s. Yau ] s. s. Yau, R. C. Cheung. Design of self-checking software. In *??*

[Safety and relyability engineering 06] Lecture: Safety and relyability engineering Chapter: Terminology. `http://www-i11.informatik.rwth-aachen.de/Safety+and+Reliability+Engineering&bl.html`, 2006.

[Software Fault Tolerance 98] Software Fault Tolerance. `www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance`, 1998.