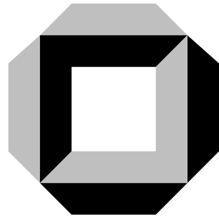

Universität Karlsruhe
Institut für Rechnerentwurf
und Fehlertoleranz
Prof. Dr.-Ing. D. Schmid

Implementierung eines Entwurfs- und Verifikationswerkzeuges für die synchrone Programmiersprache PURR

Diplomarbeit

von

Jens Zimmermann

Betreuer: Prof. Dr.-Ing. D. Schmid
Betreuender Mitarbeiter: Dr.rer.nat. K. Schneider

Tag der Anmeldung : 01.03.1999
Tag der Abgabe: 31.08.1999

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die im Literaturverzeichnis aufgeführten verwendet habe.

Karlsruhe, 31. August 1999

Inhaltsverzeichnis

1 Einleitung **5**

1.1 Motivation . . . 5

1.2 Aufgabenstellung . . . 8

1.3 Gliederung . . . 9

2 Grundlagen **11**

2.1 Beschreibungssprachen reaktiver Systeme . . . 11

2.2 ESTEREL . . . 12

2.3 PURR . . . 13

2.3.1 Datentypen von PURR . . . 13

2.3.2 Die Ausdrücke in PURR . . . 15

2.3.3 Die Anweisungen in PURR . . . 16

2.4 Verifikation reaktiver Systeme . . . 24

2.5 CTL . . . 30

2.6 Modellprüfung . . . 37

2.7 Binäre Entscheidungsdiagramme . . . 40

3 Implementierung **45**

3.1 Einlesen eines PURR-Programms . . . 45

3.1.1 Yacc . . . 46

3.1.2 Lex . . . 46

3.1.3 Erzeugen eines Syntaxbaumes zu einem PURR-Programm . . . 47

3.2 Entfernen der Redundanz aus einem PURR-Programm . . . 48

3.3 Die Transitionsrelation eines PURR-Programms . . . 56

3.3.1 Die Semantik von PURR . . . 56

3.3.2 Die Semantik der Ausdrücke in PURR . . . 58

3.3.3 Die Berechnung des Kontrollflusses . . . 59

3.3.4 Die Berechnung des Datenflusses . . . 69

3.3.5 Kombination von Kontroll- und Datenfluss . . . 73

3.4 Die Modellprüfung . . . 74

3.5 Aufrufsyntax des Werkzeugs 85
3.6 Experimente 86

4 Schlussfolgerungen und Ausblick 93
4.1 Schlussfolgerungen 93
4.2 Ausblick und Erweiterungsmöglichkeiten 93

A Die Grammatik des Parsers 95

Abbildungsverzeichnis

2.1 Kripke Struktur zur Systemmodellierung 31
2.2 Die CTL-Operatoren 35
2.3 Ablauf einer Modellprüfung 38
2.4 Kofaktorisierung einer Funktion f 41
2.5 Vollständiger Baum der Funktion $f = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ 41
2.6 Entfernen redundanter Entscheidungsknoten 42
2.7 Reduzierte Darstellung der Funktion aus Abbildung 2.5 42
2.8 Transition 44

3.1 Knoten für die Sequenz 47
3.2 Syntaxbaum des Programms P_2 48
3.3 Redundante Ersetzung der Anweisung ***halt*** 49
3.4 Redundante Ersetzung der Anweisung ***sustain* x** 49
3.5 Redundante Ersetzung der Anweisung ***await* x** 49
3.6 Redundante Ersetzung der Anweisung ***abort*** S_1 ***when*** x ***do*** S_2 50
3.7 Redundante Ersetzung der Anweisung ***every*** x ***do*** S 50
3.8 Redundante Ersetzung der Anweisung ***loop*** S ***each*** x 51
3.9 Redundante Ersetzung der Anweisung ***sustain*** x ***after 1*** 51
3.10 Redundante Ersetzung der Anweisung ***abort*** S ***when case*** $EXPR_1$ ***do*** S_1 ***case*** $EXPR_n$ ***do*** S_n 52
3.11 Redundante Ersetzung der Anweisung ***await case*** $EXPR_1$ ***do*** S_1 ***case*** $EXPR_n$ ***do*** S_n 53
3.12 Redundante Ersetzung der Anweisung ***present case*** $EXPR_1$ ***do*** S_1 ***case*** $EXPR_n$ ***do*** S_n ***else do*** S_{n+1} 54
3.13 Syntaxbaum und redundante Ersetzungen im Programm P_3 55
3.14 Kontrollfluss des Programms P_5 69
3.15 Verhalten des Programms P_5 74
3.16 CTL-Modellprüfung ohne Fairness 75
3.17 Die Funktion CheckEG(φ) 76
3.18 Die Funktion CheckEF(φ) 76
3.19 Die Funktion CheckEU(φ, ψ) 77

3.20 Die Funktion CheckESU(φ, ψ) 77
3.21 Die Funktion CheckEB(φ, ψ) 78
3.22 Die Funktion CheckESB(φ, ψ) 78
3.23 Die Funktion CheckEW(φ, ψ) 79
3.24 Die Funktion CheckESW(φ, ψ) 79
3.25 CTL-Modellprüfung mit Fairness 80
3.26 Die Funktion CheckfairEG(φ) 81
3.27 Die Funktion CheckfairEF(φ) 81
3.28 Die Funktion CheckfairEU(φ, ψ) 82
3.29 Die Funktion CheckfairESU(φ, ψ) 82
3.30 Die Funktion CheckfairEB(φ, ψ) 83
3.31 Die Funktion CheckfairESB(φ, ψ) 83
3.32 Die Funktion CheckfairEW(φ, ψ) 84
3.33 Die Funktion CheckfairESW(φ, ψ) 84
3.34 Kontroll- und Datenfluss des Programms P_7 87

Kapitel 1

Einleitung

1.1 Motivation

Unser tägliches Leben ist in einem starken Ausmaß durch technische Systeme geprägt. Mit diesen kommen wir in den unterschiedlichsten Lebensbereichen in Kontakt. Als Beispiele lassen sich hier u.a. Geldautomaten, Mobiltelefone, Informationssysteme auf Flughäfen oder an Bahnhöfen, Steuerungen in Fahrzeugen oder Geräte in der Medizin oder auch viele Haushaltsgeräte nennen. Diese Systeme enthalten eingebettete Systeme, d.h. Systeme, die aus Soft- und Hardware bestehen und die auf Eingaben von außen innnerhalb von sehr kurzen Zeitspannen reagieren müssen. Eingebettete Systeme, die solche Anforderungen an ihre Reaktionszeit erfüllen werden meist als reaktive Systeme bezeichnet.

Der Entwurf dieser Systeme geschieht in zunehmendem Maße mit Hilfe von Programmiersprachen auf einer relativ hohen Ebene. Die Umwandlung solcher Programme auf niedrigere Ebenen wie etwa Assemblerprogramme oder Gatterrepräsentation wird dann von geeigneten Entwurfswerkzeugen automatisch vorgenommen. Da diese automatisch erstellten Ergebnisse annähernd so gut sind wie manuell erstellte Entwürfe, aber weitaus kostengünstiger und flexibler entworfen werden können, ist man nicht mehr so stark auf den Entwurf auf niedrigeren Ebenen angewiesen, was auch die Anzahl der möglichen Fehler minimiert. Entwurf auf niedrigeren Ebenen wird in den meisten Fällen nur noch bei Optimierungen benötigt. Die oben genannten Programmiersprachen besitzen einen großen Befehlsumfang und eignen sich daher zum Entwurf komplexer Systeme. Generell ist der Entwurf von reaktiven Systemen, die kontinuierlich auf Eingaben aus der Umgebung reagieren müssen, eine anspruchsvolle Aufgabe bei der leicht Fehler auftreten können. Für den Entwurf von reaktiven Systemen ist man daher auf spezielle Programmiersprachen wie z.B. ESTEREL oder PURR angewiesen. Diese synchronen Sprachen haben zwar im Vergleich zu anderen Programmiersprachen wie VHDL oder Ada einen kleineren Befehlsumfang, besitzen dafür jedoch eine klar definierte Semantik, was auch die Entwicklung von Verifikationswerkzeugen vereinfacht.

Im Allgemeinen ist man darauf angewiesen, dass technische Systeme einwandfrei funktionieren. Während man bei einem Mobiltelefon den Abriss des Gesprächs wohl noch hinnehmen kann, ist das Auftreten eines Fehlers während einer finanziellen Transaktion an der Börse schon besonders ärgerlich, da dies unter Umständen eine Menge Geld kosten kann. In der Medizin kann das Auftreten eines Fehlers möglicherweise sogar das Leben eines Menschen bedrohen. Man ist also im Allgemeinen sehr stark daran interessiert, keine fehlerhaften technischen Systeme einzusetzen, um das Risiko eines Schadens für Personen oder Sachen möglichst klein zu halten.

Auch aus finanziellen Gesichtspunkten ist das Auftreten von Fehlern nicht wünschenswert. So kostet das Beheben eines Fehlers an einem technischen System oft eine Menge zusätzlicher Entwurfszeit und erfordert oft sogar einen kompletten neuen Entwurf. Wird der Fehler erst nach der Markteinführung bekannt, kann das Auftreten des Fehlers zu einem erheblichen Imageverlust in der Öffentlichkeit führen. Ein gutes Beispiel hierfür ist die Firma Intel, die durch den Fehler im Pentium-Prozessor nicht nur finanziellen Schaden erlitten hat (der finanzielle Schaden allein wird auf ca. 100 Millionen Dollar geschätzt).

Bei der Komplexität heutiger technischer Systeme ist das Vermeiden von Fehlern jedoch nicht so einfach. Mikroprozessoren bestehen mittlerweile aus mehr als 5 Millionen Transistoren und bei der nächsten Generation der Mikroprozessoren wird die Zahl der Transistoren noch größer sein. Auch die Software, die in den eingebetteten Systemen verwendet wird, nimmt in ihrem Umfang deutlich zu. Heutzutage sind die verwendeten Programme noch einige KBytes groß, doch in der Zukunft wird es auch hier Programme geben, deren Grösse im MByte-Bereich anzusiedeln ist.

Die möglichen Fehler, die an einem technischen System auftreten können, lassen sich grob in vier Klassen einteilen: Spezifikations-, Entwurfs-, Fertigungs- und Betriebsfehler.

- *Spezifikationsfehler* sind Fehler, die schon vor der Implementierung bestehen. In einem solchen Fall ist die Beschreibung dessen, was das System leisten soll, fehlerhaft. Gründe dafür können z.B. eine nicht exakte oder sogar widersprüchliche Beschreibung des Verhaltens oder Fehler bei der Umsetzung der umgangssprachlichen Beschreibung in eine formale Beschreibung sein.

- *Entwurfsfehler* entstehen bei der Umsetzung der Spezifikation in eine programmiersprachliche Beschreibung.

- *Fertigungsfehler* sind der Grund dafür, dass auch korrekt entworfene Systeme nicht wie gewünscht funktionieren. Bei Schaltungen treten Fertigungsfehler z.B. durch Defekte in den Belichtungsmasken für einzelne Ebenen des Chips auf. Gefertigte Chips werden

daher durch Fertigungstests auf Fehler geprüft. Dies geschieht mit Hilfe von Testmusterfolgen, die an die Schaltung angelegt werden und mit denen Fertigungsfehler entdeckt werden können. Hierbei gibt es für bestimmte Klassen von Fertigungsfehlern bestimmte Testmusterfolgen, mit deren Hilfe man diese Fehler entdecken kann. Bei der Software entsprechen Fertigungsfehler einer nicht korrekten Übersetzung in Maschinensprache. Hier bemüht man sich um verifizierte Compiler.

- *Betriebsfehler* sind nach der Fertigung oftmals noch gar nicht vorhanden, sondern treten erst im laufenden Betrieb nach einiger Zeit auf. Dies kann durch Alterung, Verschleiß, Bedienungs- oder Wartungsfehler geschehen.

Entwurfs-, Fertigungs- und Spezifikationsfehler entstehen vor dem Einsatz des Systems und lassen sich also rechtzeitig erkennen und vermeiden. Dies gilt jedoch nicht für Betriebsfehler, so dass hier durch Einsatz von Fehlertoleranzmaßnahmen dafür gesorgt werden muss, dass sich solche Fehler nicht negativ auf die Funktion des Systems auswirken.

Fertigungs- und Betriebsfehler treten erst nach der Fertigung einer Schaltung auf und sind unabhängig davon, ob der Entwurf oder die Spezifikation der Schaltung korrekt war. Die meisten Fertigungsfehler lassen sich durch Funktionstests nach der Fertigung der Schaltung entdecken und Betriebsfehler lassen sich durch Fehlertoleranzmaßnahmen entdecken. Entwurfs- oder Spezifikationsfehler sind Fehler, die während der Entwicklung eines Systems entstehen und oftmals nicht offensichtlich sind.

Um solche Fehler vor der Produktion einer Schaltung zu entdecken, muss man beweisen, dass das entwickelte System auch die Anforderungen der Spezifikation erfüllt. Da heutige Systeme sehr komplex sein können, ist es von großem Vorteil, wenn der Nachweis der Korrektheit eines Entwurfs gegenüber seiner Spezifikation vollautomatisch oder zumindest rechnergestützt erfolgen kann. Ein vollautomatischer Beweis ist sehr wünschenswert, da dies meist mit geringeren Kosten erbracht werden kann.

Die formale Verifikation befasst sich mit dem Nachweis der Abwesenheit bzw. dem Entdecken von Entwurfsfehlern. Daher wird ein mathematischer Beweis geführt, der zeigt, dass ein System alle durch die Spezifikation festgelegten Anforderungen erfüllt. Das Führen eines solchen Beweises ist vor allem dann wichtig, wenn das System in Umgebungen eingesetzt wird, in denen es für die Sicherheit von Menschenleben benötigt wird. Solche Beweise könnte man theoretisch auch durch eine vollständige Simulation aller möglichen Eingangsbelegungen und internen Zustände eines Systems führen. Da die Zahl der möglichen Eingangsbelegungen jedoch exponentiell mit der Zahl der Eingänge und der Zahl der internen Zustände eines Systems wächst, kann eine solche vollständige Simulation schnell zu einem nicht mehr praktikablen Zeitaufwand führen. Man benötigt also andere Methoden, um die Korrektheit einer Implementierung gegenüber einer Spezifikation zu beweisen. Hierzu bedient man sich formaler Logiken

wie z.B. μ-Kalül, ω-Automaten oder temporale Logiken. Für diese Logiken sind Algorithmen bekannt, welche entscheiden können, ob für ein Programm P die Eigenschaft ϕ gilt oder nicht. Im Fehlerfall kann ein Gegenbeispiel erzeugt werden, welches den Fehler demonstriert. In der Regel lässt sich mit diesen Verfahren die Korrektheit einer Implementierung gegenüber einer Spezifikation mit einem wesentlich geringeren Zeit- und Ressourcenaufwand beweisen. Die Hardware-Verifikation kann damit also auch zur Senkung der Produktionskosten beitragen, indem Entwurfsfehler relativ früh im Entwurf des Systems entdeckt werden.

1.2 Aufgabenstellung

Zum Entwurf werden, wie oben beschrieben, immer häufiger Programmiersprachen auf relativ hoher Ebene eingesetzt. Die bekannten Verifikationsverfahren beziehen sich jedoch nur auf endliche Automaten bzw. Kripke Strukturen. Aufgabe dieser Arbeit war daher, ein Programm in einer solchen Programmiersprache in einen endlichen Automaten zu übersetzen. Damit kann dann eine Verifikation stattfinden, oder auch ein weiterer Entwurf vorgenommen werden. Man beachte, dass die Programme in der Regel aus mehreren nebenläufigen Threads bestehen, der endliche Automat jedoch quasi nur aus einem einzigen Thread.

Als Ausgangspunkt wurde die Programmiersprache PURR, welche am Institut für Rechnerentwurf und Fehlertoleranz der Universität Karlsruhe entwickelt wurde, ausgewählt [10, 11]. Mit dieser Sprache lassen sich nebenläufige reaktive Systeme (engl. koop. Threads) beschreiben. PURR ist eine Erweiterung von ESTEREL [2], und kann wie ESTEREL nach Transformation auf endliche Automaten zur Software- und Hardware-Synthese verwendet werden. PURR besteht aus einer kleinen Menge von Anweisungen, mit einer klar definierten Syntax und einer klar definierten Semantik.

Ziel dieser Arbeit war es nun, für diese Programmiersprache mit Hilfe der Semantik eine Umsetzung vom Programmtext in einen endlichen Automaten zu schaffen, der genau das Verhalten des vorliegenden Programms modelliert. Hierzu wird der Programmtext zuerst in eine geeignete Datenstruktur eingelesen, wonach dann innerhalb der Datenstruktur noch einige Transformationen durchgeführt werden, um die Erzeugung des Automaten zu vereinfachen. Nachdem diese Transformationen durchgeführt wurden, wird dann zuerst der Kontrollfluss des Programms erzeugt. Hierzu werden im Einzelnen Bedingungen für das Eintreten des Kontrollflusses in eine Anweisung, für die Fortführung einer Anweisung sowie für die Terminierung der Anweisungen erzeugt, die dann zu der Repräsentation des Kontrollflusses zusammengesetzt werden. Diese Bedingungen lassen sich ausnahmslos als aussagenlogische Formeln beschreiben. In den letzten Jahren wurden effiziente Datenstrukturen zur Repräsentation von aussagenlogischen Formeln in Normalformen wie BDD's [3] entwickelt. Es sind auch effiziente Programmpakete für diese BDD's im Internet erhältlich. Grundlage dieser Arbeit ist das Paket von R. K. Ranjan,

J. V. Sanghavi, R. K. Brayton und A. Sangiovanni-Vincentelli von der Universität Berkeley in Californien [15]. Aufgrund einer Untersuchung von E. Sentovich [17] stellte sich dieses BDD-Paket als das effizienteste heraus. Damit unterscheidet sich die Art der Übersetzung deutlich von den bisher verwendeten Techniken[2, 4].

Danach wird dann eine BDD-Repräsentation des Datenflusses erzeugt, die mit der des Kontrollflusses zu der BDD-Repräsentation der Kripke Struktur des Programms kombiniert wird. Eine genaue Beschreibung dieses Vorgangs befindet sich in Abschnitt 3.3. Ein weiteres Ziel der Arbeit war die Implementierung eines Modellprüfers für die temporale Logik CTL. Hierzu ist es nötig im Programmtext die zu prüfende Eigenschaft durch eine CTL-Formel zu spezifizieren. Der Modellprüfer testet dann, ob die spezifizierte Eigenschaft in der vorher erzeugten Kripke Struktur gilt.

1.3 Gliederung

Das folgende Kapitel befasst sich mit den Grundlagen, die für diese Arbeit benötigt werden. Hier wird kurz auf synchrone Programmiersprachen insbesondere auf ESTEREL eingegangen. Daran schliesst sich ein Abschnitt an, in dem die Syntax der synchronen Programmiersprache PURR erläutert wird. Danach folgen Abschnitte über Verifikation, die temporale Logik CTL sowie deren Modellprüfung und binäre Entscheidungsdiagramme.

Das dritte Kapitel befasst sich mit der Implementierung des Verifikationswerkzeuges. Der erste Abschnitt dieses Kapitels behandelt das Einlesen eines PURR-Programms sowie die Erzeugung des zugehörigen Syntaxbaumes. Danach folgt ein Abschnitt, der einige Transformationen behandelt, die vor der Weiterverarbeitung auf dem Syntaxbaum ausgeführt werden müssen. Der dritte Abschnitt befasst sich mit der Erzeugung der Transitionsrelation aus dem zuvor aufgebauten Syntaxbaum. Hier wird unter anderem die Semantik der Anweisungen in PURR sowie die Berechnungen des Kontroll- und des Datenflusses sowie deren spätere Kombination erläutert. Der vierte Abschnitt befasst sich mit der Implementierung des Modellprüfers für PURR-Programme. Danach folgen zwei Abschnitte, in denen die Aufrufsyntax erläutert sowie einige Experimente vorgestellt werden.

Im vierten Kapitel befinden sich Schlussfolgerungen zu dieser Arbeit sowie ein Abschnitt über Erweiterungsmöglichkeiten der hier vorgestellten Implementierung.

Kapitel 2

Grundlagen

2.1 Beschreibungssprachen reaktiver Systeme

Zur Beschreibung von eingebetteten Systemen werden in zunehmendem Maße synchrone Sprachen verwendet. Diese Sprachen erlauben, Parallelität auf der Ebene leichtgewichtiger Prozesse zu beschreiben. Die Anweisungen synchroner Programmier- und Modellierungssprachen unterteilen sich in zeitverbrauchende und zeitlose. Jede zeitverbrauchende Anweisung benötigt ein Vielfaches einer fest gewählten logischen Zeiteinheit, so daß alle Prozesse synchron zueinander arbeiten. Die Kommunikation zwischen den Prozessen erfolgt in der Regel durch Senden von global sichtbaren Signalen, die auch Daten tragen können. Dieses elementare Kommunikationsprinzip erlaubt auch die Implementierung anderer Kommunikationsarten wie etwa Datenaustausch über Kanäle oder gemeinsame Daten. Graphische synchrone Sprachen wie z.B. Statecharts, Argos und SyncCharts entstanden früh aus der Modellierung von endlichen Automaten. Im Gegensatz zu konventionellen endlichen Automaten können hier nebenläufige kommunizierende Automaten beschrieben werden, deren Zustände selbst wieder Automaten enthalten können. Insbesondere Statecharts haben für den Entwurf eingebetteter Systeme bereits eine weite Verbreitung in der Industrie gefunden.

Ferner wurden auch aus dem Bereich der Datenflusssprachen synchrone Sprachen entwickelt. Die derzeit wichtigsten Vertreter sind Lustre und SIGNAL. Diese Sprachen fassen Variablen und Terme als Datenströme auf. Programme sind prinzipiell Gleichungssysteme, die die Datenstöme rekursiv in Abhängigkeit von den momentanen sowie den unmittelbar vorangegangenen Werten bestimmen. Diese Sprachen verwenden Konzepte aus der Regelungstechnik, speziell aus dem Bereich der rückgekoppelten Systeme.

Einen dritten Zweig bilden imperative synchrone Sprachen, wie z.B. ESTEREL, Reactive-C (eine synchrone Erweiterung der Sprache C) und SML (synchronous modelling language). Bei imperativen synchronen Sprachen besteht eine Reaktion in der Ausführung eines Codesegments

zwischen zwei Kontrollpunkten, d.h. Stellen im Programmtext, bei denen eine Zeiteinheit verbraucht werden soll, bzw. Stellen, an denen sich verschiedene Kontrollflüsse synchronisieren. Jedem Prozess ist zu einem Zeitpunkt genau ein Kontrollpunkt zugeordnet. Speziell ESTEREL bietet für den Entwurf eingebetteter Systeme wichtige Konstrukte zur Unterbrechungsbehandlung. Diese Konstrukte gewährleisten, dass eine Unterbrechung sofort, d.h. noch während der momentanen Interaktion stattfindet. Dieses Verhalten kann ansonsten nur mit Hilfe von Echtzeitbetriebssystemen gewährleistet werden, was bei ESTEREL jedoch nicht notwendig ist.

Ferner wurden für ESTEREL spezielle Algorithmen zur Software- und Hardwaresynthese entwickelt. Bei der Softwaresynthese werden ESTEREL-Module, die aus sehr vielen Prozessen bestehen können, auf einen einzigen sequentiellen Prozess abgebildet. Der dabei entstehende Prozess kann dann als C-Programm ausgegeben werden und steht somit für nahezu alle Mikrocontroller zur Verfügung. Besonders vorteilhaft ist, dass dabei kein (Echtzeit-) Betriebssystemkern zur Prozessverwaltung mehr notwendig ist, der einen hohen Teil der für die Software verfügbaren Ressourcen beanspruchen würde. Daneben wurden auch Methoden zur direkten Hardwaresynthese aus ESTEREL-Programmen entwickelt, um die vorhandene Parallelität direkt nutzen zu können. Häufig sind die so entwickelten Schaltungen auch ohne weitere Optimierungen sehr effizient.

Die zur Synthese von ESTEREL-Programmen verwendeten Techniken sind als korrekt nachgewiesen worden, so dass sie formale Syntheseschritte darstellen, deren Resultate per Konstruktion zum gegebenen ESTEREL-Entwurf äquivalent sind. Somit kann sich die Verifikation vollständig auf ESTEREL-Entwürfe beschränken. Für die Verifikation bieten synchrone Sprachen entscheidende Vorteile: während für reale Zeitmodelle aufgrund von Unentscheidbarkeitsresultaten kaum Rechnerunterstützung zu erwarten ist, eignen sich Systembeschreibungen auf synchronen, logischen Zeitmodellen gut für eine automatische Verifikation. Logische Zeitmodelle stehen in engem Zusammenhang mit dem realen Zeitverbrauch, so dass Rückschlüsse auf das reale Echtzeitverhalten möglich sind. Bei der Hardwaresynthese können heute verfügbare Werkzeuge die minimalen Taktzeiten ermitteln und ggf. Optimierungen vornehmen. Nach der Softwaresynthese von ESTEREL kann die maximale Anzahl an Maschinenbefehlen bestimmt werden, die während eine Interaktion auszuführen ist, da innerhalb einer Interaktion keine datenabhängige Schleife vorkommen kann. Zusammen mit den Kenndaten des Mikroprozessors ergeben sich somit in einfacher Weise Abschätzungen für reale Zeitschranken.

2.2 ESTEREL

ESTEREL wurde am Centre de Mathématiques Appliquées, Ecole des Mines Paris entwickelt. Beim Bau eines selbstfahrenden Roboters stellten die Entwickler fest, dass sich die Struktur des reaktiven Systems des Roboters nur umständlich mit herkömmlichen imperativen Program-

miersprachen beschreiben lässt. Es wurde also eine Programmiersprache benötigt, mit der sich synchrone reaktive Systeme beschreiben lassen. G. Berry entwickelt daraufhin eine Semantik sowie einen Compiler für ESTEREL [1, 2]. In der Semantik ist festgelegt, dass ESTEREL-Programme ohne Zeitverzögerung auf Eingaben reagieren, wobei die Zeitpunkte zu denen die Eingaben erfolgen, nicht äquidistant sein müssen. Die Modellierung der Zeit für ESTEREL geschieht mit Hilfe der natürlichen Zahlen. Die Interaktion von Threads erfolgt über Broadcasting von Signalen, welche, neben einem Datum, zu jedem betrachteten Zeitpunkt einen der beiden Zustände *present* oder *absent* haben. ESTEREL definiert keine vollständige Sprache, da es nur sehr wenige Grunddatentypen gibt. Alle weiteren Datentypen werden von einer imperativen Programmiersprache wie z.B. C entlehnt und dann in ESTEREL importiert. Die ESTEREL-Programme werden dann in diese imperative Programmiersprache übersetzt und kümmern sich daher im Prinzip nur um die Kontrolle der Parallelität. Es ist nicht möglich mit Hilfe von ESTEREL nichtdeterministische Systeme zu beschreiben, sondern nur reaktive deterministische Systeme. Auf die Syntax von ESTEREL wird an dieser Stelle nicht eingegangen, da sie der von PURR, welche im nächsten Abschnitt beschrieben wird, sehr ähnlich ist.

2.3 PURR

Die Programmiersprache PURR wurde am Institut für Rechnerentwurf und Fehlertoleranz der Universität Karlsruhe entwickelt. Mit der Sprache PURR ist es möglich synchrone parallele Systeme zu beschreiben. Die Syntax ist so klein wie möglich, aber so groß wie nötig, um alle wichtigen Funktionen von synchronen parallelen Systemen zu beschreiben. Es ist möglich, eigene Datentypen sowie eigene Operatoren zu definieren, was aber hier nicht betrachtet werden soll. Hier werden nur einfache Datentypen wie boolesche Werte und endliche Felder von booleschen Werten betrachtet. Die Definition komplexerer Datentypen auf der Grundlage dieser Grunddatentypen ist jedoch möglich. Das Erzeugen von nichtdeterministischen Programmen ist mit PURR zwar grundsätzlich möglich, soll jedoch in dieser Arbeit nicht behandelt werden.

Im nächsten Abschnitt wird die Syntax von PURR beschrieben. Hierbei werden alle Aspekte, die mit dem Parser der Sprache zu tun haben, vernachlässigt. Diese Details werden erst in dem Kapitel, welches sich mit der Implementierung befasst behandelt. Die Semantik folgt dann in Kapitel 3.3.2

2.3.1 Datentypen von PURR

Bei der Definition der Syntax von PURR werden zuerst alle möglichen Ausdrücke, dann alle möglichen Anweisungen und am Schluss die dadurch erzeugbaren Module bzw. Programme definiert. Wie schon weiter oben erwähnt werden hier nur einfache Datentypen, wie boolesche Werte und endliche Felder von booleschen Werten betrachtet, mit denen es aber möglich wäre,

komplexere Datentypen zu definieren. Es wurden hier bereits einige Operatoren für arithmetische Ausdrücke berücksichtigt. Dabei werden die booleschen Felder als Zahldarstellung im Zweierkomplement interpretiert.

Definition 1 (Zweierkomplement) *Ein boolesches Feld der Länge n definiert eine vorzeichenbehaftete Zahl im Zweierkomplement wie folgt:* $\phi([b_n, \cdots, b_0]) := -b_n \cdot 2^n + \sum_{i=0}^{n-1} b_i \cdot 2^i$

In der Zweierkomplementdarstellung steht bei einer negativen Zahl an der Bitposition mit der höchsten Wertigkeit eine 1, während dort bei einer positiven Zahl eine 0 steht. Im Zweierkomplement gibt es nur eine Darstellung für die 0, es gibt jedoch auch eine negative Zahl, deren positives Gegenstück nicht darstellbar ist, z.B. $-2^n = \phi([1, 0, \cdots, 0])$.
Als Beispiel soll hier die Multiplikation für zwei vorzeichenbehaftete Zahlen vorgestellt werden. Das Produkt zweier vorzeichenbehafteter Zahlen kann berechnet werden, indem zuerst das Produkt der Absolutwerte der beiden Zahlen berechnet wird, um dann in einem zweiten Schritt das Vorzeichen des Produktes zu bestimmen. Die Multiplikation zweier Bitvektoren $[a_n, \cdots, a_0]$ und $[b_m, \cdots, b_0]$, die nicht vorzeichenbehaftet sind, wird durchgeführt indem jedes Bit des Bitvektors $[a_n, \cdots, a_0]$ mit jedem Bit des Bitvektors $[b_n, \cdots, b_0]$ konjunktiv verknüpft wird. Die sich hierbei ergebenden Bitvektoren werden dann unter Verwendung von Shiftleft-Operationen addiert:

	$a_n \wedge b_m$	$a_n \wedge b_{m-1}$ $a_{n-1} \wedge b_m$	$a_n \wedge b_1$ $a_{n-1} \wedge b_{m-1}$ $\cdots$	$a_n \wedge b_0$ $a_{n-1} \wedge b_1$ $\cdots$ $a_1 \wedge b_m$	$a_{n-1} \wedge b_0$ $\cdots$ $a_1 \wedge b_{m-1}$ $a_0 \wedge b_m$	$\cdots$ $a_1 \wedge b_1$ $a_0 \wedge b_{m-1}$	$a_1 \wedge b_0$ $a_0 \wedge b_1$	$a_0 \wedge b_0$
p_{n+m+1}	p_{n+m}	p_{n+m-1}	$\cdots$	p_{m+1}	p_m	p_{m-1}	$\cdots$	p_0

Die Bitbreite des Ergebnisvektors ist dabei durch $n + m + 2$ nach oben beschränkt, denn es gilt:

$$\phi([a_n, \cdots, a_0]) \cdot \phi([b_n, \cdots, b_0]) \leq (2^{n+1} - 1)(2^{m+1} - 1) < 2^{n+1}2^{m+1} = 2^{n+m+2}$$

Der Absolutwert einer Zahl im Zweierkomplement kann wie folgt berechnet werden:

$$|\phi([a_n, \cdots, a_0])| = \begin{cases} \phi([a_n, \cdots, a_0]) & : \text{ wenn } a_n = 0 \\ \phi(\text{2COMPL}([a_n, \cdots, a_0])) & : \text{ wenn } a_n = 1 \end{cases}$$

wobei 2COMPL$([a_n, \cdots, a_0])$:= INC $([\bar{a_n}, \cdots, \bar{a_0}])$

und INC $([a_n, \cdots, a_0]) := [b_n \oplus \bigwedge_{i=0}^{n-1} b_i, \cdots, b_j \oplus \bigwedge_{i=0}^{j-1} b_i, \cdots, b_1 \oplus b_0, b_0 \oplus 1]$.

Im Prinzip bleibt es jedoch bei booleschen Typen und Feldern.

2.3.2 Die Ausdrücke in PURR

In PURR besitzt jeder Ausdruck einen Typ. Da hier nur boolesche Werte und endliche Felder von booleschen Werten betrachtet werden, kann jeder dieser Datentypen durch eine natürliche Zahl repräsentiert werden. Der Datentyp 4 beschreibt z.B. ein Feld der Länge 4, besitzt also die Elemente $x[0]$, $x[1]$, $x[2]$ und $x[3]$. Hat x jedoch den Typ 0, dann bedeutet das, dass x ein einfacher boolescher Wert ist, der nicht indiziert werden kann. Es ist wichtig zu erwähnen, dass in PURR zwischen einfachen booleschen Werten mit dem Typ 0 und Feldern der Länge 1 unterschieden wird. Jeder Operator f, der auf booleschen Werten oder Feldern von booleschen Werten definiert ist, bekommt also k Argumente n_1 , $\cdots$, n_k und liefert einen Wert mit dem Typ $m \in \mathbb{N}$ zurück. Für einen Operator f gilt also $f : \mathbb{N}^k \to \mathbb{N}$. Wenn man nun $\mathbb{N}^* := \bigcup_{i=0}^{\infty} \mathbb{N}^i$ setzt, dann kann man jedem Typ eines Operators einen Wert aus $\mathbb{N}^*$ zuordnen. Als Beispiel soll hier nun die Multiplikation betrachtet werden. Die Multiplikation, angewandt auf zwei boolesche Felder der Länge $l \in \mathbb{N}$, welche als binäre Zahlen interpretiert werden, hat also den Typ $(l, l, 2l) \in \mathbb{N}^3 \subseteq \mathbb{N}$. Dies bedeutet, dass zwei Felder der Länge l auf ein Feld der Länge $2l$ abgebildet werden. Die formale Definition der möglichen Ausdrücke ist wie folgt:

Definition 2 *Eine PURR-Signatur $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ besteht aus einer Menge von Variablen V_Σ, einer Menge von Konstanten C_Σ sowie einer Typzuordnungsfunktion $s_\Sigma : V \cup C \to \mathbb{N}^*$ mit $s_\Sigma(x) \in \mathbb{N}^*$ für alle $x \in V_\Sigma$. Die Menge der Variablen $V_\Sigma = \Sigma_I \cup \Sigma_O \cup \Sigma_L \cup \Sigma_V$ besteht aus den disjunkten Mengen der Eingaben Σ_I, der Ausgaben Σ_O, der Programmstellen Σ_L und der lokalen Variablen Σ_V. Es gilt $s_\Sigma(x) > 0$ für $x \in \Sigma_V$. Zu einer gegebenen PURR-Signatur Σ definieren die folgenden Regeln die möglichen Ausdrücke und ihre Typen:*

- *$x \in PTerm_\Sigma$ mit dem Typ $typ_\Sigma(x) := s_\Sigma(x)$ für alle $x \in \Sigma_I \cup \Sigma_O \cup \Sigma_V$*
- *$\tau[i] \in PTerm_\Sigma$ mit dem Typ $typ_\Sigma(\tau[i]) := 0$, wenn $\tau \in PTerm_\Sigma$ mit dem Typ $typ_\Sigma(\tau[i]) := n > 0$*
- *$f(\tau_1, \cdots, \tau_n) \in PTerm_\Sigma$ mit dem Typ $typ_\Sigma(f(\tau_1, \cdots, \tau_n)) := m$, wenn $s_\Sigma(f) = (typ_\Sigma(\tau_1), \cdots, typ_\Sigma(\tau_n), m)$*
- ***not** τ_1 , (τ_1 **and** τ_2), (τ_1 **or** τ_2) $\in PTerm_\Sigma$ mit dem Typ $typ_\Sigma(\tau_1$ **and** $\tau_2) := typ_\Sigma(\tau_1$ **or** $\tau_2) := typ_\Sigma($**not** $\tau_1) := 0$, wenn $\tau_1, \tau_2 \in PTerm_\Sigma$ mit $typ_\Sigma(\tau_1) = typ_\Sigma(\tau_2) = 0$*

Es gilt weiterhin, dass für alle $\tau \in PTerm_\Sigma$, $typ_\Sigma(\tau)$ eine natürliche Zahl ist.

Die Σ_L sind Markierungen im Programm, welche zur Darstellung des Kontrollflusses dienen. Diese können daher nicht für Ausdrücke verwendet werden. Die Σ_I bzw. Σ_O sind die Bezeichner, die für die Ein- bzw. Ausgaben verwendet werden und die Σ_V sind die Bezeichner, die für lokale Variablen verwendet werden. $\Sigma_I, \Sigma_O, \Sigma_L$ und Σ_V sind disjunkte Mengen. Das in dieser Arbeit verwendete C_Σ ist $C_\Sigma = \{<, >, =, +, -, *, \mathit{Shiftleft}, \mathit{Konkatenation}\}$.

Ausdrücke können z.B. ***not*** $d[2]$ oder $a + b$ sein.

Alle Ausdrücke sind entweder boolesche Werte oder Felder von booleschen Werten. Die obige Definition beschreibt alle möglichen Ausdrücke in PURR. Hierzu gehören z.B. alle Operationen auf Datentypen oder Manipulationen von Datentypen in PURR. Jedes Signal hat zu einem Zeitpunkt nur einen einzigen Wert. Jedes Signal des Typs k kann also als eine Funktion des Typs $\mathbb{N} \rightarrow \mathbb{B}^k$ betrachtet werden, die dem Signal für jeden Zeitpunkt $t \in \mathbb{N}$ einen Wert aus $\mathbb{B}^k$ zuordnet, wobei k vom Typ des Signals abhängt. Auch der Verlauf von Signalwerten ist jedoch von dem Typ abhängig: Wenn ein Signal den Typ boolesches Feld hat (Typ > 0), dann ändert sich sein Wert nicht, wenn während des Verstreichens einer Zeiteinheit keine Operation auf diesem Signal ausgeführt wird. Dies ist die normale Vorgehensweise in imperativen Programmiersprachen. Diese Vorgehensweise ist anders als in ESTEREL. Hier wird der Status eines Signals zu jedem Zeitpunkt neu berechnet. Der Status des Signals kann zu diesem Zeitpunkt nur *present* werden, wenn es zu diesem Zeitpunkt ein zu diesem Signal gehörendes ***emit*** gibt. Allerdings ist es in ESTEREL möglich, dass Signale wertbehaftet sind. Solche wertbehafteten Signale tragen neben dem Signalstatus noch einen Wert, der nur durch eine Zuweisung geändert werden kann. Wenn man also in ESTEREL den Status von wertbehafteten Signalen weglässt, dann verhalten sie sich wie die booleschen Felder in PURR. Wenn andererseits der Wert des Signals nur boolesch ist, es sich jedoch nicht um ein Feld handelt (Typ = 0), dann ändert sich ihr Wert während des Verstreichens einer Zeiteinheit. In einem solchen Fall wird der Wert des Signals zuerst auf 0 gesetzt und wird erst dann 1, wenn eine Anweisung ihn auf 1 setzt. Boolesche Felder der Länge 1 verhalten sich also anders als boolesche Signale. Dies zeigt sich auch in den Anweisungen: Zur Änderung von Signalen des Typs 0 wird ***emit***, sonst ***:=*** verwendet. Dies ist kein Unterschied zu ESTEREL, da der Signalstatus in ESTEREL wie Signale des Typs 0 behandelt wird.

2.3.3 Die Anweisungen in PURR

Nun werden die grundlegenden Anweisungen von PURR definiert, aus denen sich die PURR-Programme zusammensetzen:

Definition 3 (grundlegende Anweisungen von PURR) *Bei einer gegebenenen PURR-Signatur Σ definieren die folgenden Regeln die grundlegenden Anweisungen $PStm_\Sigma$ von PURR. Es gilt im Folgenden stets $typ_\Sigma(\sigma) = 0$, d.h. σ ist ein boolescher Ausdruck ($\sigma \in PTerm_\Sigma$ mit $typ_\Sigma(\sigma) = 0$).*

- ***nothing*** $\in PStm_\Sigma$
- ***pause*** $\in PStm_\Sigma$
- ***emit*** $x \in PStm_\Sigma$ *und* ***emit*** x ***after 1*** $\in PStm_\Sigma$, *wenn* $x \in \Sigma_O \cup \Sigma_L$ *mit* $typ_\Sigma(x) = 0$

- *x* **:=** $\tau \in$ *PStm*$_\Sigma$ *und x* **:=** τ ***after 1*** $\in$ *PStm*$_\Sigma$*, wenn* $x \in \Sigma_V$ *und* $\tau \in$ *PTerm*$_\Sigma$ *mit typ*$_\Sigma(\tau)$ $= s_\Sigma(x) > 0$

- ***present*** σ ***then*** S_1 ***else*** S_2 ***end*** $\in$ *PStm*$_\Sigma$*,* $S_1, S_2 \in$ *PStm*$_\Sigma$

- S_1 ***;*** $S_2 \in$ *PStm*$_\Sigma$*, wenn* $S_1, S_2 \in$ *PStm*$_\Sigma$

- $S_1 \parallel S_2 \in$ *PStm*$_\Sigma$*, wenn* $S_1, S_2 \in$ *PStm*$_\Sigma$

- ***while*** σ ***do*** S ***end*** $\in$ *PStm*$_\Sigma$*, wenn* $S \in$ *PStm*$_\Sigma$

- ***suspend*** S ***when*** $\sigma \in$ *PStm*$_\Sigma$*, wenn* $S \in$ *PStm*$_\Sigma$

- ***weak suspend*** S ***when*** $\sigma \in$ *PStm*$_\Sigma$*, wenn* $S \in$ *PStm*$_\Sigma$

- ***abort*** S ***when*** $\sigma \in$ *PStm*$_\Sigma$*, wenn* $S \in$ *PStm*$_\Sigma$

- ***weak abort*** S ***when*** $\sigma \in$ *PStm*$_\Sigma$*, wenn* $S \in$ *PStm*$_\Sigma$

- ***signal*** $x{:}\alpha$ ***in*** S ***end*** $\in$ *PStm*$_\Sigma$*, wenn* $x \in \Sigma_L$ *mit* $s_\Sigma(x) = \alpha$ *und* $S \in$ *PStm*$_\Sigma$

- ***run*** *m* $[y_1/x_1, \cdots, y_n/x_n] \in$ *PStm*$_\Sigma$*, wenn* $x_i \in$ *PTerm*$_\Sigma$ *und* $y_i \in$ *PTerm*$_\Sigma$ *mit* $s_\Sigma(x_i) = s_\Sigma(y_i)$

Da PURR eine synchrone Sprache ist, hängen die Werte der Variablen und damit die Werte der Ausdrücke von der Zeit ab. Die Zeit wird durch die natürlichen Zahlen repräsentiert. Die Abarbeitung einer Anweisung S wird zu einem bestimmten Zeitpunkt t_1 begonnen. Die Abarbeitung von S kann zum selben Zeitpunkt, zu einem späteren Zeitpunkt t_2 oder niemals beendet werden. Wenn S nicht zum Zeitpunkt t_1 terminiert, dann verbraucht S Zeit.
Bevor die Semantik der oben definierten Anweisungen formal eingeführt wird, folgt nun eine kurze Beschreibung der Anweisungen.

- ***pause***: Dies ist die einzige Anweisung, die Zeit verbraucht. Die Abarbeitung dieser Anweisung ändert keine Werte von Signalen und hat keine Seiteneffekte. Diese Anweisung verbraucht eine Zeiteinheit.

- ***nothing***: Diese Anweisung tut nichts. Sie ändert keine Werte von Signalen und hat keine Seiteneffekte. ***nothing*** verbraucht auch keine Zeiteinheit. Diese Anweisung wird als Platzhalter, d.h. als leere Anweisung verwendet.

- ***emit*** x: Diese Anweisung emittiert das Signal x. Der Typ des Signals x ist also boolesch und in dem Zeitschritt, in dem die Anweisung abgearbeitet wird, hat x den Wert 1. Man beachte, dass x den Wert 0 hat (unabhängig vom vorigen Wert), falls zu diesem Zeitpunkt keine Emission von x stattfand. ***emit*** x verbraucht keine Zeiteinheit.

- ***emit*** x ***after 1***: Diese Anweisung hat genau die gleiche Wirkung wie die Anweisung ***emit*** x, mit dem einzigen Unterschied, dass die Emission des Signals x erst eine Zeiteinheit nach der Abarbeitung der Anweisung auftritt. Diese Anweisung verbraucht ebenfalls keine Zeiteinheit.

- x ***:=*** τ: Für diese Anweisung gilt die Einschränkung, dass x ein boolesches Feld der Länge l sein muss. Die Abarbeitung der Anweisung ändert den Wert von x auf den momentanen Wert des Ausdrucks τ. Man beachte, dass x seinen Wert beibehält, falls zu diesem Zeitpunkt keine Zuweisung stattfindet. Diese Anweisung verbraucht keine Zeiteinheit.

- x ***:=*** τ ***after 1***: Diese Anweisung hat die gleiche Wirkung wie die Anweisung x ***:=*** τ mit dem einzigen Unterschied, dass die Änderung von x erst eine Zeiteinheit nach der Abarbeitung der Anweisung durchgeführt wird. Diese Anweisung verbraucht ebenfalls keine Zeiteinheit.

- ***present*** σ ***then*** S_1 ***else*** S_2 ***end***: Diese Anweisung verhält sich wie S_1, wenn sich σ in diesem Zeitschritt zu 1 auswerten lässt, andernfalls wie S_2. Hierzu muss σ ein Ausdruck mit $\text{typ}_\Sigma(\sigma) = 0$, also z.B. ein boolescher Wert sein.

- S_1 ***;*** S_2: Diese Anweisung beschreibt die Hintereinanderausführung von Anweisungen. Bei der Abarbeitung dieser Anweisung wird zuerst die Anweisung S_1 und danach die Anweisung S_2 abgearbeitet. Hierbei ist es möglich, dass S_1 niemals terminiert und daher die Abarbeitung von S_2 niemals begonnen wird.

- $S_1 \parallel S_2$: Diese Anweisung beschreibt die synchrone parallele Ausführung von Anweisungen. Bei der Abarbeitung von $S_1 \parallel S_2$ wird gleichzeitig mit der Abarbeitung von S_1 und S_2 begonnen. Die Abarbeitung dieser Anweisungen wird erst beendet, wenn die Abarbeitung von S_1 und S_2 beendet wird. Wenn z.B. die Abarbeitung von S_1 beendet wird, dann verhält sich $S_1 \parallel S_2$ von da an nur noch wie S_2, bis auch die Abarbeitung von S_2 beendet wird. Die Ausführungen von S_1 und S_2 sind synchron, da an jeder ***pause***-Anweisung eine Synchronisierung stattfindet.

- ***while*** σ ***do*** S ***end***: Diese Anweisung unterscheidet zwei Möglichkeiten. Wenn σ zum aktuellen Zeitpunkt zu 1 ausgewertet werden kann, dann wird mit der Abarbeitung von S begonnen, ansonsten wird S nicht ausgeführt. Es ist möglich, dass S niemals terminiert. Ansonsten verhält sich die Anweisung wie ***present*** σ ***then*** S.

- ***suspend*** S ***when*** σ: Diese Anweisung implementiert das zeitweilige Aussetzen von Prozessen. Wenn mit der Abarbeitung dieser Anweisung begonnen wird, dann werden zunächst alle Anweisungen von S, die zum momentanen Zeitpunkt ausgeführt werden können, ausgeführt, unabhängig vom momentanen Wert von σ. Danach wird nur dann

mit der Ausführung von S fortgefahren, wenn sich σ zu 0 auswerten lässt. Ansonsten wird die Ausführung von S solange ausgesetzt, bis sich σ wieder zu 0 auswerten lässt.

- ***weak suspend*** S ***when*** σ: Diese Anweisung verhält sich fast genau so wie ***suspend*** S ***when*** σ. Der einzige Unterschied besteht im Datenfluss : Falls σ auftritt, werden die Datenmanipulationen, die von ***emit*** und ***:=*** durchgeführt werden, im Gegensatz zu ***suspend***, übernommen.

- ***abort*** S ***when*** σ: Diese Anweisung wird zum Abbrechen von Prozessen benutzt. Wenn mit der Abarbeitung dieser Anweisung begonnen wird, dann werden alle Anweisungen von S, die zum momentanen Zeitpunkt ausgeführt werden können, ausgeführt, unabhängig vom momentanen Wert von σ. Danach wird die Ausführung von S sofort beendet, wenn sich σ zu 1 auswerten lässt. Ansonsten wird S normal abgearbeitet solange sich σ zu 0 auswerten lässt. Die Abarbeitung von ***abort*** S ***when*** σ kann also auf zwei Arten beendet werden: einmal dadurch, dass die Abarbeitung von S ganz normal beendet wird oder dadurch, dass σ zu 1 wird, was die Abarbeitung sofort stoppt.

- ***weak abort*** S ***when*** σ: Diese Anweisung verhält sich fast genau so wie ***abort*** S ***when*** σ. Der einzige Unterschied besteht wieder im Datenfluss: Falls σ auftritt, werden die Datenmanipulationen, die von ***emit*** und ***:=*** durchgeführt werden im Gegensatz zu ***abort***, übernommen.

- ***signal*** x:α ***in*** S ***end*** : Diese Anweisung deklariert ein lokales Signal x vom Typ α mit dem Gültigkeitsbereich S.

- ***run*** m $[y_1/x_1, \cdots, y_n/x_n]$: Diese Anweisung ruft das Modul m auf. Das Hauptmodul ist jedoch das einzige Modul, das andere Module aufrufen kann. Die x_i sind die Ein- und Ausgaben welche bei der Implementierung von m verwendet werden. Diese werden beim Aufruf von m durch aktuelle Parameter y_i ersetzt. Der Aufruf von ***run*** m $[y_1/x_1, \cdots, y_n/x_n]$ verbraucht keine Zeiteinheit, und stellt nur eine textuelle Ersetzung dar.

Durch die Möglichkeit, nebenläufige Prozesse zu beschreiben, gibt es auch die Möglichkeit, dass es zu Schreibkonflikten kommen kann. Es muss also definiert sein, was passiert, wenn es zu mehr als einem Schreibzugriff auf eine Variable zu einem Zeitpunkt kommt. Dies geschieht mit Hilfe einer Resolutionsfunktion f_α, welche bei der Deklaration angegeben werden muss, wie folgt: Alle Zuweisungen $x := \tau_1, \cdots, x := \tau_n$ werden aufgesammelt, danach wird mit Hilfe der Funktion f_α der Wert von x bestimmt durch $x = f_\alpha(\tau_1, f_\alpha(\tau_2, \cdots, f_\alpha(\tau_n - 1, \tau_n) \cdots))$. Die Funktion f_α muss hierzu assoziativ und kommutativ sein, damit die Reihenfolge der Zuweisungen keine Rolle spielt. Für boolesche Signale ist diese Funktion einfach die Disjunktion über alle Signalwerte.

Die obige Definition beschreibt lediglich die grundlegenden Anweisungen in PURR. Da das Programmieren mit diesen Anweisungen sehr umständlich werden kann, gibt es noch einige zusammengesetzte Anweisungen, die das Programmieren einfacher gestalten. Diese Anweisungen werden nun definiert.

Definition 4 (zusammengesetzte Anweisungen in PURR) *Bei einer gegebenenen PURR-Signatur* Σ *definieren die folgenden Regeln die zusammengesetzten Anweisungen* $PStm_\Sigma$ *von PURR.*

- ***halt*** := ***while 1 do pause end***
- ***await*** σ := ***abort halt when*** σ
- ***sustain*** x := ***while 1 do emit*** x ***; pause end***
- ***sustain*** $x(\tau)$:= ***while 1 do*** x ***:=*** τ ***; pause end***
- ***loop*** S ***end*** := ***while 1 do*** S ***end***
- ***repeat*** S ***until*** σ := S ***; while*** $\neg\sigma$ ***do*** S ***end***
- ***loop*** S ***each*** σ := ***loop abort*** S ***; halt when*** σ ***end***
- ***every*** σ ***do*** S ***end*** := ***await*** σ ***; loop*** S ***each*** σ
- ***repeat*** n ***times*** S ***end*** := $\underbrace{S;\cdots;S}_{n-mal}$
- ***(weak) abort*** S_1 ***when*** σ ***do*** S_2 ***end*** := $\left[\begin{array}{l}\textit{\textbf{(weak) abort }} S_1 \textit{\textbf{ when }} \sigma \textit{\textbf{ ;}}\\ \textit{\textbf{present }} \sigma \textit{\textbf{ then }} S_2 \textit{\textbf{ end}}\end{array}\right]$
- ***(weak) abort*** S ***when immediate*** σ := $\left[\begin{array}{l}\textit{\textbf{(weak) abort}}\\ \quad \textit{\textbf{present }} \sigma \textit{\textbf{ else }} S \textit{\textbf{ end}}\\ \textit{\textbf{when }} \sigma \textit{\textbf{ end}}\end{array}\right]$
- ***(weak) suspend*** S ***when immediate*** σ := $\left[\begin{array}{l}\textit{\textbf{(weak) suspend}}\\ \quad \textit{\textbf{present }} \sigma \textit{\textbf{ then pause end}};\\ \quad S\\ \textit{\textbf{when }} \sigma \textit{\textbf{ end}}\end{array}\right]$
- $\left[\begin{array}{l}\textit{\textbf{present}}\\ \quad \textit{\textbf{case }} \sigma_1 \textit{\textbf{ do }} S_1\\ \qquad \vdots\\ \quad \textit{\textbf{case }} \sigma_n \textit{\textbf{ do }} S_n\\ \quad \textit{\textbf{else }} S_{n+1}\\ \textit{\textbf{end}}\end{array}\right] := \left[\begin{array}{l}\textit{\textbf{present }} \sigma_1 \textit{\textbf{ then }} S_1\\ \textit{\textbf{else present }} \sigma_2 \textit{\textbf{ then }} S_2\\ \qquad \vdots\\ \textit{\textbf{else present }} \sigma_n \textit{\textbf{ then }} S_n\\ \textit{\textbf{else }} S_{n+1}\\ \textit{\textbf{end}} \cdots \textit{\textbf{end}}\end{array}\right]$

- $$\left[\begin{array}{l}\textbf{\textit{await}}\\ \quad \textbf{\textit{case}}\ \sigma_1\ \textbf{\textit{do}}\ S_1\\ \quad\quad \vdots\\ \quad \textbf{\textit{case}}\ \sigma_n\ \textbf{\textit{do}}\ S_n\\ \textbf{\textit{end}}\end{array}\right] := \left[\begin{array}{l}\textbf{\textit{await}}\ \sigma_1\ \textbf{\textit{or}}\ \cdots\ \textbf{\textit{or}}\ \sigma_n\ \textbf{\textit{;}}\\ \textbf{\textit{present}}\\ \quad \textbf{\textit{case}}\ \sigma_1\ \textbf{\textit{do}}\ S_1\\ \quad\quad \vdots\\ \quad \textbf{\textit{case}}\ \sigma_n\ \textbf{\textit{do}}\ S_n\\ \quad \textbf{\textit{else}}\ S_{n+1}\\ \textbf{\textit{end}}\end{array}\right]$$

- $$\left[\begin{array}{l}\textbf{\textit{abort}}\ S\ \textbf{\textit{when}}\\ \quad \textbf{\textit{case}}\ \sigma_1\ \textbf{\textit{do}}\ S_1\\ \quad\quad \vdots\\ \quad \textbf{\textit{case}}\ \sigma_n\ \textbf{\textit{do}}\ S_n\\ \textbf{\textit{end abort}}\end{array}\right] := \left[\begin{array}{l}\textbf{\textit{abort}}\ S\ \textbf{\textit{when}}\ \sigma_1\ \textbf{\textit{or}}\ \cdots\ \textbf{\textit{or}}\ \sigma_n\ \textbf{\textit{;}}\\ \textbf{\textit{present}}\\ \quad \textbf{\textit{case}}\ \sigma_1\ \textbf{\textit{do}}\ S_1\\ \quad\quad \vdots\\ \quad \textbf{\textit{case}}\ \sigma_n\ \textbf{\textit{do}}\ S_n\\ \quad \textbf{\textit{else}}\ S_{n+1}\\ \textbf{\textit{end}}\end{array}\right]$$

Auch für diese zusammengesetzten Anweisungen erfolgt nun eine kurze informelle Beschreibung der Semantik:

- ***halt***: Diese Anweisung terminiert nie und hat keinerlei Seiteneffekte. Sie verbraucht die gesamte restliche Zeit, kann jedoch durch eine sie umgebende ***abort***-Anweisung beendet werden.

- ***await*** σ: Diese Anweisung wartet auf σ und verbraucht bis zum ersten Eintreten von σ Zeit. ***await*** σ terminiert mit dem Eintreten von σ.

- ***sustain*** x: Diese Anweisung hält das Signal x für alle Folgezeitpunkte auf dem Wert 1. ***sustain*** x terminiert nicht, kann aber durch eine umgebende ***abort***-Anweisung beendet werden.

- ***sustain*** $x(\tau)$: Diese Anweisung hält das Signal x für alle Folgezeitpunkte auf dem Wert τ. ***sustain*** $x(\tau)$ terminiert nicht, kann aber durch eine umgebende ***abort***-Anweisung beendet werden.

- ***loop*** S ***end***: Diese Anweisung startet die Abarbeitung von S nach jeder Terminierung von S neu. ***loop*** S ***end*** terminiert nicht, kann aber durch eine umgebende ***abort***-Anweisung beendet werden.

- ***repeat*** S ***until*** σ: Diese Anweisung führt zunächst S aus, und wiederholt S, falls nach Terminierung von S σ nicht gilt. S wird daher mindestens ein Mal ausgeführt.

- ***loop*** S ***each*** σ: Diese Anweisung beginnt sofort mit der Abarbeitung von S. Wenn σ währenddessen zu 1 ausgewertet werden kann, dann wird die Abarbeitung von S abgebrochen und sofort mit einer erneuten Abarbeitung von S begonnen. Wurde die Abarbeitung von S beendet, bevor σ wieder zu 1 ausgewertet werden konnte, dann wird darauf gewartet, dass σ wieder zu 1 ausgewertet werden kann. Dann wird wieder mit der Abarbeitung von S begonnen.

- ***every*** σ ***do*** S ***end***: Diese Anweisung wartet darauf, dass σ zu 1 ausgewertet werden kann und startet dann die Abarbeitung von S. Die Abarbeitung von S wird sofort wieder begonnen, wenn σ wieder zu 1 ausgewertet werden kann, auch dann, wenn die Abarbeitung von S vorher nicht beendet wurde. Wurde die Abarbeitung von S beendet, bevor σ wieder zu 1 ausgewertet werden konnte, dann wird darauf gewartet, dass σ wieder zu 1 ausgewertet werden kann und es wird dann wieder mit der Abarbeitung von S begonnen.

- ***repeat*** n ***times*** S ***end***: Diese Anweisung führt S n-Mal hintereinander aus.

- ***(weak) abort*** S_1 ***when*** σ ***do*** S_2 ***end***: Diese Anweisung startet zunächst S_1. Die Abarbeitung von S_1 wird beendet, wenn σ zu 1 ausgewertet werden kann. In diesem Fall wird dann sofort mit der Abarbeitung von S_2 begonnen.

- ***(weak) abort*** S ***when immediate*** σ: Diese Anweisung beendet die Abarbeitung von S in dem Moment, wenn σ zu 1 ausgewertet werden kann. Im Gegensatz zu ***(weak) abort*** S ***when*** σ wird jedoch der Wert von σ zum Startzeitpunkt mit berücksichtigt.

- ***(weak) suspend*** S ***when immediate*** σ: Diese Anweisung setzt die Abarbeitung von S in jedem Moment aus, zu dem σ zu 1 ausgewertet werden kann. Im Gegensatz zu ***(weak) suspend*** S ***when*** σ wird jedoch auch der Wert von σ zum Startzeitpunkt mit berücksichtigt.

- $\left[\begin{array}{l} \textbf{\textit{present}} \\ \quad \textbf{\textit{case}}\ \sigma_1\ \textbf{\textit{do}}\ S_1 \\ \quad\quad \vdots \\ \quad \textbf{\textit{case}}\ \sigma_n\ \textbf{\textit{do}}\ S_n \\ \quad \textbf{\textit{else}}\ S_{n+1} \\ \textbf{\textit{end}} \end{array}\right]$: Diese Anweisung verhält sich wie S_i, wenn σ_i zum Startzeitpunkt zu 1 ausgewertet werden kann.

- $\left[\begin{array}{l} \textbf{\textit{await}} \\ \quad \textbf{\textit{case}}\ \sigma_1\ \textbf{\textit{do}}\ S_1 \\ \quad\quad \vdots \\ \quad \textbf{\textit{case}}\ \sigma_n\ \textbf{\textit{do}}\ S_n \\ \textbf{\textit{end}} \end{array}\right]$: Diese Anweisung wartet solange, bis eines der Ereignisse σ_i zu 1 ausgewertet werden kann und beginnt dann mit der Abarbeitung der S_i, die zu den eingetretenen Ereignissen gehören.

- $\left[\begin{array}{l} \textbf{\textit{abort}}\ S\ \textbf{\textit{when}} \\ \quad \textbf{\textit{case}}\ \sigma_1\ \textbf{\textit{do}}\ S_1 \\ \quad\quad \vdots \\ \quad \textbf{\textit{case}}\ \sigma_n\ \textbf{\textit{do}}\ S_n \\ \textbf{\textit{end abort}} \end{array}\right]$: Es wird sofort mit der Abarbeitung von S begonnen. Kann eines der σ_i während der Abarbeitung zu 1 ausgewertet werden, dann wird die Abarbeitung von S beendet und es wird mit der Abarbeitung der S_i begonnen, die zu den eingetretenen Ereignissen gehören.

Nachdem nun alle wichtigen Anweisungen von PURR vorgestellt wurden, muss nur noch definiert werden, wie ein gültiges PURR-Programm aussieht. Ein PURR-Programm besteht aus einer Liste von Modulen, die die folgende Form haben:

module m
input $x_1 : l_1, \cdots, x_m : l_m$;
output $y_1 : k_1, \cdots, y_n : k_n$;
  S_m
end module

m ist der Modulname, l_i und k_i sind natürliche Zahlen, die den Typ der Ein- und Ausgaben angeben und die Anweisung $S_m \in \text{PStm}_\Sigma$ ist der Modulrumpf. Die Module können benutzt werden, um ein System in eine hierarchische Struktur zu gliedern oder, um Module mehrfach nutzen zu können. Im Folgenden sollen nun beispielhaft die beiden PURR-Programme $SPEED$ und $CONTROL$ vorgestellt und ihre Funktion erklärt werden.

```
module SPEED
input cm, Second
output Speed : 8
loop
  signal distance : 8 in
    weak abort
      every cm do
        distance := distance + 1
      end every
    when Second do
      Speed := distance
    end abort
  end signal
end loop
end module
```

Dieses Programm zählt die Zahl der pro Sekunde zurückgelegten cm. Zu jeder vollen Sekunde wird die Zahl als Maß für die Geschwindigkeit der Ausgabevariablen $Speed$ zugewiesen. Falls die Signale $Second$ und cm gleichzeitig auftreten sollten, dann wird cm dem vorigen Intervall zugeordnet.

module $CONTROL$
input req, rdy_1, rdy_2
output req_1, req_2, rdy
loop
  abort
    l_1 : ***sustain*** rdy
  when immediate req;
  emit req_1;
  l_2 : ***await*** rdy_1;
  emit req_2;
  l_3 : ***await*** rdy_2;
end loop
end module

Dieses Programm realisiert einen Controler für ein Handshake-Protokoll. Das Programm emittiert an der Programmstelle l_1 solange das Signal rdy, bis das Signal req den Wert 1 annimmt. Danach wird vom Programm sofort das Signal req_1 emittiert und an Programmstelle l_2 gewartet, bis das Signal rdy_1 den Wert 1 annimmt. Hiernach wird dann das Signal req_2 emittiert und es wird an Programmstelle l_3 gewartet, bis das Signal rdy_2 den Wert 1 annimmt. Nachdem die Schleife einmal abgearbeitet wurde, wird wieder mit der Abarbeitung der Schleife an Programmstelle l_1 begonnen.

2.4 Verifikation reaktiver Systeme

In unserer Gesellschaft kommen mittlerweile fast überall technische Systeme zum Einsatz. Diese technischen Systeme beeinflussen in starkem Maße direkt entscheidende Lebensbereiche. So sind z.B. in der Medizin viele Menschenleben vom einwandfreien Funktionieren der dort eingesetzten Geräte abhängig ebenso wie in der Verkehrsleittechnik, der Luftraumüberwachung oder beim computerunterstützten Betrieb von Stellwerken im Bahnverkehr. Devisen- oder Wertpapiergeschäfte mit einem Volumen von mehreren Milliarden Mark werden von den Banken fast ausschließlich mit Hilfe von Computern abgewickelt und auch die Kontoführung wäre ohne Computer heute in fast allen Ländern undenkbar. Diese starke Verbreitung technischer Systeme führt zu der Frage, ob man diesen Systemen so sehr vertrauen kann, dass man ihnen sein Leben und sein Geld anvertrauen kann. Laprie [12] führte in diesem Zusammenhang den Be-

griff der Vertrauenswürdigkeit ein, der die Aspekte eines Rechnersystems umfasst, die für den einwandfreien Betrieb notwendig sind:

- *Zuverlässigkeit:* Das System hört nicht auf zu funktionieren.
- *Verfügbarkeit:* Das System ist immer einsatzbereit.
- *Schutz:* Das System verhindert nicht autorisierten Zugriff auf Informationen.
- *Sicherheit:* Durch den Betrieb des Systems können keine Personen- oder Sachschäden enstehen.

Die Aspekte der Zuverlässigkeit und der Verfügbarkeit werden durch die Verwendung von fehlertoleranten Rechnersystemen abgedeckt. Der Schutz von Daten und die Verhinderung von nicht autorisiertem Zugriff auf ein Rechnersystem kann durch die Verwendung von kryptographischen Verfahren sichergestellt werden.

Daher wird von jetzt an nur noch der Sicherheitsaspekt eines Systems betrachtet.

Unter dem Begriff der Sicherheit versteht man im Allgemeinen, dass sich keine Gefahren für Personen oder Sachen ergeben können. Besonders wichtig ist dies bei Systemen, bei denen das Ausmaß eines möglichen Schadens groß werden kann, also bei Systemen, bei denen Menschenleben direkt vom Funktionieren der Systeme abhängen. Hierunter fallen z.B.:

- Rechnersysteme an Bord von Luft- und Raumfahrzeugen
- Sicherungseinrichtungen in Kraftwerken
- Geräte in der Intensivmedizin
- Rechnergestützte Komponenten in Fahrzeugen (ABS, Airbag, ...)
- Kommunikation- und Navigationsselektronik an Bord von Schiffen

Der Hauptgrund dafür, dass Systeme nicht wie gewünscht funktionieren, liegt im Vorhandensein von Fehlern. Um zu zeigen, dass ein System sicher ist, muss nun gezeigt werden, dass das System fehlerfrei ist, oder dass das Vorhandensein von Fehlern die Funktion des Systems nicht beeinträchtigen kann. Kann man dies beides nicht zeigen, dann muss zumindest gelten, dass eine Funktionsbeeinträchtigung nicht zu katastrophalen Folgen führen kann. Neben den Gefahren für Personen und Sachen können Fehler in Systemen auch sehr teuer werden, im Besonderen, wenn es sich bei dem System um eine integrierte Schaltung handelt. Während die Beseitigung eines Fehlers und die Auslieferung der neuen Version bei Softwareprodukten noch relativ einfach und billig zu bewerkstelligen ist, ist dies bei Hardwareprodukten weitaus kostspieliger. Die

Änderung eines Entwurfs auf einer hohen Abstraktionsebene ist hierbei nicht so aufwendig wie die Änderung auf einer niedrigen Abstraktionsebene, da in einem solchen Fall viel manuelle Entwurfsarbeit noch einmal wiederholt werden müsste. Das Ersetzen einer bereits gefertigten und ausgelieferten Komponente ist bei Hardwareprodukten eine sehr teure Angelegenheit. Ein gutes Beispiel hierfür ist der Fehler im Divisionswerk des Pentium-Prozessors von Intel. Die Beseitigung dieses Fehlers hat Intel ca. 100 Millionen Dollar gekostet. Hierbei ist der Imageverlust in der Öffentlichkeit noch nicht einmal mit einbezogen.

Ein weiterer Grund für den Wunsch nach fehlerfreien Systementwürfen ist der Zeitpunkt der Markteinführung. Da der zu erwartende Gewinn mit einer verspäteten Markteinführung immer mehr sinkt, ist man daran interessiert, keine Verzögerungen durch das Vorhandensein von Entwurfsfehlern bei der Markteinführung hinnehmen zu müssen.

Gründe für das Vorhandensein von Entwurfsfehlern sind die große Komplexität heutiger Systeme sowie fehlerhafte, unvollständige oder nicht eindeutige Spezifikationen. Man unterscheidet unterschiedliche Arten von Entwurfsfehlern:

- *Fehler bei Entwurfsschritten:* Durch die fehlerhafte Umsetzung einer Spezifikation auf einer Abstraktionsebene in eine Implementierung auf einer niedrigeren Abstraktionsebene wird nicht die gewünschte Funktion realisiert. Gründe hierfür können z.B. falsches Verständnis der Spezifikation oder fehlerhafte Algorithmen zur Realisierung von in der Spezifikation festgelegten Funktionen sein. Um solche Fehler zu entdecken, muss die Spezifikation mit der Implementierung verglichen werden.

- *Fehler bei lokalen Optimierungen:* Lokale Optimierungen werden verwendet, um Implementierungen auf einer bestimmten Abstraktionsebene hinsichtlich bestimmter Anforderungen zu verbessern, bevor sie für weitere Entwurfsschritte verwendet werden. Auch bei solchen Optimierungen können Fehler auftreten, vor allem, wenn sie manuell durchgeführt werden. Optimierungsfehler können durch den Vergleich zweier Implementierungen auf gleichem Abstraktionsniveau entdeckt werden.

- *Implementierungsinhärente Fehler:* Eine Implementierung erfüllt auch dann nicht die gewünschte Funktion, wenn bestimmte Bedingungen der einzelnen Entwurfsebenen nicht eingehalten werden. Eine solche Bedingung kann z.B. ein Mindestabstand zwischen zwei Leiterbahnen sein. Die Einhaltung solcher Bedingungen ist vor allem auf den niedrigen Abstraktionsebenen wichtig. Zur Erkennung solcher Fehler muss lediglich die Implementierung auf die Einhaltung der Bedingungen hin geprüft werden.

Zur Entdeckung von Entwurfsfehlern müssen entweder zwei Implementierungen oder eine Implementierung und die dazugehörige Spezifikation miteinander verglichen werden. Werden

zwei Implementierungen miteinander verglichen, so ist oft die Verhaltensgleichheit das Beweisziel, da sich die optimierte Schaltung genau so verhalten soll, wie die ursprüngliche Schaltung. Es müssen also für alle möglichen Eingaben und alle Zeiten beide Implementierungen die gleichen Ausgaben liefern. Wenn eine Implementierung und eine Spezifikation miteinander verglichen werden, dann ist zu zeigen, dass das Verhalten der Spezifikation erbracht wird.

Beim Nachweis von Entwurfsfehlern unterscheidet man zwischen Validierung und Verifikation. Unter Validierung versteht man die Prüfung eines Systems auf Eignung für den Verwendungszweck in der Einsatzumgebung. Es ist jedoch schwierig, vor dem Einsatz eines Systems dessen korrektes Funktionieren nachzuweisen. Bei der Verifikation jedoch prüft man ein System auf die Erfüllung vorgegebener Anforderungen. Diese Überprüfung ist klarer definiert, wenn man von formalen Anforderungen ausgeht, die das System erfüllen muss. Bei der Verifikation wird sichergestellt, dass die Anforderungen für alle Zeiten und Eingaben eingehalten werden. Die Simulation ist ein heute gängiges Mittel zur Validierung. Durch die Simulation lässt sich feststellen, ob z.B. eine Implementierung und eine Spezifikation unterschiedliche Ergebnisse liefern. Da die Simulationsstimuli jedoch nur eine Teilmenge aller möglichen Eingangsbelegungen und internen Zustände abdeckt, können Fehler unentdeckt bleiben. Eine Simulation ist allerdings nur dann eine Verifikation, wenn sie jede mögliche Verhaltensänderung der Implementierung gegenüber der Spezifikation aufdecken würde. Da alle Systeme immer nur eine endliche Zahl an Zuständen annehmen können, ist eine vollständige Simulation prinzipiell möglich. Die vollständige Simulation, bei der alle internen Zustände erreicht werden, ist im Prinzip eine Möglichkeit der Verifikation der Schaltung, ist jedoch bei großen Systemen aus Komplexitätsgründen sehr schnell nicht mehr durchführbar.

Eine mögliche Methode, alle Anforderungen an eine Implementierung vollständig zu überprüfen, ist ein mathematischer Beweis. Hat man bewiesen, dass sich eine Implementierung für alle Eingaben und alle Zeiten gemäss der Spezifikation verhält, kann man auf eine Simulation verzichten. Um solche Beweise mit Hilfe von Rechnern durchführen zu können, müssen die Spezifikation, die Implementierung und die Korrekheitsrelation zwischen der Spezifikation und der Implementierung in einem Formalismus vorliegen, der Beweise erlaubt. Sowohl die Spezifikation als auch die Beziehungen zwischen der Spezifikation und der Implementierung sind vom Menschen vorgegebene Zielvorgaben. Das gewünschte Verhalten muss also möglichst vollständig und fehlerfrei in einem Formalismus erfasst werden. Hierbei gibt es mehrere Eigenschaften, die sich formalisieren lassen. Hierzu gehören:

- *Funktionale Korrektheit:* Beispiel: Der Addierer addiert.
- *Zeitverhalten:* Beispiel: Das Ergebnis ist nach einer bestimmten, festen Zeit berechnet.

- *Sicherheitseigenschaften:* Beispiel: Die Bahnschranke geht niemals nach oben, solange ein Zug den Übergang passiert.

- *Lebendigkeitseigenschaften:* Beispiel: Die Bahnschranke geht irgendwann wieder nach oben.

Vor dem Entwurf eines Systems ist meist nur eine informelle Spezifikation gegeben, obwohl für die Verifikation eine formelle Spezifikation benötigt wird. Da informelle Spezifikationen oftmals nicht eindeutig, unvollständig, unstrukturiert oder gar widersprüchlich sind, kann es zu Problemen bei der Umsetzung in eine formelle Spezifikation kommen. Fehler in der Spezifikation müssen unbedingt vermieden werden, da die spätere Verifikation nur die Korrektheit einer Implementierung gegen die formelle Spezifikation beweisen kann. Die Erstellung der Implementierungsbeschreibung im zur Verifikation geeigneten Formalismus bereitet normalerweise keine Schwierigkeiten, falls eine formale Semantik der Sprache vorliegt. In dieser Arbeit wird dies exemplarisch für PURR gezeigt.

Bei der Verifikation arbeitet man nur noch mit Beschreibungen von Objekten, aber nicht mehr mit den Objekten selber. Daher benötigt man ein möglichst genaues Modell des Objektes. Dieses Modell muss in einem Formalismus vorliegen, nur die Eigenschaften umfassen, die für die konkreten Anforderungen notwendig sind, aber ausreichend genau sein, um alle wichtigen Eigenschaften zu erfassen. Wichtig ist hierbei, dass das Modell nicht zu genau ist, damit die Verifikation noch in sinnvoller Zeit durchgeführt werden kann. Andererseits darf das Modell auch nicht zu wenig Informationen enthalten, damit keine wichtigen Eigenschaften vergessen werden.

Hat man alle benötigten Modelle, die Spezifikation und die Implementierung formalisiert, dann muss noch der eigentliche Korrektheitsbeweis durchgeführt werden. Dieser Schritt hängt stark vom zugrundeliegenden Formalismus ab. Wichtige Punkte für die Wahl des Formalismus sind z.B. Korrektheit und Vollständigkeit der Beweismethode, der Grad der Automatisierbarkeit und die Komplexität der verwendeten Algorithmen. Besonders wichtig bei der Wahl der Logik sind ihre Ausdrucksstärke sowie die Möglichkeit zur Automatisierung von Beweisen. Die Ausdrucksstärke bestimmt welche Objekte und Datentypen in einer Logik eingesetzt werden können. Aussagenlogiken sind z.B. nicht in der Lage, die natürlichen Zahlen zu formalisieren. Zur Modellierung von reaktiven Systemen ist es darüberhinaus auch notwendig, Zeitverhalten zu formalisieren.

Ein Hauptziel der Verifikation besteht darin, dem Entwerfer nutzbare Verifikationswerkzeuge zur Verfügung zu stellen. Wichtige Kriterien für solche Verifikationswerkzeuge sind gut lesbare Spezifikationen, Möglichkeiten der Abstraktion, ein hoher Grad an Automatisierung, die Integration in kommerzielle Entwurfswerkzeuge, die Verifizierbarkeit realer Entwürfe sowie die

Eignung des Werkzeugs, auch komplexe Entwürfe verifizieren zu können.

Bei all den Vorteilen, die die formale Verifikation bietet, kann auch sie keine vollkommenen und fehlerlosen Systeme garantieren, denn:

- Verifikation erlaubt nur die Entdeckung von Spezifikations- oder Entwurfsfehlern, nicht jedoch die von Fertigungs- oder Betriebsfehlern.
- Verifikation vergleicht ein System mit einer formalen Spezifikation, die möglicherweise nicht das gewünschte Verhalten repräsentiert.
- Die Spezifikation kann unvollständig sein oder Widersprüche enthalten.
- Die Modellierung kann inkonsistent sein oder Widersprüche enthalten
- Die Modellierung kann falsch sein.
- Die Modellierung kann nicht exakt genug sein.
- Die Beweiswerkzeuge können fehlerhaft sein.

Bei der Hardware-Verifikation gibt es im wesentlichen zwei Ansätze [7]:

- *Modellbasierte Ansätze* verwenden meist temporale Logiken [5], μ-Kalkül [14, 8, 9] oder endliche Automaten [18] zur Formalisierung. Hier existieren entscheidbare Kalküle, weshalb automatische Beweisverfahren möglich sind. Für die temporalen Aussagenlogiken kommt aus Komplexitätsgründen oft die Logik CTL zum Einsatz. Bei größeren Systemen stoßen diese Verfahren schnell an ihre Grenzen. Die geringe Ausdrucksmächtigkeit erlaubt auch keine komplexen Datentypen. Der große Vorteil dieser Verfahren ist jedoch ihre vollständige Automatisierbarkeit.
- *Beweisbasierte Ansätze* verhalten sich völlig anders als die modellbasierten. Hier ist die Verwendung komplexer Datentypen möglich. Diese Verfahren sind jedoch nicht mehr automatisierbar. Die Beweise müssen interaktiv durchgeführt werden. Der Entwerfer muss also selbst mit Logiken und Beweisen umgehen, um eine Verifikation durchzuführen. Die beweisbasierten Ansätze sind überall dort sinnvoll, wo auch komplexe Datentypen verifiziert werden müssen. Beispiele hierfür sind HOL [6] oder PVS [13].

Die Wahl des Ansatzes zur Verifikation ist stark vom konkreten Problem abhängig. In dieser Arbeit wird ein modellbasierter Ansatz auf Basis der temporalen Logik CTL verwendet, da ja unter anderem ein automatischer Modellprüfer für die temporale Logik CTL implementiert werden soll.

2.5 CTL

Um das Verhalten von Schaltungen und damit auch von Automaten beschreiben zu können, benötigt man eine Spezifikationssprache die auch allgemeine Aussagen über die Zustandssequenzen bzw. die Ausgabesequenzen zulässt. Hierzu benötigt man Zeitoperatoren, denn es muss möglich sein, über verschiedene Zeitpunkte zu argumentieren.

Die möglichen Ausgabesequenzen, die ein System erzeugen kann, müssen möglichst einfach dargestellt werden, wozu die Belegungen der einzelnen Variablen zu den verschiedenen Zeitpunkten festgelegt werden müssen. Es muss also ein geeignetes Zeitmodell gewählt werden. Hierzu bieten sich folgende Möglichkeiten an:

- *Lineare oder verzweigende „Zeit":* Bei der ersten Variante geht man von einem linearen Zeitmodell aus. Jeder Zustand eines Systems kann also nur einen Folgezustand haben. Bei der anderen Variante kann es zu einem Systemzustand mehrere Folgezustände geben, die die möglichen zukünftigen Entwicklungen berücksichtigen.

- *Zeitpunkte oder Zeitintervalle:* Die temporalen Operatoren können sich auf einzelne Zeitpunkte oder ganze Zeitintervalle beziehen.

- *Diskrete „Zeit" oder kontinuierliche „Zeit":* Beim diskreten Zeitmodell entspricht ein einzelner Moment dem augenblicklichen Systemzustand und der Folgezustand entspricht dann dem nächsten Moment. Diskrete Zeit lässt sich mit Hilfe der natürlichen Zahlen modellieren. Es gibt auch Temporallogiken, die die kontinuierliche Zeit, welche mit Hilfe der reellen Zahlen modelliert werden kann, als Grundlage verwenden.

- *Vergangenheit oder Zukunft:* Hierbei stellt sich die Frage, ob die temporalen Operatoren sich nur auf die Zukunft oder auch auf die Vergangenheit beziehen sollen. Technische Systeme haben jedoch oftmals einen ausgezeichneten Startzustand, so dass oft nur die Temporallogiken zum Einsatz kommen, die sich nur auf die Zukunft beziehen.

Meist werden Systeme ab einem definierten Startzustand betrachtet, die ihre Berechnungen in einzelnen Schritten ausführen. Daher werden hier häufig nur Logikvarianten benötigt, die auf einem diskreten Zeitmodell mit Zeitpunkten beruhen. Man benötigt also temporale Aussagenlogiken wie CTL, LTL oder CTL* [5]. Der Unterschied zur Aussagenlogik liegt darin, dass die Belegungen der Variablen, die eine temporallogische Formel erfüllen können, nicht mehr statisch sind. Statt dessen werden hier Folgen von Werten für jede Variable betrachtet. In dieser Arbeit wird im Folgenden nur die Logik CTL betrachtet. Bei der in dieser Arbeit verwendeten Logik CTL handelt es sich um eine Logik mit verzweigendem Zeitmodell.

Um das Zeitmodell zu formalisieren, werden temporale Strukturen verwendet. Diese Strukturen, die auch als Kripke Strukturen bezeichnet werden, sind wie folgt definiert:

Definition 5 *Eine temporale Struktur* $\mathcal{M} := (\mathcal{S}_0, \mathcal{S}, \mathcal{R}, \mathcal{L})$ *besteht aus*

1. *einer Menge von endlich vielen Zuständen* $\mathcal{S}$*; wobei* $\mathcal{S}_0 \subseteq \mathcal{S}$ *die Menge der Initialzustände ist*
2. *einer totalen Übergangsrelation* $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ *mit* $\forall s \in \mathcal{S} \exists s' \in \mathcal{S}.(s, s') \in \mathcal{R}$
3. *einer Markierungsfunktion* $L : \mathcal{S} \rightarrow \wp(\mathcal{V})$*, wobei* $\mathcal{V}$ *die Menge der aussagenlogischen Variablen und* $\wp(\mathcal{V})$ *die Menge aller Teilmengen von* $\mathcal{V}$ *ist*

Eine solche Struktur ist also ein Graph, bei dem die Knoten durch $\mathcal{S}$ und die Kanten durch $\mathcal{R}$ gegeben sind. Die Knoten sind dabei durch $\mathcal{L}$ mit Variablen markiert. Wenn man die temporale Struktur ausgehend von einem Startzustand durchläuft, dann bekommt man einen unendlichen Baum, den sogenannten Berechnungsbaum des durch die Struktur dargestellten Systems. Die Struktur repräsentiert also das verzweigende Zeitmodell. Jeder Zustand beschreibt einen Systemzustand und ist mit den Variablen markiert, die in dem Zustand gelten sollen. Die Markierung entspricht sowohl der momentanen Eingabebelegung sowie der momentanen Belegung der internen Variablen. Ein Beispiel für eine solche Struktur zeigt die folgende Abbildung in der der Startzustand durch eine zusätzliche Linie gekennzeichnet ist:

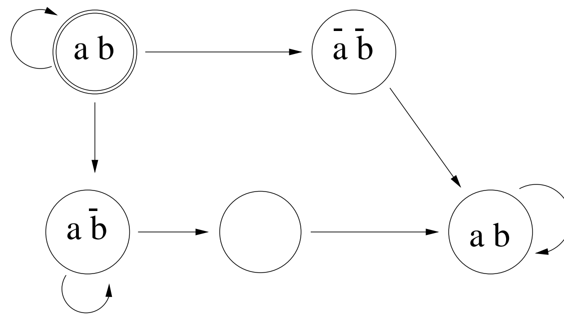

Abbildung 2.1: Kripke Struktur zur Systemmodellierung

Es ist möglich, dass verschiedene Zustände dieselben Variablen aufweisen (siehe Abb. 2.1)

Definition 6 *Ein Pfad* π *einer Struktur* $\mathcal{M}$ *ist eine unendliche Folge* $\pi := < s^0, s^1, \cdots >$ *von Zuständen, wobei gelten muss,* $\forall i \geq 0.(s^i, s^{(i+1)}) \in \mathcal{R}$.

Jeder Pfad entspricht dem Ablauf des Systems unter einer gewissen Eingabesequenz, welche in der Sequenz $L(s^0), L(s^1) \cdots$ enthalten ist. Es wird davon ausgegangen, dass eine Transition

eine Zeiteinheit verbraucht. Temporale Logiken erlauben nun, Aussagen über diese Systemzustände bzw. Pfade zu machen.

Liegt eine Implementierung nun als temporale Struktur vor, so muss gezeigt werden, dass alle durch die Struktur gegebenen Sequenzen die temporallogische Formel erfüllen. Dieses Problem nennt sich auch Modellprüfung, da Strukturen welche eine Formel erfüllen, oft als Modelle dieser Formel bezeichnet werden. Für die Modellprüfung wird in dieser Arbeit die Logik CTL eingesetzt, weshalb diese nun im Folgenden behandelt wird.

Die Logik CTL verwendet neben den aussagenlogischen Operatoren zwei Arten von modalen Operatoren, die dem zugrundeliegenden Zeitmodell entsprechen. Die eine Art der Operatoren macht Aussagen über einen bestimmten Pfad, d.h. Aussagen über zeitliche Beziehungen zwischen Variablen auf den Zuständen. Eine Formel kann für jeden Zustand eines Pfades gelten (G oder *generally*), nur für mindestens einen möglichen Folgezeitpunkt des Pfades (F oder *sometimes*) oder man kann über den unmittelbaren Folgezustand Aussagen machen (X oder *next*). Man kann außerdem zwei Formeln derart miteinander verknüpfen, dass die erste solange wahr sein muss, bis die zweite gilt (U oder *until*).

Die zweite Art von Operatoren macht Aussagen darüber, ob die Formel auf allen Folgepfaden (A oder *for All paths*) oder nur auf einem einzigen Folgepfad (E oder *there Exists a path*) gelten muss. Im Gegensatz zu CTL* dürfen in CTL Pfadquantoren und temporale Operatoren nur in Paaren auftreten.

Definition 7 (Syntax von CTL) *Sei $\mathcal{V}$ die Menge der aussagenlogischen Variablen, mit denen die Kripke Struktur markiert ist, dann definiert man die Menge der CTL-Formeln rekursiv:*

1. *Ist $\phi \in \mathcal{V}$, so ist ϕ eine CTL-Formel.*
2. *Sind und ϕ und ψ CTL-Formeln, so sind $\neg\phi$ und $\phi \vee \psi$ CTL-Formeln.*
3. *Sind und ϕ und ψ CTL-Formeln, so sind $EX\phi$ und $E(\phi \;\underline{U}\; \psi)$ CTL-Formeln.*

Die semantische Folgerbarkeit $\models$ ist wie folgt festgelegt: Die Notation $\mathcal{M}, s \models \phi$ bedeutet, dass die CTL-Formel ϕ im Zustand s der Kripke Struktur $\mathcal{M}$ erfüllt ist. Falls eindeutig ist, welche Kripke Struktur zu Grunde liegt, schreibt man auch $s \models \phi$.

Definition 8 (Semantik von CTL) *Sei $\mathcal{V}$ die Menge der aussagenlogischen Variablen, mit denen die Kripke Struktur $\mathcal{M}$ markiert ist sowie S die Menge aller Zustände in $\mathcal{M}$, dann definiert man die Semantik der CTL-Formeln rekursiv:*

1. *Für alle $s \in S$ gilt:*
   $s \models true$
   $s \not\models false$

2. *Für alle atomaren Formeln $v \in \mathcal{V}$ gilt:*
   $s \models v \Leftrightarrow v \in L(s)$

3. *Sind ϕ und ψ CTL-Formeln, so gilt:*
   $s \models \neg\phi \quad \Leftrightarrow s \not\models \phi$
   $s \models \phi \wedge \psi \quad \Leftrightarrow s \models \phi$ *und* $s \models \psi$
   $s \models \phi \vee \psi \quad \Leftrightarrow s \models \phi$ *oder* $s \models \psi$
   $s \models \phi \rightarrow \psi \quad \Leftrightarrow s \not\models \phi$ *oder* $s \models \psi$
   $s \models \phi \leftrightarrow \psi \quad \Leftrightarrow (s \models \phi$ *und* $s \models \psi)$ *oder* $(s \not\models \phi$ *und* $s \not\models \psi)$

4. *Sind ϕ und ψ CTL-Formeln, so gilt:*
   $s \models EX\phi \quad \Leftrightarrow$ *Es existiert ein Zustand $s^1 \in S$ mit $(s^0, s^1) \in R$ und $s^1 \models \phi$*
   $s \models E(\phi \,\underline{U}\, \psi) \quad \Leftrightarrow$ *Es existiert ein Pfad $s^0, s^1, \cdots$ in $\mathcal{M}$ und ein $k \in \mathbb{N}$ mit $s^0 \models \phi, \cdots, s^{k-1} \models \phi$ und $s^k \models \psi$*
   $s \models EG\phi \quad \Leftrightarrow$ *Es existiert ein Pfad $s^0, s^1, \cdots$ in $\mathcal{M}$, so dass für alle $k \in \mathbb{N}$ gilt $s^k \models \phi$*
   $s \models EF\phi \quad \Leftrightarrow$ *Es existiert ein Pfad $s^0, s^1, \cdots$ in $\mathcal{M}$ und ein $k \in \mathbb{N}$ mit $s^k \models \phi$*
   $s \models AX\phi \quad \Leftrightarrow$ *Für alle Zustände $s^1 \in S$ mit $(s^0, s^1) \in R$ gilt $s^1 \models \phi$*
   $s \models A(\phi \,\underline{U}\, \psi) \quad \Leftrightarrow$ *Für alle Pfade $s^0, s^1, \cdots$ in $\mathcal{M}$ und ein $k \in \mathbb{N}$ mit $s^0 \models \phi, \cdots, s^{k-1} \models \phi$ und $s^k \models \psi$*
   $s \models AG\phi \quad \Leftrightarrow$ *Für alle Pfade $s^0, s^1, \cdots$ in $\mathcal{M}$ so dass für alle $k \in \mathbb{N}$ gilt $s^k \models \phi$*
   $s \models AF\phi \quad \Leftrightarrow$ *Für alle Pfade $s^0, s^1, \cdots$ in $\mathcal{M}$ und ein $k \in \mathbb{N}$ mit $s^k \models \phi$*

Zur besseren Schreibweise wird zusätzlich noch der Und-Operator eingeführt:

$$\psi \wedge \phi :\Leftrightarrow \neg(\neg\phi \vee \neg\psi)$$

Weitere Operatoren lassen sich alle mit EX und E$\underline{U}$ ausdrücken.

- $AX\phi \leftrightarrow \neg EX(\neg\phi)$: für alle unmittelbaren Folgezustände gilt ϕ
- $AG\phi \leftrightarrow \neg E(true\ \underline{U}\ (\neg\phi))$: für alle Pfade gilt immer ϕ
- $AF\phi \leftrightarrow A(true\ \underline{U}\ \phi)$: für alle Pfade gilt in mindestens einem Zustand ϕ
- $EF\phi \leftrightarrow E(true\ \underline{U}\ \phi)$: es existiert ein Pfad, auf dem mindestens einmal ϕ gilt
- $A(\phi \underline{U} \psi) \leftrightarrow \neg E(\neg\psi \underline{U} \neg\phi \wedge \neg\psi) \wedge \neg EG(\neg\psi)$: für alle Pfade gilt solange ϕ bis ψ gilt
- $EG\phi \leftrightarrow \neg A(true\ \underline{U}\ (\neg\phi))$: es gibt einen Pfad auf dem immer ϕ gilt

Abbildung 2.2 veranschaulicht diese acht CTL-Operatoren, wobei der Zustand s^0 jeweils der Startzustand ist.

Es gibt noch weitere CTL-Operatoren, die mit Hilfe der bisher vorgestellten Operatoren definiert werden können:

$$
\begin{array}{lcl}
E(\phi\ U\ \psi) & \Leftrightarrow & E[\phi\ \underline{U}\ \psi] \vee EG\ \phi \\
E(\phi\ \underline{W}\ \psi) & \Leftrightarrow & E[(\neg\ \psi)\ \underline{U}\ (\phi \wedge \psi)] \\
E(\phi\ W\ \psi) & \Leftrightarrow & E[(\neg\ \psi)\ U\ (\phi \wedge \psi)] \\
E(\phi\ \underline{B}\ \psi) & \Leftrightarrow & \neg\ E[(\neg\ \phi)\ \underline{U}\ \psi] \\
E(\phi\ B\ \psi) & \Leftrightarrow & \neg\ E[(\neg\ \phi)\ U\ \psi]
\end{array}
$$

Die Semantik dieser Operatoren wird nun beschrieben:

$$
\begin{array}{lcl}
s \models E(\phi\ U\ \psi) & \Leftrightarrow & \text{Es existiert ein Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M} \text{ und ein } k \in \mathbb{N} \text{ mit} \\
 & & s^0 \models \phi, \cdots, s^{k-1} \models \phi \text{ und } s^k \models \psi \text{ oder es existiert ein} \\
 & & \text{Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M}\text{, auf dem für alle } s^i \text{ gilt: } s^i \models \phi \\
s \models E(\phi\ \underline{W}\ \psi) & \Leftrightarrow & \text{Es existiert ein Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M} \text{ und ein } k \in \mathbb{N} \text{ mit} \\
 & & s^k \models (\phi \wedge \psi) \text{ und } s^j \not\models \psi \text{ für alle } j < k. \\
s \models E(\phi\ W\ \psi) & \Leftrightarrow & \text{Es existiert ein Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M} \text{ und ein } k \in \mathbb{N} \text{ mit} \\
 & & s^k \models (\phi \wedge \psi) \text{ und } s^j \not\models \psi \text{ für alle } j < k \text{ oder es existiert} \\
 & & \text{ein Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M}\text{, auf dem für alle } s^i \text{ gilt: } s^i \models \neg\psi \\
s \models E(\phi\ \underline{B}\ \psi) & \Leftrightarrow & \text{Es existiert ein Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M} \text{ und ein } k \in \mathbb{N} \\
 & & \text{mit } s^k \models \phi \text{ und } s^j \not\models \psi \text{ für alle } j \leq k \\
s \models E(\phi\ B\ \psi) & \Leftrightarrow & \text{Es existiert ein Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M} \text{ und ein } k \in \mathbb{N} \text{ mit} \\
 & & s^k \models \phi \text{ und } s^j \not\models \psi \text{ für alle } j \leq k \text{ oder es existiert ein} \\
 & & \text{Pfad } s^0, s^1, \cdots \text{ in } \mathcal{M}\text{, auf dem für alle } s^i \text{ gilt: } s^i \models \neg\psi
\end{array}
$$

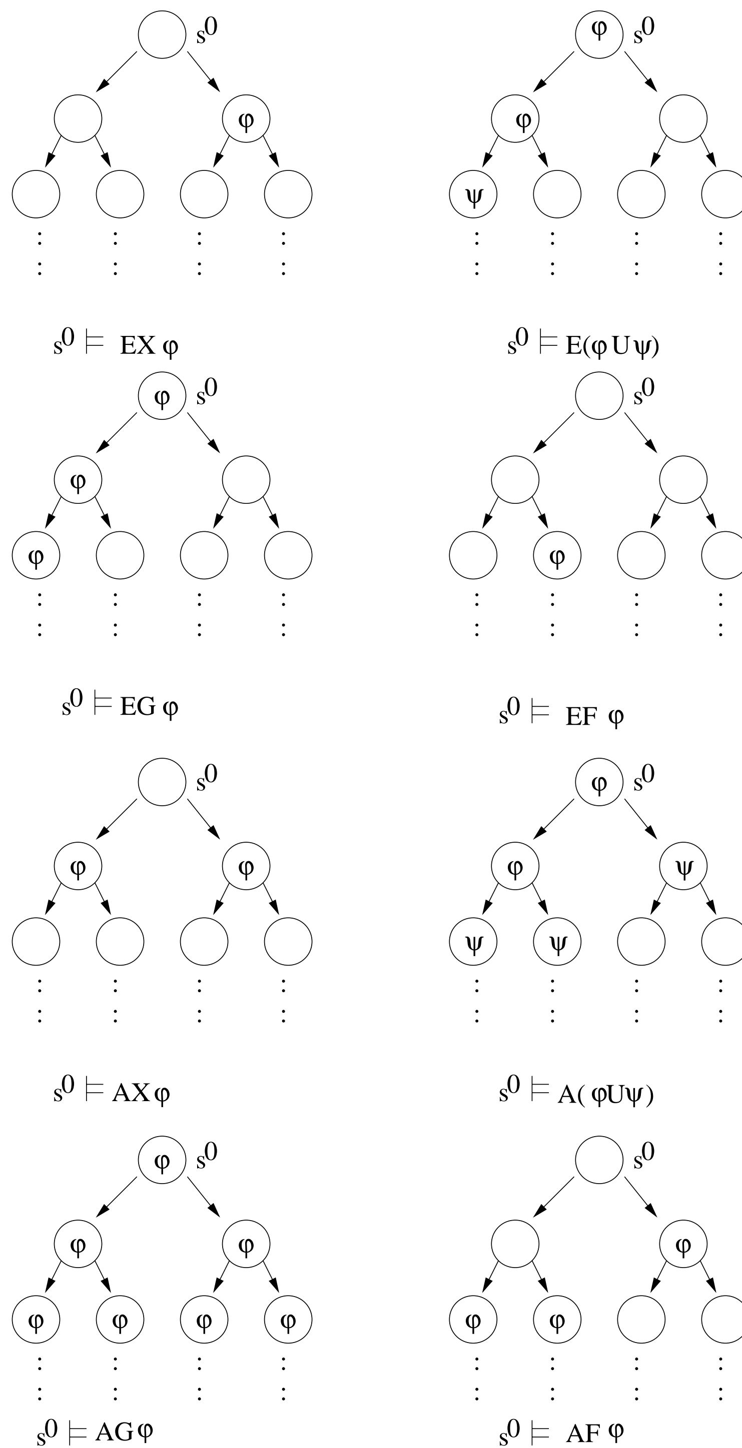

Abbildung 2.2: Die CTL-Operatoren

Definition 9 (Erfülbarkeit) *Sei ϕ eine CTL-Formel. Ist eine Struktur $\mathcal{M}$ gegeben, so heißt die Formel ϕ gültig in $\mathcal{M}$, wenn ϕ in allen Startzuständen von $\mathcal{M}$ gilt.*

Man unterscheidet nun zwischen lokaler und globaler Modellprüfung: Bei der lokalen Modellprüfung muss gezeigt werden, dass $\mathcal{M}, s \models \phi$ gilt und bei der globalen Modellprüfung muss die Menge $\{s \in \mathcal{S} | \mathcal{M}, s \models \phi\}$ bestimmt werden. Bei der lokalen Modellprüfung muss nicht unbedingt der ganze Zustandsraum durchlaufen werden.

Für die Logik CTL existieren effiziente globale Modellprüfungsverfahren, die auf Fixpunktcharakterisierungen der temporalen Operatoren beruhen. Grundlage dieser Fixpunktcharakterisierungen sind Funktionen, die Mengen von Zuständen einer Struktur wieder auf Zustandsmengen abbilden.

Diese Fixpunktcharakterisierungen sind die Grundlage der globalen Modellprüfung: Sie ordnen bei einer gegebenen Struktur $\mathcal{M}$ der CTL-Formel die Menge aller Zustände der Struktur zu, in denen die Formel gilt. Die beiden CTL-Operatoren *AX* und *EX* machen lediglich Aussagen über direkte Folgezustände. Daher benötigt man für diese beiden Operatoren keine Fixpunktiteration. Für die anderen CTL-Operatoren gilt:

- $A(\phi\ U\ \psi) = fp_{min}(Z, \psi \vee (\phi \wedge AX\ Z))$
- $E(\phi\ U\ \psi) = fp_{min}(Z, \psi \vee (\phi \wedge EX\ Z))$
- $AF\phi = fp_{min}(Z, \phi \vee AX\ Z)$
- $EF\phi = fp_{min}(Z, \phi \vee EX\ Z)$
- $AG\phi = fp_{max}(Z, \phi \wedge AX\ Z)$
- $EG\phi = fp_{max}(Z, \phi \wedge EX\ Z)$

Die Bedeutung der Operatoren wird in zwei Teile zerlegt. Im ersten Teil wird nur eine Aussage über den augenblicklichen Zustand getroffen, daher werden keine temporalen Operatoren benötigt. Im zweiten Teil werden dann Aussagen über das gewünschte Verhalten auf direkte Folgezustände gemacht, wobei hier auf Rekursion zurückgegriffen wird.

Beispiel:

Die Formel $EF\phi$ wird dann wahr, wenn es in der Struktur einen Zustand gibt, in dem ϕ gilt. In der Fixpunktgleichung wird dies dadurch ausgedrückt, dass entweder im momentanen Zustand ϕ gelten muss, oder dass es einen Folgezustand gibt, in dem wieder irgendwann $EF\phi$ gelten muss: $EF\phi = \phi \vee (EF\phi)$.

Die Fixpunkte lassen sich durch einfache Iterationen implementieren. Es gilt:

function $fp_{min}(f)$	**function** $fp_{max}(f)$
$Q_1 := \emptyset; Q_2 := f(\emptyset);$	$Q_1 := \mathcal{S}; Q_2 := f(\mathcal{S});$
while $Q_1 \neq Q_2$ **do**	**while** $Q_1 \neq Q_2$ **do**
$Q_1 := Q_2;$	$Q_1 := Q_2;$
$Q_2 := f(Q_2);$	$Q_2 := f(Q_2);$
end	**end**
return Q_1;	**return** Q_1;

Beispiel:

Die Funktion für die Fixpunktiteration der CTL-Formel $EG\phi$ sieht wie folgt aus:

function $EG(\phi)$
$Q_1 := \mathcal{S}; Q_2 := \phi \wedge EX(EG\phi);$
while $Q_1 \neq Q_2$ **do**
$Q_1 := Q_2;$
$Q_2 := \phi \wedge EX(EG\phi);$
end
return Q_1;

Diese Zustandsmengen werden durch charakteristische Funktionen repräsentiert, die selbst wieder boolesche Funktionen sind. Man benötigt also eine effiziente Darstellung von booleschen Funktionen. Diese effiziente Darstellung kann durch binäre Entscheidungsdiagramme geschehen, die in Abschnitt 2.7 erläutert werden.

2.6 Modellprüfung

Hat man eine CTL-Formel und eine Struktur gegeben, ist es Aufgabe der globalen Modellprüfung alle Zustände der Struktur zu finden, in denen diese Formel gilt. Grundlage des in dieser Arbeit verwendeten CTL-Modellprüfungsverfahrens ist die gegebene Sematik der CTL-Operatoren sowie die zu den Operatoren gehörenden Fixpunktcharakterisierungen.

Um nun zu einer gegebenen CTL-Formel alle Zustände zu bestimmen, in denen die Formel gilt, werden ausgehend von den Blättern des Syntaxbaums der Formel, nacheinander alle die Zustandsmengen bestimmt, in denen die jeweilige Teilformel gilt, welche zu diesem Unterbaum gehört. Wenn die gesamte Formel abgearbeitet wurde, dann muss noch überprüft werden, ob alle Startzustände in der Menge der berechneten Zustände liegen. Wenn dies der Fall ist, dann

ist die gegebene CTL-Formel in der Struktur wahr. Diese Vorgehensweise soll an folgendem Beispiel deutlich gemacht werden:

Beipiel:

Sei die Kripke Struktur $\mathcal{M}$ gegeben. Es soll gezeigt werden, dass $EF\ b$ in der Struktur $\mathcal{M}$ gilt. Die Iterationsfunktion für $EF\ b$ ist $f(EF\ b) = b \vee EF\ b$

Es gilt: $f^1(\emptyset) = \{s^2\}$, $f^2(\emptyset) = \{s^2, s^1\}$, $f^3(\emptyset) = \{s^2, s^1, s^0\}$, $f^4(\emptyset) = f^3(\emptyset)$

Da $s^0 \in f^3(\emptyset)$ gilt, ist die Formel also in der Struktur $\mathcal{M}$ wahr.

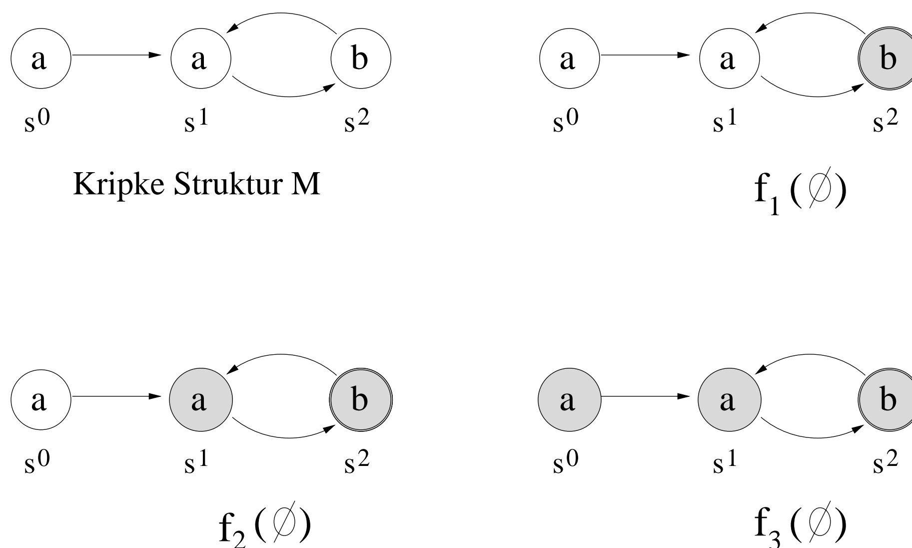

Abbildung 2.3: Ablauf einer Modellprüfung

Es gibt Fälle, in denen eine CTL-Formel nur auf speziellen, sog. fairen Pfaden einer Struktur betrachtet werden soll. Ein Pfad ist fair, wenn gewisse Teilmengen von Zuständen von ihm unendlich oft besucht werden. Modellprüfung mit Fairness kann z.B. benutzt werden, um Terminierung oder Konsistenz von PURR-Programmen zu entdecken. Ein solches Programm mit einem Konsistenzproblem ist z.B. das Programm P_1:

$$P_1 : \textbf{\textit{present}}\ \ o\ \ \textbf{\textit{else}}\ \ \textbf{\textit{emit}}\ \ o\ \ \textbf{\textit{end}}$$

Nimmt man bei diesem Programm an, dass o präsent ist, führt die Annahme dazu, dass o nicht ausgegeben wird. Andererseits wäre es möglich, dass o nicht präsent ist, was aber dazu führt,

dass o ausgegeben wird. Die Kripke Struktur zu diesem Programm enthält keinen unendlichen Pfad, was sich mit fairer Modellprüfung rasch entdecken lässt.

Die faire Modellprüfung kann auf die normale Modellprüfung ohne Fairness und die faire Prüfung von $E\phi G\varphi$ zurückgeführt werden. Die Fixpunktiteration von $E\phi G\varphi$ ist wie folgt definiert:

function $E_\phi G\varphi$ $(Q_1, \cdots, Q_f, \varphi)$
  $P_1 := true$;
  repeat
    $P_0 := P_1$;
    for $i := 1$ to f **do**
      $Q_i := Q_1 \cap P_1$;
      $R_{i,1:=Q_1}$;
      repeat
        $R_{i,0} := R_{i,1}$;
        $R_{i,1} := Q_i \cup (predecessor^R_\exists(R_{i,0} \cap \varphi))$;
      until $R_{i,1} = R_{i,0}$
    end
    $P_1 := \varphi \cap \bigcap_{i=1}^{f} predecessor^R_\exists(R_{i,0})$;
  until $P_1 = P_0$
  return P_1;

Hierbei sind die Q_i die Zustandsmengen von denen mindestens ein Zustand unendlich oft durchlaufen werden muss und φ die zu prüfende CTL-Formel. Weiterhin wird noch eine Funktion ϕ_{fair} benötigt, welche die Menge der Zustände bestimmt,von denen ein Pfad ausgeht, auf dem von jeder Menge Q_i mindestens ein Zustand unendlich oft durchlaufen wird. Diese Funktion lässt sich auf die Fixpunktiteration für $E_\phi G\varphi$ zurückführen als $\phi_{fair} := E_\phi G\ true$.

Alle anderen Operatoren für die faire Modellpüfung lassen sich wie folgt auf die Operatoren für die Modellpüfung ohne Fairness zurückführen:

$$
\begin{array}{lcl}
E_\phi X\varphi & = & EX(\phi_{fair} \wedge \varphi) \\
E_\phi F\varphi & = & EF(\phi_{fair} \wedge \varphi) \\
E_\phi(\varphi\ \underline{U}\ \psi) & = & E[\varphi\ \underline{U}\ (\psi \wedge \phi_{fair})] \\
E_\phi(\varphi\ U\ \psi) & = & E_\phi[\varphi\ \underline{U}\ \psi] \vee E_\phi G\phi \\
E_\phi(\varphi\ \underline{W}\ \psi) & = & E[\varphi\ \underline{W}\ (\psi \wedge \phi_{fair})]
\end{array}
$$

$$
\begin{array}{lcl}
E_\phi(\varphi\, W\, \psi) & = & E_\phi[\varphi\, \underline{W}\, \psi] \vee E_\phi G \neg\psi \\
E_\phi(\varphi\, \underline{B}\, \psi) & = & E[(\varphi \wedge \phi_{fair})\, \underline{B}\, \psi] \\
E_\phi(\varphi\, B\, \psi) & = & E_\phi[\varphi\, \underline{B}\, \psi] \vee E_\phi G(\neg\varphi \wedge \neg\psi) \\
A_\phi X \varphi & = & \neg E_\phi X(\neg\varphi) \\
A_\phi F \varphi & = & \neg E_\phi G(\neg\varphi) \\
A_\phi G \varphi & = & \neg E_\phi F(\neg\varphi) \\
A_\phi(\varphi\, \underline{U}\, \psi) & = & \neg E_\phi[(\neg\varphi)\, B\, \psi] \\
A_\phi(\varphi\, U\, \psi) & = & \neg E_\phi[(\neg\varphi)\, \underline{B}\, \psi] \\
A_\phi(\varphi\, \underline{W}\, \psi) & = & \neg E_\phi[(\neg\varphi)\, W\, \psi] \\
A_\phi(\varphi\, W\, \psi) & = & \neg E_\phi[(\neg\varphi)\, \underline{W}\, \psi] \\
A_\phi(\varphi\, \underline{B}\, \psi) & = & \neg E_\phi[(\neg\varphi)\, U\, \psi] \\
A_\phi(\varphi\, B\, \psi) & = & \neg E_\phi[(\neg\varphi)\, \underline{U}\, \psi]
\end{array}
$$

2.7 Binäre Entscheidungsdiagramme

Die Zustandsmengen, die sich bei der Berechnung der Transitionsrelation eines PURR-Programms ergeben, werden durch aussagenlogische Formeln repräsentiert. Zur Darstellung von aussagenlogischen Formeln in einem Rechner benötigt man eine Darstellungsform, die wenig Speicherplatz benötigt.

Aus einer Wertetabelle kann man den Funktionswert einer booleschen Funktion für eine konkrete Variablenbelegung direkt ablesen. Die Darstellung der gesamten Funktionstabelle erfordert jedoch einen Speicheraufwand der Größenordnung O(2^n). Aussagenlogische Formeln sind zwar eine kompakte Darstellung boolescher Funktionen, hier ist der Erfüllbarkeitstest jedoch NP-vollständig.

Definition 10 *Betrachtet man eine boolesche Funktion $f(x_1, \cdots, x_n)$, so nennt man den Ausdruck $f(x_1, \cdots, x_{i-1}, T, \cdots, x_n)$ den Kofaktor von f nach x_i (geschrieben $f\,|_{x_i}$). Der Ausdruck $f(x_1, \cdots, x_{i-1}, F, \cdots, x_n)$ ist der Kofaktor von f nach $\neg x_i$ (geschrieben $f\,|_{\neg x_i}$).*

Satz 1 (Shannonscher Entwicklungssatz) *Es gilt der Entwicklungssatz:*

$$f(x_1, ..., x_i, ..., x_n) = (x_i \wedge f\,|_{x_i}) \vee (\neg x_i \wedge f\,|_{\neg x_i})$$

Diese Kofaktorbildung lässt sich auch graphisch veranschaulichen:
Die Wirkungsweise des Entwicklungssatzes auf eine boolesche Funktion wird im folgenden Beispiel deutlich.

Gegeben sei die Funktion $f := x_1 \neg x_3 \vee x_2 x_3$. Es gilt $f\,|_{x_3} = x_2$ und $f\,|_\neg x_3 = x_1$ und somit $f = (x_3 \wedge f\,|_{x_3}) \vee (\neg x_3 \wedge f\,|_{\neg x_3}) = (x_3 \wedge x_2) \vee (\neg x_3 \wedge x_1)$.

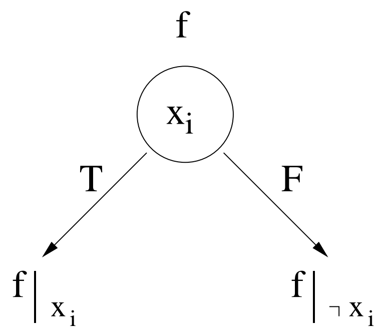

Abbildung 2.4: Kofaktorisierung einer Funktion f

Man kann also eine Funktion f mit n Variablen auf zwei neue Funktionen mit n-1 Variablen zurückführen.

Satz 2 *Die Verknüpfung zweier boolescher Funktionen f und g durch einen zweistelligen booleschen Operator $\circ$ lässt sich wie folgt durchführen: $f := (f \circ g)\,|_{x_i} = f\,|_{x_i} \circ g\,|_{x_i}$*
Weiterhin gilt: $(\neg f)\,|_{x_i} = \neg(f\,|_{x_i})$

Entwickelt man eine boolesche Funktion nach allen Variablen und verwendet die obige graphische Notation, so erhält man eine Baumdarstellung der Funktion, an deren Blättern die Funktionswerte für die $2n$ verschiedenen Belegungen zu finden sind. Der Baum entspricht also einer vollständigen Wertetabelle. Ein Beispiel für einen solchen Baum findet sich in Abbildung 2.5.

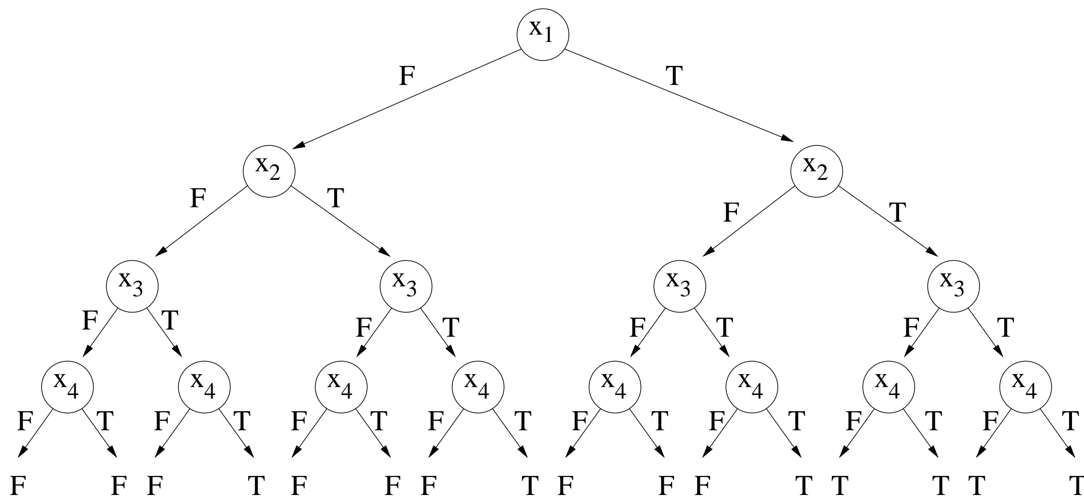

Abbildung 2.5: Vollständiger Baum der Funktion $f = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

Diese Darstellung besitzt gegenüber den Wertetabellen noch keinen nennenswerten Vorteil. Die Baumstruktur kann jedoch redundante Teile, wie z.B. identische Teilbäume oder Entschei-

dungsknoten, die nicht benötigt werden, enthalten, die sich eliminieren lassen. Ein solches Eliminieren ist in Abbildung 2.6 dargestellt.

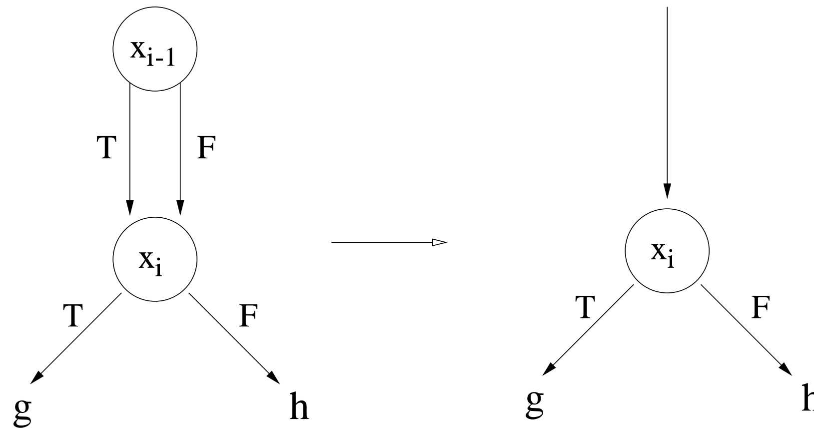

Abbildung 2.6: Entfernen redundanter Entscheidungsknoten

Eliminiert man diese redundaten Teile im Baum aus Abbildung 2.5, so erhält man einen Baum wie in Abbildung 2.7 dargestellt. Diese kompakte Darstellung nennt man ein binäres Entscheidungsdiagramm.

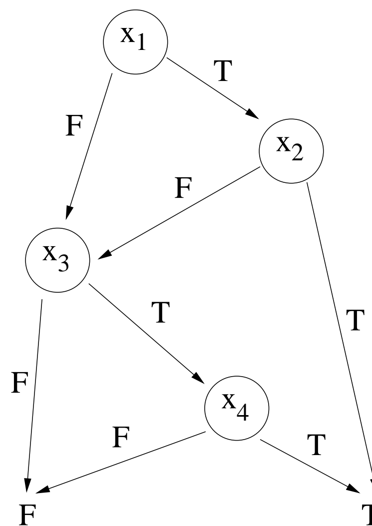

Abbildung 2.7: Reduzierte Darstellung der Funktion aus Abbildung 2.5

Wenn man nun noch fordert, dass beim Durchlaufen des Baumes von der Wurzel zu den Blättern die auf dem Pfad vorhandenen Variablen immer in der gleichen Reihenfolge durchlaufen werden, dann ist das binäre Entscheidungsdiagramm eindeutig. Ein solches binäres Entschei-

dungsdiagramm wird auch als reduziertes geordnetes binäres Entscheidungsdiagramm (Reduced Ordered Binary Decision Diagram, ROBDD) bezeichnet.
Formal ist ein ROBDD wie folgt definiert:

Definition 11 *Ein ROBDD mit n Variablen ist ein gerichteter zyklenfreier Graph $\mathcal{G} := (\mathcal{V}, \mathcal{E})$. Ein Knoten v $\in \mathcal{V}$ ist entweder ein innerer Knoten oder ein Blatt. Ein Blatt hat keine Söhne, aber einen Wert. Ein innerer Knoten hat einen linken Sohn* ***links****: $\mathcal{V} \to \mathcal{V}$ und einen rechten Sohn* ***rechts****: $\mathcal{V} \to \mathcal{V}$ sowie einen Index, dem durch die Abbildung $var : \{1, \cdots, n\} \to \mathcal{V}$ eine Variable aus $\mathcal{V}$ zugeordnet ist. Der Graph ist geordnet. Es gilt also Index(v) < Index(links(v)) und Index(v) < Index(rechts(v)). Die Kanten e $\in \mathcal{E}$ sind alle Paare (v,links(v)) und (v,rechts(v)).*

Definition 12 *Ein ROBDD mit der Wurzel $\in \mathcal{V}$ definiert eine boolesche Funktion f wie folgt:*

$$f(v) := \begin{cases} val(v) & , \text{ : falls v ein Blatt ist} \\ \neg var(Index(v)) \wedge f(links(v)) \vee \\ var(Index(v)) \wedge f(rechts(v)) & , \text{ : sonst} \end{cases}$$

ROBDDs bieten einige Vorteile gegenüber Wertetabellen hinsichtlich dem benötigten Speicherbedarf. Da sie eine Normalformeigenschaft besitzen, ist auch der Tautologietest einfacher als bei aussagenlogischen Formeln: Der Tautologietest lässt sich auf einen Vergleich mit dem 1-Blatt reduzieren. Da geordnete und reduzierte binäre Entscheidungsdiagramme eindeutig sind, lässt sich mit ihnen der semantische Vergleich zweier Formeln durch einen syntaktischen Vergleich der binären Entscheidungsdiagramme durchführen, d.h. syntaktischer und semantischer Vergleich sind hier identisch. Der semantische Vergleich zweier aussagenlogischer Formeln ist dagegen wesentlich aufwendiger als deren syntaktischer Vergleich.

Die Bestimmung des ROBDDs für eine Funktionsverknüpfung lässt sich wie folgt durchführen: Sei $f := f_1 \circ f_2$. Mit Satz 2 gilt:

$$\begin{aligned} f &= (x_i \wedge f\,|_{x_i}) \vee (\neg x_i \wedge f\,|_{\neg x_i}) \\ &= (x_i \wedge (f_1 \circ f_2)\,|_{x_i}) \vee (\neg x_i \wedge (f_1 \circ f_2)\,|_{\neg x_i}) \\ &= (x_i \wedge (f_1\,|_{x_i} \circ f_2\,|_{x_i})) \vee (\neg x_i \wedge (f_1\,|_{\neg x_i} \circ f_2\,|_{\neg x_i})) \end{aligned}$$

Die Negation eines ROBDDs erfolgt durch Komplementierung der Blätter, eine Existenzquantifizierung durch $\exists x_i.f = f\,|_{x_i} \vee f\,|_{\neg x_i}$ und eine Allquantifizierung durch $\forall x_i.f = f\,|_{x_i} \wedge f\,|_{\neg x_i}$.

ROBDDs sind sehr nützlich, um Zustandsmengen und Transitionsrelationen, wie sie bei der CTL-Modellprüfung benötigt werden, zu repräsentieren, da sie hinsichtlich des Speicherbedarfs und der Manipulierbarkeit sehr effiziente Datenstrukturen sind. In den meisten BDD-Paketen sind Funktionen implementiert, die das Erstellen von BDD's und die Manipulation von

BDD's auf sehr einfache Art und Weise ermöglichen. So gibt es z.B. Funktionen, mit denen die konjunktive oder die disjunktive Verknüpfung von zwei BDD's berechnet werden können oder mit denen eine Existenzquantifizierung über einem BDD erfolgen kann. Mit Hilfe dieser Funktionen lassen sich neben den Zustandsmengen und den Transitionsrelationen auch die Fixpunktiterationen einfach beschreiben.

Auch die Berechnung der Vorgängermenge einer Zustandsmenge lässt sich mit diesen Funktionen beschreiben. Sei $Q(q_1, \cdots, q_n)$ eine aussagenlogische Formel, die eine Zustandsmenge Q mit Zuständen $q_1, \cdots, q_n$ beschreibt. Die erfüllenden Belegungen von $Q(q_1, \cdots, q_n)$ sind genau die Zustände, die in Q vorkommen. Sei weiterhin $R(q_1, \cdots, q_n, q'_1, \cdots, q'_n, x_1, \cdots, x_m)$ eine aussagenlogische Formel, die eine Transitionsrelation R mit Transitionen (q_i, q'_i, x_i) beschreibt. Hierbei wird eine Transition (q_i, q'_i, x_i) vom Zustand q_i in den Zustand q'_i unter der Eingabe x_i ausgeführt. Eine solche Transition ist in Abbildung 2.8 dargestellt. Die erfüllenden Belegungen von $R(q_1, \cdots, q_n, q'_1, \cdots, q'_n, x_1, \cdots, x_m)$ sind genau die Zustände, die in R vorkommen.

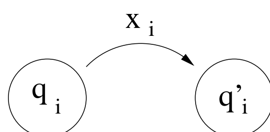

Abbildung 2.8: Transition

Die Menge der Vorgängerzustände von Q lässt sich berechnen durch $\exists q'_1, \cdots, q'_n . Q(q'_1, \cdots, q'_n) \wedge R(q_1, \cdots, q_n, q'_1, \cdots, q'_n, x_1, \cdots, x_m)$. Diese Berechnung, die auch relationales Produkt genannt wird, lässt sich durch Verwenden der Funktionen für die Existenzquantifizierung sowie der Konjunktion zweier BDD's implementieren. In vielen BDD-Paketen ist jedoch bereits eine Funktion für dieses relationale Produkt vorhanden, die sehr viel effizienter ist als eine Implementierung mittels Existenzquantifizierung und Konjunktion von BDD's.

Kapitel 3

Implementierung

3.1 Einlesen eines PURR-Programms

Ein zu verifizierendes PURR-Programm muss folgenden Aufbau haben:

module m
input $x_1 : l_1, \cdots, x_m : l_m$***;***
output $y_1 : k_1, \cdots, y_n : k_n$***;***
S_m
spec $s_1, \cdots, s_n$
end module

Dabei ist ***m*** der Modulname und l_i und k_i sind natürliche Zahlen, die den Typ der Ein- und Ausgaben angeben. $S_m \in \text{PStm}_\Sigma$ ist eine Anweisung, welche als Modulrumpf bezeichnet wird. $s_1, \cdots, s_n$ ist eine Liste von Spezifikationen.

Die Spezifikationen s_i sind von der Form:

S_{name} *CTL-Formel* ***with Fairness*** *CTL-Formel#* $\cdots$ *#CTL-Formel*

Wobei S_{name} der Name der jeweiligen Spezifikation und *CTL-Formel* eine CTL-Formel ist. Die nach dem Schlüsselwort ***with Fairness*** angegebenen CTL-Formeln stellen die Fairnessbedingungen dar. Eine solche Fairnessbedingung Φ ***with Fairness*** $\varphi_1, \cdots, \varphi_n$ wird folgendermaßen ausgewertet: Zunächst werden $S_i := \{s \in S | \mathcal{M}, s \models \varphi_i\}$ ohne jegliche Fairnessbedingungen bestimmt und als faire Teilmenge verwendet. Danach wird dann $\phi_{fair} := \{s \in S | \exists \pi. \pi^0 = s \wedge \bigwedge_{i=1}^{n} \forall t_1 \exists t_2. \pi^{t_1+t_2} \in S_i\}$ bestimmt. Dies geschieht mit den in Abschnitt 2.6 beschriebenen Fixpunktiterationen. Der genaue Aufbau der Spezifikationen ist aus der Grammatik des Parsers ersichtlich, die im Anhang angegeben ist.

Das Einlesen der PURR-Programme ist mit Hilfe der Programme Yacc und Lex realisiert.

3.1.1 Yacc

Yacc ist ein Werkzeug, welches aus einer kontextfreien Grammatik ein C-Programm erzeugt, welches in der Lage ist, Ableitungen zu finden, d.h. zu parsen. Ferner kann der Parsebaum in Yacc auch attributiert werden. Jede Eingabe für ein Programm muss eine gewisse Syntax besitzen, die vom betreffenden Programm vorausgesetzt wird. Diese Syntax einer Eingabe kann sehr einfach sein, wie z.B. eine Folge von Zahlen, die dann von einem Sortierprogramm in die richtige Reihenfolge zu bringen ist. Eine solche Eingabestruktur kann jedoch auch äußerst komplex sein, z.B. so komplex wie die Programmiersprache C. In diesem Fall ist es die Aufgabe des betreffenden Programms, nämlich des C-Compilers, den vorgelegten C-Code zu analysieren und in die Maschinensprache zu übersetzen. Würde man z.B. einem C-Compiler als Eingabe ein FORTRAN-Programm vorlegen, so könnte er damit wenig anfangen, da er nicht für eine solche Eingabestruktur ausgelegt wurde. Die Beschreibung der Eingabestruktur muss dabei in einer Form angegeben werden, die weitgehend der Backus-Naur-Form (BNF) entspricht.

Yacc ist ein Software-Werkzeug, welches sich nicht nur im Compilerbau, sondern auf nahezu jedem Anwendungsgebiet nutzbringend einsetzen lässt, da jede Software eine gewisse Syntax für ihre Eingabe voraussetzt.

Natürlich könnten Aufgaben dieser Art auch ohne Zuhilfenahme von Yacc direkt mit einfachen üblichen höheren Programmiersprachen (wie C oder PASCAL) gelöst werden. Yacc aber bietet den Vorteil, dass es bei Aufgabenstellungen dieser Art dem Programmierer nicht nur viel Arbeit abnimmt, sondern ihn auch zwingt, sich Gedanken über seine Eingabestruktur zu machen und diese zu formalisieren. So können oft bereits in der Design-Phase Widersprüche und Lücken aufgedeckt und dann beseitigt werden. Weitere Vorteile für die praktische Software-Entwicklung beim Arbeiten mit Yacc sind noch: kürzere Entwicklungszeit, bessere Lesbarkeit und leichtere Änderbarkeit.

3.1.2 Lex

Lex wurde als Ergänzung zu Yacc entwickelt. Da man mit Yacc über ein Werkzeug für die syntaktische Analyse verfügte, lag es nahe, sich auch ein Werkzeug für die lexikalische Analyse, welche der Syntaxanalyse vorgeschaltet ist, zu schaffen.

Ein Lex-Programm beschreibt im Wesentlichen eine reguläre Grammatik.

Lex ist eine Art von Übersetzer, der Lex-Beschreibungen in ein C-Programm umsetzt, das dann einen zugehörigen akzeptierenden endlichen Automaten implementiert. Lex-Programme müssen in einer dem Lex verständlichen Sprache geschrieben sein. Diese Programme sind dabei eine Tabelle von regulären Ausdrücken und zugehörigen C-Programmteilen.

Diese Tabelle wird von Lex in ein C-Programm übersetzt, welches einen Eingabetext zeichenweise liest und diesen in Tokens zerlegt, die im Lex-Programm durch vorgegebene reguläre Ausdrücke definiert sind. Zu jedem einzelnen String wird dann der zum entsprechenden regulären Ausdruck gehörige C-Programmteil ausgeführt.

Lex ist ein Software-Werkzeug, welches sich auf vielen Anwendungsgebieten erfolgreich einsetzen lässt, wie z.B. Textverarbeitung, Kommandoprozessoren, Chiffrierungen oder Compilerbau. So könnte man z.B. bei einer Textverarbeitung einen Text auf Rechtschreibung untersuchen lassen, bei der Implementierung von Kommandoprozessoren oder Bau eines Compilers von Lex die im Eingabetext vorhandenen Token extrahieren lassen oder bei der Chiffrierung mit Hilfe von Lex bestimmte Strings in andere Strings umformen lassen. Natürlich könnten alle diese Aufgaben auch ohne Zuhilfenahme von Lex direkt mit einer üblichen höheren Programmiersprache (wie C oder PASCAL) gelöst werden. Lex aber bietet den Vorteil, dass es bei Aufgabenstellungen dieser Art dem Programmierer viel Arbeit abnimmt, da dieser die gesuchten Strings nicht mühsam per Hand aus den Eingabetexten extrahieren und dann analysieren muss, sondern diese über reguläre Ausdrücke von Lex automatisch extrahieren lassen kann. Lex ist nun in der Lage, aus den vorgelegten Lex-Programmen C-Programme zu generieren.

3.1.3 Erzeugen eines Syntaxbaumes zu einem PURR-Programm

Zur weiteren Verarbeitung muss aus einem vorliegenden PURR-Programm mit Hilfe von Lex und Yacc ein Syntaxbaum erzeugt werden. Dazu benötigt man eine Datenstruktur, die aus einem Knoten für die jeweilige eingelesene Anweisung und einer Menge von Nachfolgern für die Argumente der Anweisung besteht. Ein solcher Knoten für die Sequenz ist in der folgenden Abbildung dargestellt.

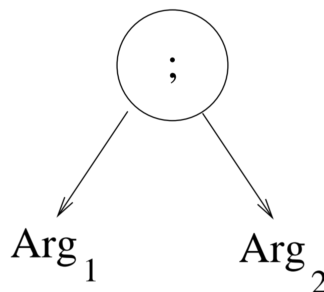

Abbildung 3.1: Knoten für die Sequenz

Wenn man nun bei der Analyse des Programmtextes durch Lex und Yacc für jede Anweisung rekursiv einen solchen Knoten erzeugen lässt, dann erhält man, nachdem das gesamte Programm eingelesen wurde, einen kompletten Syntaxbaum, der das Programm repräsentiert.

Die folgende Abbildung 3.2 zeigt einen solchen Syntaxbaum für das Programm P_2:

P_2: l_0 : ***pause*** ***;*** (**emit** x ***;*** l_1 : ***pause***)$\|$ (***emit*** y ***;*** l_2 : ***pause***)

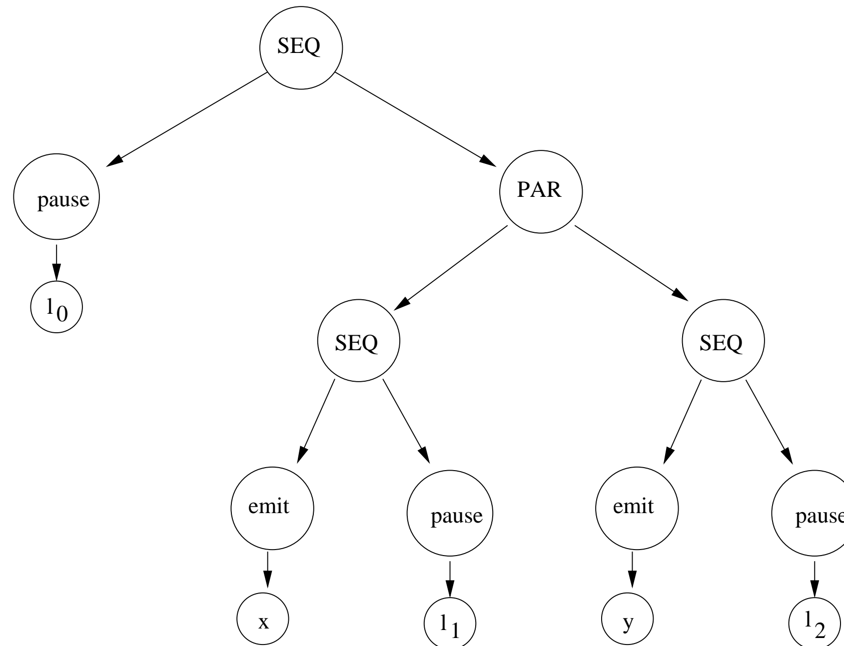

Abbildung 3.2: Syntaxbaum des Programms P_2

Ein solcher Syntaxbaum kann auch zusammengesetzte Anweisungen nach Definition 4 enthalten. In einem nächsten Schritt müssen diese nun entfernt werden.

3.2 Entfernen der Redundanz aus einem PURR-Programm

Das Entfernen der zusammengesetzten Anweisungen aus dem Syntaxbaum eines PURR-Programms geschieht dadurch, dass die Knoten, die eine zusammengesetzte Anweisung nach Definition 4 enthalten, durch einen Baum, der nur noch Knoten mit grundlegenden Anweisungen nach Defintion 3 enthält, ersetzt werden. Dies geschieht mit den Ersetzungen aus Abschnitt 2.4.2. Die folgenden Abbildungen zeigen, wie diese Ersetzungen in dem Syntaxbaum durchgeführt werden. Zu beachten ist hierbei, dass nur in Abbildung 3.7 eine weitere Programmstelle zu markieren ist. Generell sind alle ***pause***-Anweisungen zu markieren, da sich hier der Kontrollfluss befinden könnte.

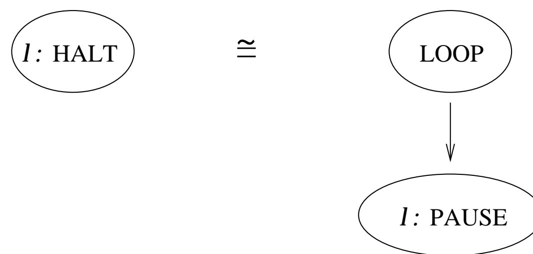

Abbildung 3.3: Redundante Ersetzung der Anweisung ***halt***

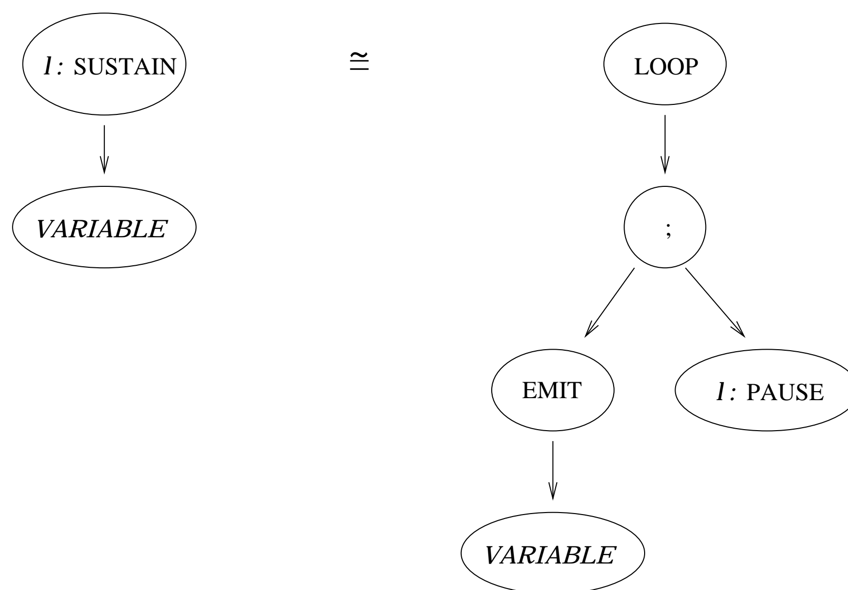

Abbildung 3.4: Redundante Ersetzung der Anweisung ***sustain*** **x**

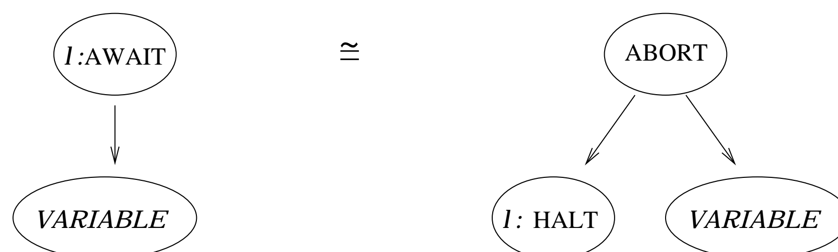

Abbildung 3.5: Redundante Ersetzung der Anweisung ***await*** **x**

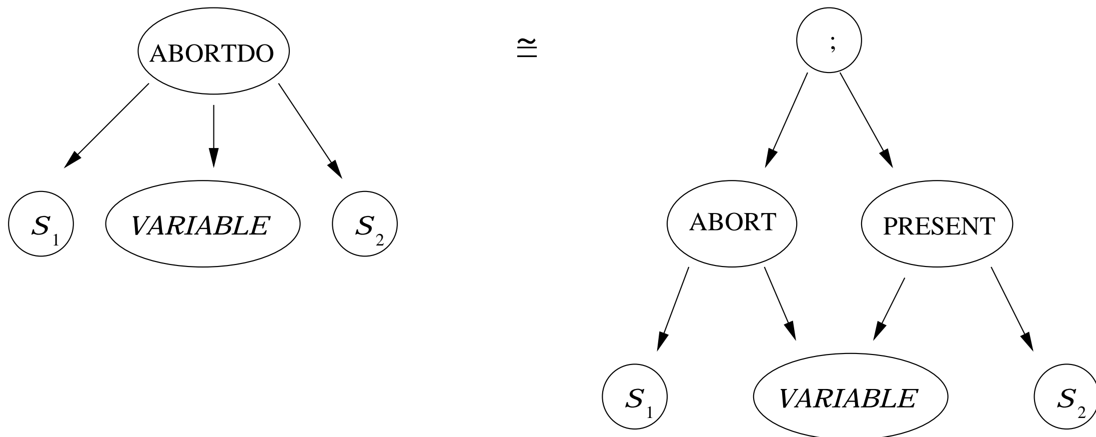

Abbildung 3.6: Redundante Ersetzung der Anweisung ***abort*** S_1 ***when*** x ***do*** S_2

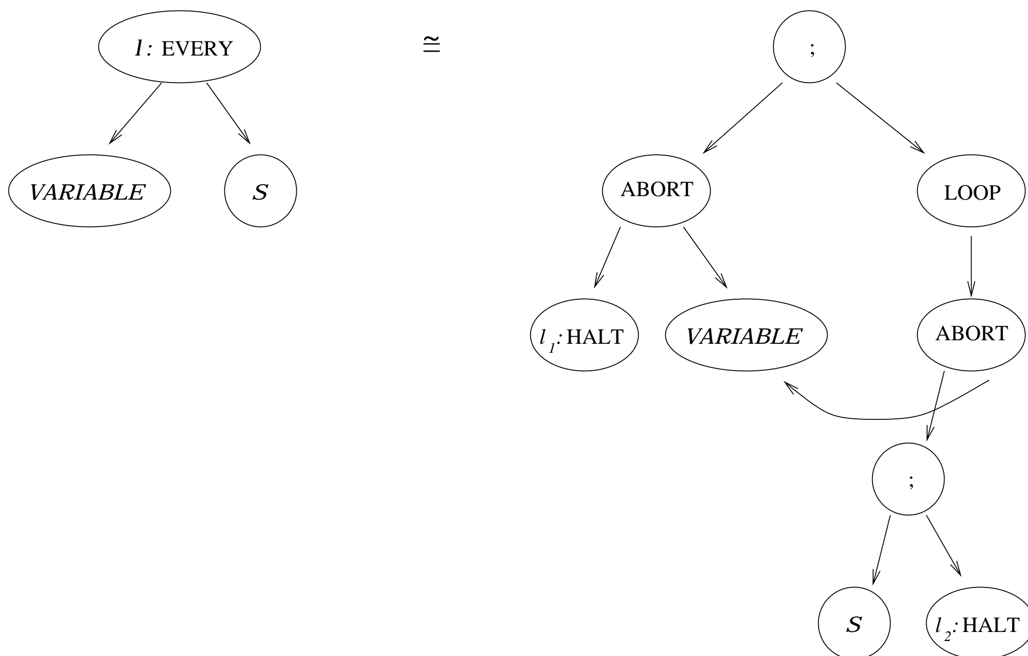

Abbildung 3.7: Redundante Ersetzung der Anweisung ***every*** x ***do*** S

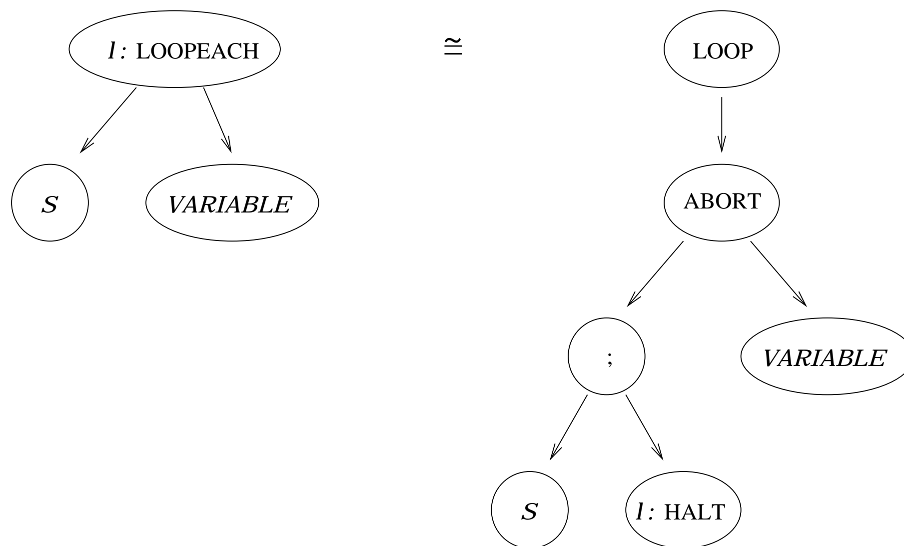

Abbildung 3.8: Redundante Ersetzung der Anweisung ***loop*** S ***each*** x

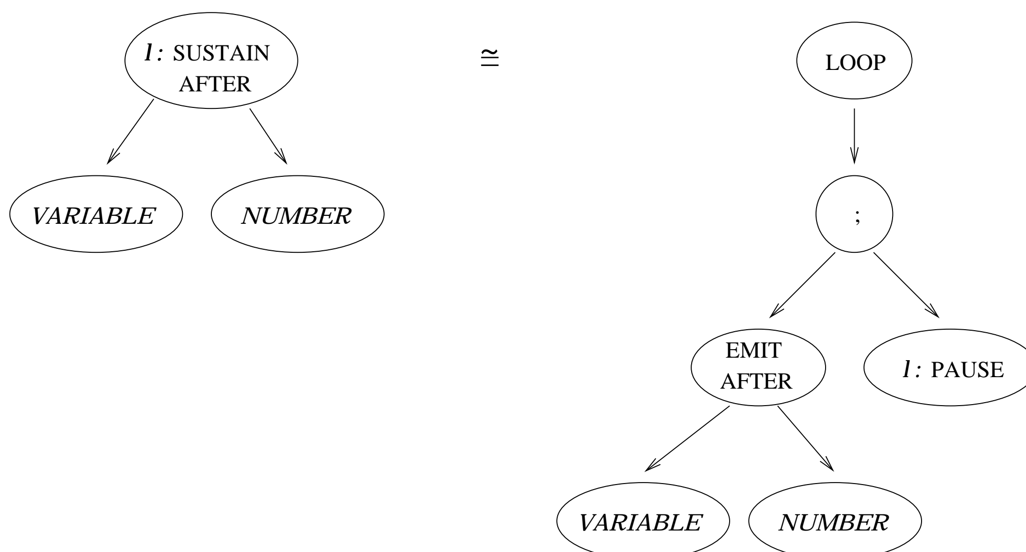

Abbildung 3.9: Redundante Ersetzung der Anweisung ***sustain*** x ***after 1***

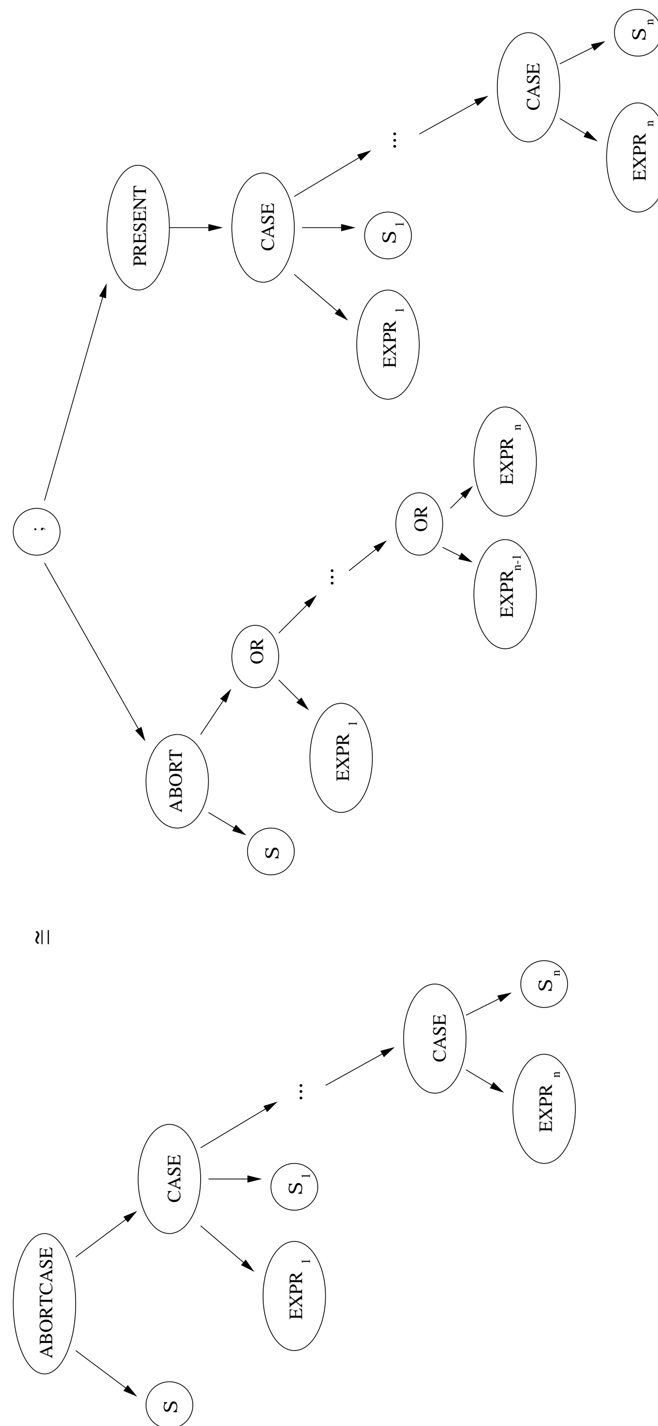

Abbildung 3.10: Redundante Ersetzung der Anweisung ***abort*** S ***when case*** $EXPR_1$ ***do*** S_1 ***case*** $EXPR_n$ ***do*** S_n

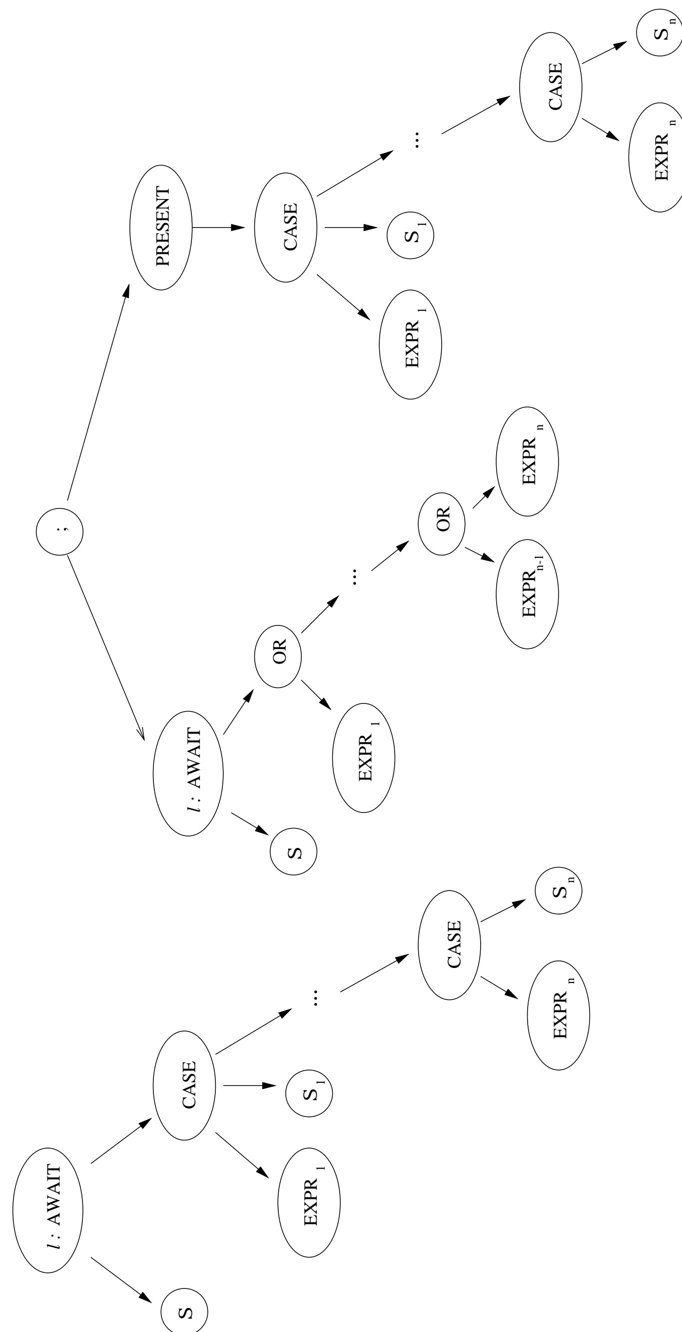

Abbildung 3.11: Redundante Ersetzung der Anweisung ***await case*** $EXPR_1$ ***do*** S_1 ***case*** $EXPR_n$ ***do*** S_n

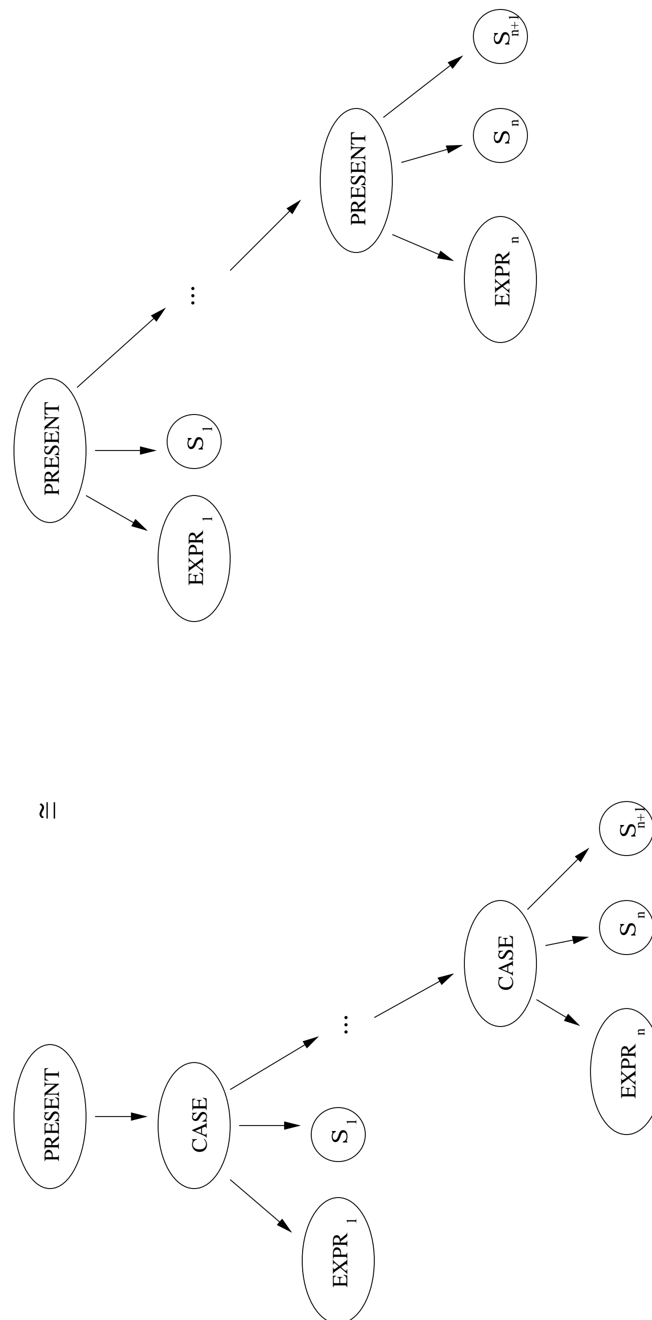

Abbildung 3.12: Redundante Ersetzung der Anweisung ***present case*** $EXPR_1$ ***do*** S_1 ***case*** $EXPR_n$ ***do*** S_n ***else do*** S_{n+1}

Diese Ersetzungen werden rekursiv im gesamten Syntaxbaum durchgeführt, bis der Syntaxbaum keine zusammengesetzten Anweisungen mehr enthält. Die folgende Abbildung verdeutlicht diese Ersetzungen anhand des Beispielprogramms P_3.

P_3: l_0 : ***pause*** ***;*** (l_1 : ***await*** a ***;*** ***emit*** x)‖ (l_2 : ***await*** b ***;*** ***emit*** y)

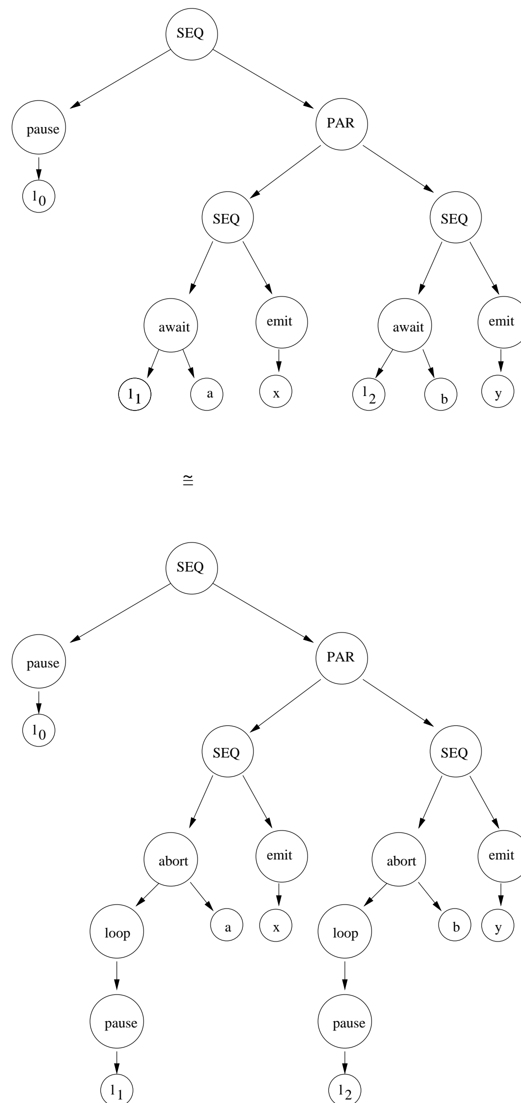

Abbildung 3.13: Syntaxbaum und redundante Ersetzungen im Programm P_3

3.3 Die Transitionsrelation eines PURR-Programms

Nachdem alle zusammengesetzten Anweisungen aus dem Syntaxbaum entfernt wurden, kann nun damit begonnen werden, die Transitionsrelation des Programms mit Hilfe des Syntaxbaumes aufzubauen. Dazu muss zuerst die Semantik der Ausdrücke und Anweisungen in PURR definiert werden.

3.3.1 Die Semantik von PURR

Die hier vorgestellte Semantik der Programmiersprache PURR setzt den Modulrumpf eines PURR-Programms direkt in eine Transitionsrelation in Form einer aussagenlogischen Formel und eine Menge von Startzuständen um. Es wird nicht erst die zu dem Programm gehörende Kripke Struktur berechnet und dann die Transitionsrelation daraus ermittelt, was dazu führt, dass die hier erzeugte Transitionsrelation eine Menge von nicht erreichbaren Zuständen enthält. Da die Anzahl der Kontrollzustände auf die Laufzeit des Verfahrens keinen Einfluss hat, wirken sich die unerreichbaren Zustände hier nicht negativ aus. Für die formale Definition der Semantik werden nur die grundlegenden Anweisungen betrachtet. Die Semantik der zusammengesetzten Anweisungen ergibt sich dann durch rekursives Berechnen aus der Semantik der grundlegenden Anweisungen. Da die ***pause***-Anweisung die einzige Anweisung ist, die Zeit verbraucht, kann der Kontrollfluss also nur an solchen Anweisungen stehen bleiben. Steht der Kontrollfluss an irgendwelchen ***pause***-Anweisungen, kann man von dort aus berechnen, an welcher ***pause***-Anweisung der Kontrollfluss nach dem nächsten Zeitschritt steht. Daher wird jede zeitverbrauchende Anweisung mit einem Label l_i versehen. Wenn man nun durch das Programm läuft, kann man zu jedem Zeitpunkt angeben, an welchen Labels der Kontrollfluss gerade steht. Als Beispiel soll das folgende Programm betrachtet werden:

module $ABRO$
input $a:0, b:0, r:0$***;***
output $o:0$***;***
loop
    abort
        $[l_a :$ ***await*** $a] \parallel [l_b :$ ***await*** $b]$;
        emit o***;***
        l_r : ***await*** r
    when r
end
end module

Bei der Abarbeitung des Programms geschieht das Folgende: Als erstes bleibt der Kontrollfluss an den beiden mit l_a und l_b markierten ***await***-Anweisungen stehen. Diese beiden Anweisungen laufen parallel und warten darauf, dass die beiden Signale a und b zu 1 ausgewertet werden können. Wenn dies geschieht, dann wird sofort, noch im selben Zeitschritt, das Signal o auf den Wert 1 gesetzt. Danach bleibt der Kontrollfluss an der mit l_r markierten ***await***-Anweisung stehen. Wenn dann irgendwann das Signal r zu 1 ausgewertet werden kann, wird die Abarbeitung neu gestartet und der Kontrollfluss steht wieder an den mit l_a und l_b markierten ***await***-Anweisungen.

Es wird also davon ausgegangen, dass jede ***pause***-Anweisung mit einem Label l_i markiert ist. Dieses Label ist eine atomare boolesche Formel. Weiterhin sei die Menge $labels(S)$, die Menge aller in S vorkommenden Labels und $inside(S) := \bigvee_{l \in labels(S)} l$ beschreibt die Menge aller Zustände, an denen sich der Kontrollfluss in S befindet.

Die Zustände der zu einem Programm gehörenden Kripke Struktur sind die möglichen Kombinationen aus den Labels sowie den Ein- und Ausgaben. Sie hängen von den Ein- und Ausgaben ab, so dass in einem bestimmten Zustand ein bestimmtes Signal in Abhängigkeit von bestimmten Eingaben den Wert 1 bekommt. Die Berechnung der Kripke Struktur geschieht in zwei Phasen:

Zuerst wird die Kripke Struktur für den Kontrollfluss und dann die für den Datenfluss des Programms berechnet. Diese beiden Strukturen werden dann zu der Kripke Struktur für das Programm zusammengesetzt.

Die Berechnung der Kripke Struktur für den Kontrollfluss geschieht in drei Phasen:
Zuerst wird der Fall betrachtet, dass sich der Kontrollfluss nicht in S befindet und zum ersten Mal in S eintritt. Danach wird der Fall betrachtet, dass sich der Kontrollfluss in S befindet, und dass der Kontrollfluss sich zum nächsten Zeitpunkt immer noch in S befindet. Der Kontrollfluss verlässt S also nicht. Hierbei wird das Eintreten in S genauso wenig betrachtet wie das Austreten aus S. Schließlich wird der Fall betrachtet, dass der Kontrollfluss S verlässt. Diese drei Phasen werden dann zusammengesetzt, um die Transitionsrelation für den Kontrollfluss eines PURR-Programms zu bekommen.

Die Berechnung der Kripke Struktur für den Datenfluss ist etwas komplizierter, da es Programme gibt, die zwar syntaktisch einwandfrei sind, jedoch keine sinnvolle Semantik besitzen. Die folgenden Anweisungen sind zwei Beispiele dafür.

P_4 : ***present*** o ***then*** ***emit*** o ***end***

P_1 : ***present*** o ***else*** ***emit*** o ***end***

P_4 hat kein deterministisches Verhalten. Es ist einerseits möglich, dass o nicht präsent ist, was dazu führt, dass o auch nicht ausgegeben wird. Auf der anderen Seite wäre möglich, dass o präsent ist, was dazu führt, dass o auch ausgegeben wird. Alle Programme müssen dem Kohärenz-Prinzip gehorchen:

Definition 13 (Kohärenzprinzip) *Ein Signal x des Typs 0 hat zu einem Zeitpunkt den Wert 1, genau dann, wenn zu diesem Zeitpunkt die Anweisung* ***emit*** *x ausgeführt wird oder zum vorherigen Zeitpunkt die Anweisung* ***emit*** *x* ***after 1*** *ausgeführt wurde.*

Das Signal x kann also zu einem Zeitpunkt nur dann den Wert 1 haben, wenn die obige Bedingung gilt. Damit hat Programm P_1 keine sinnvolle Semantik: wäre o präsent, würde o nicht ausgegeben, so dass das Kohärenzprinzip verletzt wäre. Wäre andererseits o nicht präsent, würde o ausgegeben, was wieder dem Kohärenzprinzip widerspricht.

Ähnliche Probleme können sich auch bei Zuweisungen ergeben. Beispiele hierfür sind die Zuweisungen x **:=** $\neg x$ und x **:=** $x + 1$. Im ersten Fall wird der Wert von x sofort auf $\neg x$ geändert. Es müsste also in diesem Zeitschritt $x = \neg x$ gelten, was jedoch nicht erfüllbar ist. Eine solche Zuweisung macht also keinen Sinn. Im zweiten Fall wird der Wert von x sofort auf $x + 1$ geändert. Es müsste also in diesem Zeitschritt $x = x + 1$ gelten, was durch keine Zahl erfüllbar ist. Eine solche Zuweisung macht also ebenfalls keinen Sinn. Bei der Verwendung von x**:=** $\neg x$ ***after 1*** bzw x **:=** $x + 1$ ***after 1*** gibt es diese Probleme nicht, da hier die Zuweisungen erst im nächsten Zeitschritt durchgeführt werden.

3.3.2 Die Semantik der Ausdrücke in PURR

Als erstes wird nun die Semantik der Ausdrücke definiert. Jedem Ausdruck $\tau \in \text{PTerm}_\Sigma$ mit $\text{typ}_\Sigma(\tau) = 0$ wird eine aussagenlogische Formel zugeordnet, während jedem Ausdruck $\tau \in \text{PTerm}_\Sigma$ mit $\text{typ}_\Sigma(\tau) = n > 0$ eine Liste mit aussagenlogischen Formeln der Länge n zugeordnet wird. Sei $\mathcal{B}^*$ die Vereinigung der Menge aller aussagenlogischen Formeln mit der Menge aller n-Tupel von aussagenlogischen Formeln. Um eine Operation f auf Bitvektoren auswerten zu können, muss es zu jedem $f \in C_\Sigma$ mit $s_\Sigma(f) = (n_1, \cdots, n_k)$ eine Funktion $\xi(f)$ geben, die jedem Tupel $(\varphi_1, \cdots, \varphi_k) \in \mathcal{B}^*$ ein Element aus $\mathcal{B}^*$ zuordnet. Es gilt weiterhin, dass, wenn $n_i > 0$, dann ist φ_i eine Liste aussagenlogischer Formeln der Länge n. Wenn $n_i = 0$, dann ist φ_i eine aussagenlogische Formel.

Der Operator $<_3$ bildet z.B. boolesche Felder der Länge 3 auf eine aussagenlogische Formel ab. Es gilt also $s_\Sigma(<_3) = (3, 3, 0)$. $\xi(<_3)$ ist damit eine Funktion, die zwei booleschen Feldern der Länge 3 ein Element aus $\{true, false\}$ zuordnet.

Definition 14 *Seien eine PURR-Signatur $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ und eine Funktion ξ zur Interpretation von C_Σ gegeben. Die Auswertung der Ausdrücke ist wie folgt definiert:*

- $[x]^\xi := x$ *für alle Variablen* $x \in PTerm_\Sigma$ *mit* $typ_\Sigma(x) = 0$
- $[x]^\xi := [x_{n-1}, \cdots, x_0]$ *für alle* $x \in PTerm_\Sigma$ *mit* $typ_\Sigma(x) = n > 0$
- $[\tau(i)]^\xi := b_i$, *wenn* $[\tau]^\xi := [b_{n-1}, \cdots, b_0]$
- $[f(\tau_1, \cdots, \tau_n)]^\xi := \xi(f)([\tau_1]^\xi, \cdots, [\tau_n]^\xi)$
- $[\textbf{\textit{not}}\ \tau]^\xi := \neg [\tau]^\xi$
- $[\tau_1\ \textbf{\textit{and}}\ \tau_2]^\xi := [\tau_1]^\xi \wedge [\tau_2]^\xi$
- $[\tau_1\ \textbf{\textit{or}}\ \tau_2]^\xi := [\tau_1]^\xi \vee [\tau_2]^\xi$

Für $\tau \in PTerm_\Sigma$ mit $typ_\Sigma(\tau) = 0$ ist $[\neg\tau]^\xi$ eine aussagenlogische Formel und für $\tau \in PTerm_\Sigma$ mit $typ_\Sigma(\tau) = n > 0$ ist $[\neg\tau]^\xi$ eine Liste mit aussagenlogischen Formeln der Länge n.

Die in der obigen Definition angegebenen Auswertungen werden symbolisch durchgeführt. Hierbei werden die Auswertungen nicht für konkrete Werte oder konkrete Bitvektoren, sondern für aussagenlogische Formeln auf einer symbolischen Ebene durchgeführt. Dies ermöglicht, alle möglichen Belegungen mit einer einzigen aussagenlogischen Formel zu erfassen. Durch die oben definierten Operatoren lassen sich unter anderem z.B. Operatoren wie Addition, Subtraktion, Multiplikation, Division, Modulo, $<$, $>$, $=$, $\leq$, $\geq$, Konkatenation von Bitvektoren sowie Shiftleft und Shiftright ausdrücken. In dieser Arbeit wurde $C_\Sigma = \{<, >, =, +, -, *, \mathit{Shiftleft}, \mathit{Konkatenation}\}$ gewählt.

3.3.3 Die Berechnung des Kontrollflusses

Der Kontrollfluss einer Anweisung in PURR kann durch einen endlichen Automaten beschrieben werden. Die Zustände dieses Automaten sind durch die Labels $labels(S)$, die in S vorhanden sind, gegeben. Es ist jedoch nicht ohne weiteres möglich, den Automaten direkt aus der Anweisungsfolge des PURR-Programmes zu erzeugen, da es Anweisungen gibt, die zu unterschiedlichen Zeitpunkten ausgeführt werden und somit nicht direkt dem momentanen Zeitpunkt zugeordnet werden können. Die Anweisung ***present*** i ***then pause; emit*** y ***else emit*** x ***end*** kann sich auf den aktuellen Zeitpunkt auswirken, wenn i nicht präsent ist, oder aber auf den nächsten Zeitpunkt, wenn i präsent ist. Diese Anweisung kann also nicht einfach dem momentanen Zeitpunkt zugeordnet werden, da sich die Ausführung des ***then***- und des ***else***-Zweiges der Anweisung auf unterschiedliche Zeitpunkte beziehen.

Der Kontrollfluss kann jedoch einfach berechnet werden, wenn die Berechnung in mehreren Teilen durchgeführt wird. Hierzu werden für jede Anweisung S die Transitionen *enter(S)*, *move(S)*, *terminate(S)* und *instant(S)* berechnet. Diese Transitionen beschreiben dann alle Möglichkeiten, wie der Kontrollfluss in S eintreten, wie er in S fortfahren und wie er S verlassen kann. Die Transition *instant(S)* beschreibt, ob die Anweisung S sofort terminiert ohne eine Zeitinheit zu verbrauchen. Der Kontrollfluss der Anweisung S ergibt sich dann aus der Vereinigung der berechneten Transitionen.

Definition 15 (sofort terminierende Anweisungen) *Seien eine PURR-Signatur* $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ *und eine Anweisung* $S \in PStm_\Sigma$ *gegeben. Die Formel instant(S) beschreibt dann alle Möglichkeiten, in denen* S *sofort terminiert ohne Zeit zu verbrauchen.*

- *instant(**nothing**) := 1*
- *instant(l : **pause**) := 0*
- *instant(**emit** x) := instant(**emit** x **after 1**) := 1*
- *instant(x := τ) := instant(x := τ **after 1**) := 1*
- *instant(**present** x **then** S_1 **else** S_2 **end**) :=* $\left(\begin{array}{l}(x \wedge instant(S_1)) \vee \\ (\neg x \wedge instant(S_2))\end{array}\right)$
- *instant(S_1 **;** S_2) := instant(S_1) ∧ instant(S_2)*
- *instant(S_1 ‖ S_2) := instant(S_1) ∨ instant(S_2)*
- *instant(**while** σ **do** S **end**) := ¬σ*
- *instant(**suspend** S **when** σ **end**) := instant(S)*
- *instant(**weak suspend** S **when** σ **end**) := instant(S)*
- *instant(**abort** S **when** σ **end**) := instant(S)*
- *instant(**weak abort** S **when** σ **end**) := instant(S)*
- *instant(**signal** x:α **in** S **end**) := instant(S)*

Ob eine Anweisung sofort terminiert oder nicht, hängt in der Regel von den Eingaben ab. Es gilt z.B. *instant*(***present*** *i* ***then pause ; emit*** *y* ***else emit*** *x* ***end***) = $i \wedge 0 \vee \neg i \wedge 1 = \neg i$. Die Anweisung terminiert also sofort, wenn i nicht präsent ist, ansonsten verbraucht sie eine Zeiteinheit.

Mit Hilfe der Definition für *instant*(S) kann nun *enter*(S) definiert werden. *Enter*(S) beschreibt alle Transitionen, mit denen der Kontrollfluss in S eintreten kann.

Definition 16 (Start von Anweisungen) *Seien eine PURR-Signatur $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ und eine Anweisung $S \in PStm_\Sigma$ gegeben. Die Formel enter(S) beschreibt dann alle Kombinationen aus Programmstellen und Ein- sowie Ausgaben, bei denen der Kontrollfluss stehen bleiben kann, wenn er zum ersten Mal in S eintritt. Enter(s) ist eine aussagenlogische Formel über $\Sigma_I \cup \Sigma_O \cup \{Xl | l \in labels(S)\}$*

- *enter(**nothing**) := 0*
- *enter(l : **pause**) := Xl*
- *enter(**emit** x) := enter(**emit** x **after 1**) := 0*
- *enter(x := τ) := enter(x := τ **after 1**) := 0*
- *enter(**present** x **then** S_1 **else** S_2 **end**) :=* $\left(\begin{array}{l}(x \wedge enter(S_1) \wedge \neg Xinside(S_2)) \vee \\ (\neg x \wedge enter(S_2) \wedge \neg Xinside(S_1))\end{array}\right)$
- *enter(S_1 **;** S_2) :=* $\left(\begin{array}{l}(\neg instant(S_1) \wedge enter(S_1) \wedge \neg Xinside(S_2)) \vee \\ (instant(S_1) \wedge enter(S_2) \wedge \neg Xinside(S_1))\end{array}\right)$
- *enter($S_1 \parallel S_2$) :=* $\left(\begin{array}{l}(instant(S_2) \wedge enter(S_1) \wedge \neg Xinside(S_2)) \vee \\ (instant(S_1) \wedge enter(S_2) \wedge \neg Xinside(S_1)) \vee \\ (enter(S_1) \wedge enter(S_2))\end{array}\right)$
- *enter(**while** σ **do** S **end**) := $\sigma \wedge$ enter(S)*
- *enter(**suspend** S **when** σ **end**) := enter(S)*
- *enter(**weak suspend** S **when** σ **end**) := enter(S)*
- *enter(**abort** S **when** σ **end**) := enter(S)*
- *enter(**weak abort** S **when** σ **end**) := enter(S)*
- *enter(**signal** x:α **in** S **end**) := enter(S)*

Sollte *instant*(S) für eine Anweisung S und eine Eingabe wahr sein, dann ist *enter*(S) für diese Anweisung und diese Eingabe falsch: Da die Anweisung S sofort terminiert, kann der Kontrollfluss nicht in S stehen bleiben. Daher werden alle Anweisungen, die keine Zeit verbrauchen, zu 0 ausgewertet. Bei *enter*(S_1 **;** S_2) muss zwischen zwei Möglichkeiten unterschieden werden. Zum einen kann S_1 sofort terminieren, so dass der Kontrollfluss im gleichen Moment S_1 passiert und in S_2 eintritt. Zum anderen kann die Abarbeitung von S_1 jedoch auch Zeit verbrauchen, so dass der Kontrollfluss nur in S_1, nicht aber in S_2 eintritt. Im Falle von *enter*($S_1 \parallel S_2$) gibt

es sogar drei mögliche Fälle, zwischen denen unterschieden werden muss. Im ersten Fall tritt der Kontrollfluss gleichzeitig in S_1 und S_2 ein, wobei S_1 sofort wieder terminiert. Im zweiten Fall tritt der Kontrollfluss gleichzeitig in S_1 und S_2 ein, wobei S_2 sofort wieder terminiert. Im dritten Fall terminieren weder S_1 noch S_2 sofort und der Kontrollfluss bleibt sowohl in S_1 als auch in S_2 stehen.

Als Nächstes kann *terminate*(S) definiert werden. *Terminate*(S) beschreibt alle Möglichkeiten, unter denen der Kontrollfluss die Anweisung S verlässt.

Definition 17 (Terminierung von Anweisungen) *Seien eine PURR-Signatur* $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ *und eine Anweisung* $S \in \mathit{PStm}_\Sigma$ *gegeben. Die Formel terminate(S) beschreibt dann alle Möglichkeiten in denen der Kontrollfluss aus* S *austritt und sich im nächsten Zeitpunkt nicht mehr in* S *befindet. Terminate(s) ist eine aussagenlogische Formel über* $\Sigma_I \cup \Sigma_O \cup \mathit{labels}(S)$

- *terminate(**nothing**) := 0*
- *terminate(l : **pause**) :=* l
- *terminate(**emit** x) := terminate(**emit** x **after 1**) := 0*
- *terminate(x :=* τ*) := terminate(x :=* τ ***after 1**) := 0*
- *terminate(**present** x **then** S_1 **else** S_2 **end**) :=*
  $\left(\begin{array}{l} (\mathit{inside}(S_1) \wedge \mathit{terminate}(S_1)) \vee \\ (\mathit{inside}(S_2) \wedge \mathit{terminate}(S_2)) \end{array} \right)$
- *terminate(S_1 **;** S_2) :=*
  $\left(\begin{array}{l} (\mathit{inside}(S_1) \wedge \mathit{terminate}(S_1)) \wedge \mathit{instant}(S_2) \vee \\ (\mathit{inside}(S_2) \wedge \mathit{terminate}(S_2)) \end{array} \right)$
- *terminate($S_1 \parallel S_2$) :=*
  $\left(\begin{array}{l} (\mathit{inside}(S_1) \wedge \neg\mathit{inside}(S_2) \wedge \mathit{terminate}(S_1)) \vee \\ (\mathit{inside}(S_2) \wedge \neg\mathit{inside}(S_1) \wedge \mathit{terminate}(S_2)) \vee \\ (\mathit{inside}(S_1) \wedge \mathit{inside}(S_2) \wedge \mathit{terminate}(S_1) \wedge \mathit{terminate}(S_2)) \end{array} \right)$
- *terminate(**while** σ **do** S **end**) :=* $\neg\sigma \wedge$ *terminate(S)*
- *terminate(**suspend** S **when** σ **end**) :=*
  $\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{terminate}(S)$
- *terminate(**weak suspend** S **when** σ **end**) :=*
  $\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{terminate}(S)$

- *terminate(**abort** S **when** σ **end**) :=*
$$\left(\begin{array}{l}(\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{terminate}(S)) \vee \\ (\mathit{inside}(S) \wedge \sigma)\end{array}\right)$$

- *terminate(**weak abort** S **when** σ **end**) :=*
$$\left(\begin{array}{l}(\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{terminate}(S)) \vee \\ (\mathit{inside}(S) \wedge \sigma)\end{array}\right)$$

- *terminate(**signal** $x{:}\alpha$ **in** S **end**) := terminate(S)*

Terminate(S) ist eine aussagenlogische Formel über V_Σ. *Terminate*(S) beschreibt nur die Möglichkeiten, unter denen der Kontrollfluss aus S austritt. *Terminate*(S) sagt jedoch nichts über den Zustand aus, in dem der Kontrollfluss als Nächstes stehen bleibt. Dies ist auch gar nicht möglich, wenn man S losgelöst vom Kontext betrachtet. Es könnte eine umgebende Anweisung geben, der S als einen Unterbefehl enthält, z.B. **loop** S **end**. In diesem Fall tritt der Kontrollfluss nach der Terminierung sofort wieder in S ein. Im Falle einer Sequenz gibt es z.B. eine Transition von S_1 nach S_2, bei der der Kontrollfluss zuerst S_1 verlässt und danach in S_2 eintritt.

Diese internen Transitionen einer Anweisung S werden nun definiert. *Move*(S) beschreibt alle Möglichkeiten, unter denen der Kontrollfluss eine Transition innerhalb von S ausführt.

Definition 18 (Kontrollfluss führt eine Transition innerhalb von S aus) *Seien eine PURR-Signatur $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ und eine Anweisung $S \in PStm_\Sigma$ gegeben. Die Formel move(S) beschreibt dann alle Möglichkeiten, in denen der Kontrollfluss einen Schritt innerhalb von S durchführt. Move(s) ist eine aussagenlogische Formel über $\Sigma_I \cup \Sigma_O \cup \Sigma_L \cup \{Xl | l \in labels(S)\}$*

- *move(**nothing**) := 0*

- *move(l : **pause**) := 0*

- *move(**emit** x) := move(**emit** x **after 1**) := 0*

- *move(x := τ) := move(x := τ **after 1**) := 0*

- *move(**present** x **then** S_1 **else** S_2 **end**) :=*
$$\left(\begin{array}{l}(\mathit{inside}(S_1) \wedge \mathit{move}(S_1) \wedge \neg X\mathit{inside}(S_2)) \vee \\ (\mathit{inside}(S_2) \wedge \mathit{move}(S_2) \wedge \neg X\mathit{inside}(S_1))\end{array}\right)$$

- *move(S_1 **;** S_2) :=*
$$\left(\begin{array}{l}(\mathit{inside}(S_1) \wedge \neg\mathit{terminate}(S_1) \wedge \mathit{move}(S_1) \wedge \neg X\mathit{inside}(S_2)) \vee \\ (\mathit{inside}(S_2) \wedge \mathit{move}(S_2) \wedge \neg X\mathit{inside}(S_1)) \vee \\ (\mathit{inside}(S_1) \wedge \mathit{terminate}(S_1) \wedge \mathit{enter}(S_2) \wedge \neg X\mathit{inside}(S_1))\end{array}\right)$$

- *move(*$S_1 \parallel S_2$*) :=*
$$\left(\begin{array}{l} (\mathit{inside}(S_1) \wedge \neg \mathit{inside}(S_2) \wedge \mathit{move}(S_1)) \vee \\ (\mathit{inside}(S_2) \wedge \neg \mathit{inside}(S_1) \wedge \mathit{move}(S_2)) \vee \\ (\mathit{inside}(S_1) \wedge \mathit{inside}(S_2) \wedge \mathit{move}(S_1) \wedge \mathit{move}(S_2)) \end{array}\right)$$

- *move(**while** σ **do** S **end**) :=*
$$\left(\begin{array}{l} (\mathit{inside}(S) \wedge \neg \mathit{terminate}(S) \wedge \mathit{move}(S)) \vee \\ (\mathit{inside}(S) \wedge \mathit{terminate}(S) \wedge \sigma \wedge \mathit{enter}(S)) \end{array}\right)$$

- *move(**suspend** S **when** σ **end**) :=*
$$\left(\begin{array}{l} (\mathit{inside}(S) \wedge \sigma \wedge \left(\bigwedge_{l \in \mathit{labels(S)}} (l = Xl)\right)) \vee \\ (\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{move}(S)) \end{array}\right)$$

- *move(**weak suspend** S **when** σ **end**) :=*
$$\left(\begin{array}{l} (\mathit{inside}(S) \wedge \sigma \wedge \left(\bigwedge_{l \in \mathit{labels(S)}} (l = Xl)\right)) \vee \\ (\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{move}(S)) \end{array}\right)$$

- *move(**abort** S **when** σ **end**) :=*
$$\left((\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{move}(S)) \right)$$

- *move(**abort** S **when** σ **end**) :=*
$$\left((\mathit{inside}(S) \wedge \neg\sigma \wedge \mathit{move}(S)) \right)$$

- *move(**signal** $x{:}\alpha$ **in** S **end**) := move(*S*)*

Bei der Definition von *move*(S) ist es notwendig, dass sich der Kontrollfluss zum momentanen Zeitpunkt in S befindet, und dass er sich auch zum nächsten Zeitpunkt in S befindet. Im Falle von ***present*** und der Sequenz ist zu bemerken, dass sich der Kontrollfluss nicht gleichzeitig in S_1 und S_2 aufhalten kann. Dies kann zu Optimierungen verwendet werden.

Für die zusammengesetzten Anweisungen erhält man nun:

- l : ***halt***:
  - *instant*(l : ***halt***) := 0
  - *enter*(l : ***halt***) := Xl
  - *terminate*(l : ***halt***) := 0
  - *move*(l : ***halt***) := l ∧ Xl

- l : ***await*** σ:
  - *instant*(l : ***await*** σ) := 0
  - *enter*(l : ***await*** σ) := Xl
  - *terminate*(l : ***await*** σ) := $l \wedge \sigma$
  - *move*(l : ***await*** σ) := $l \wedge$ X$l \wedge \neg\sigma$
- l : ***sustain*** x:
  - *instant*(l : ***sustain*** x) := 0
  - *enter*(l : ***sustain*** x) := Xl
  - *terminate*(l : ***sustain*** x) := 0
  - *move*(l : ***sustain*** x) := $l \wedge$ Xl
- l : ***sustain*** x(τ):
  - *instant*(l : ***sustain*** x(τ)) := 0
  - *enter*(l : ***sustain*** x(τ)) := Xl
  - *terminate*(l : ***sustain*** x(τ)) := 0
  - *move*(l : ***sustain*** x(τ)) := $l \wedge$ Xl
- ***loop*** S ***end***:
  - *instant*(***loop*** S ***end***) := *instant*(S)
  - *enter*(***loop*** S ***end***) := *enter*(S)
  - *terminate*(***loop*** S ***end***) := 0
  - *move*(***loop*** S ***end***) := $\left(\begin{array}{l} \mathit{inside}(\mathrm{S}) \wedge \mathit{terminate}(S) \wedge \mathit{enter}(S) \vee \\ \mathit{inside}(S) \wedge \neg\mathit{terminate}(S) \wedge \mathit{move}(S) \end{array}\right)$
- ***repeat*** S ***until*** σ:
  - *instant*(***repeat*** S ***until*** σ) := *instant*(S)
  - *enter*(***repeat*** S ***until*** σ) := *enter*(S)
  - *terminate*(***repeat*** S ***until*** σ) := *inside*(S) $\wedge$ *terminate*(S) $\wedge\ \sigma$
  - *move*(***repeat*** S ***until*** σ) := $\left(\begin{array}{l} \mathit{inside}(S) \wedge \mathit{terminate}(S) \wedge \neg\sigma \wedge \mathit{enter}(S) \vee \\ \mathit{inside}(S) \wedge \neg\mathit{terminate}(S) \wedge \mathit{move}(S) \end{array}\right)$

- l : ***loop*** S ***each*** σ ***end***:
  - *instant*(l : ***loop*** S ***each*** σ ***end***) := 0
  - *enter*(l : ***loop*** S ***each*** σ ***end***) := *enter*(S)
  - *terminate*(l : ***loop*** S ***each*** σ ***end***) := 0
  - *move*(l : ***loop*** S ***each*** σ ***end***) :=
    $$\left(\begin{array}{l} inside(S) \wedge terminate(S) \wedge \mathrm{X}l \vee \\ \neg inside(S) \wedge l \wedge \sigma \wedge enter(S) \vee \\ \neg inside(S) \wedge l \wedge \neg\sigma \wedge \neg X inside(S) \wedge \mathrm{X}l \end{array}\right)$$
- ***suspend*** S ***when immediate*** σ:
  - *instant*(***suspend*** S ***when immediate*** σ) := *instant*(S) $\vee\sigma$
  - *enter*(***suspend*** S ***when immediate*** σ) := *enter*(S) $\wedge\neg\sigma$
  - *terminate*(***suspend*** S ***when immediate*** σ) :=
    terminate(***suspend*** S ***when*** σ)
  - *move*(***suspend*** S ***when immediate*** σ) :=
    move(***suspend*** S ***when*** σ)
- ***weak suspend*** S ***when immediate*** σ:
  - *instant*(***weak suspend*** S ***when immediate*** σ) := *instant*(S) $\vee\sigma$
  - *enter*(***weak suspend*** S ***when immediate*** σ) := *enter*(S) $\wedge\neg\sigma$
  - *terminate*(***weak suspend*** S ***when immediate*** σ) :=
    terminate(***weak suspend*** S ***when*** σ)
  - *move*(***weak suspend*** S ***when immediate*** σ) :=
    move(***weak suspend*** S ***when*** σ)
- ***abort*** S ***when immediate*** σ:
  - *instant*(***abort*** S ***when immediate*** σ) := *instant*(S) $\vee\sigma$
  - *enter*(***abort*** S ***when immediate*** σ) := *enter*(S) $\wedge\neg\sigma$
  - *terminate*(***abort*** S ***when immediate*** σ) :=
    terminate(***abort*** S ***when*** σ)
  - *move*(***abort*** S ***when immediate*** σ) :=
    move(***abort*** S ***when*** σ)

- ***weak abort*** S ***when immediate*** σ:
  - *instant*(***weak abort*** S ***when immediate*** σ) := *instant*(S) $\vee\sigma$
  - *enter*(***weak abort*** S ***when immediate*** σ) := *enter*(S) $\wedge\neg\sigma$
  - *terminate*(***weak abort*** S ***when immediate*** σ) :=
    terminate(***weak abort*** S ***when*** σ)
  - *move*(***weak abort*** S ***when immediate*** σ) :=
    move(***weak abort*** S ***when*** σ)

Für die restlichen zusammengesetzten Anweisungen werden *instant*(S), *enter*(S), *move*(S) und *terminate*(S) an dieser Stelle nicht berechnet. Die Formeln für *instant*(S), *enter*(S), *move*(S) und *terminate*(S) ergeben sich durch rekursives Einsetzen der einzelnen Anweisungen in den redundanten Ersetzungen der zusammengesetzten Anweisungen.

Die Semantik der ***immediate***-Varianten der ***suspend***- und ***abort***-Anweisungen unterscheiden sich nur bei *enter*(S). Der Kontrollfluss von ***suspend*** und ***weak suspend*** ist gleich, genauso wie der von ***abort*** und ***weak abort***. Diese Anweisungen unterscheiden sich nur im Datenfluss. Die aussagenlogischen Formeln *enter*(S), *move*(S) und *terminate*(S) definieren den Kontrollfluss einer Anweisung S. Die Ein- und Ausgaben werden im Moment nicht betrachtet. Die Kombination von *enter*(S), *move*(S) und *terminate*(S) ergibt nun den Kontrollfluss der Anweisung S.

Definition 19 (Kontrollfluss einer Anweisung) *Seien eine PURR-Signatur $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ und eine Anweisung $S \in PStm_\Sigma$ sowie $l_{boot} \in \Sigma_L \setminus labels(S)$ gegeben. Der Kontrollfluss von S ist definiert als Kripke Struktur $(I, \wp(V_\Sigma), R, L))$ über V_Σ, wobei die Initialzustände I und die Transitionsrelation R durch ihre charakteristischen Funktionen definiert sind und L die Markierungsfunktion ist, die einem Zustand $s \in \wp(V_\Sigma)$ die Markierung $L(s) := s$ zuordnet.*

$$\phi_I^K := l_{boot} \wedge \neg inside(S)$$

$$\phi_R^K := \begin{pmatrix} (\neg inside(S) \wedge l_{boot} \wedge enter(S)) \vee \\ (inside(S) \wedge \neg terminate(S) \wedge move(S) \wedge \neg l_{boot}) \vee \\ (inside(S) \wedge terminate(S) \wedge \neg X inside(S) \wedge \neg l_{boot}) \vee \\ (\neg inside(S) \wedge \neg X inside(S) \wedge \neg l_{boot}) \end{pmatrix} \wedge \neg X l_{boot}$$

ϕ_I definiert die Startzustände, in denen ein zusätzliches Label l_{boot} eingefügt wird. Mit Hilfe dieses Labels wird zwischen dem Startzustand und den Zuständen in denen das Programm terminiert, unterschieden: l_{boot} gilt genau in den Startzuständen. ϕ_R definiert die Transitionsrelation des Programms. ϕ_R besteht aus vier Teilen:

Im ersten Teil wird beschrieben wie der Kontrollfluss in S eintritt. Im zweiten Teil werden alle Transitionen beschrieben, die der Kontrollfluss innerhalb von S ausführen kann. Im dritten Teil wird beschrieben wie der Kontrollfluss aus S austritt und im vierten Teil wird sichergestellt, dass nach der Terminierung keine weiteren Anweisungen mehr ausgeführt werden.

Als Beispiel soll nun das folgende Programm P_5 betrachtet werden. Das Programm gibt im ersten Schritt die beiden Signale x und y aus. Im zweiten Schritt werden keine Signale ausgegeben und das Programm terminiert.

P_5: l_0 : ***pause*** (**emit** x ***;*** l_1 : ***pause***)$\|$ (***emit*** y ***;*** l_2 : ***pause***)

Für das Programm P_5 ergeben sich die folgenden Formeln für *enter*(S), *move*(S), *terminate*(S) sowie *inside*(S):

$$
\begin{aligned}
&\textit{inside}(S) := l_0 \lor l_1 \lor l_2\\
&\textit{enter}(S) := Xl_0 \land \neg Xl_1 \land \neg Xl_2\\
&\textit{move}(S) := l_0 \land \neg Xl_0 \land Xl_1 \land Xl_2\\
&\textit{terminate}(S) := l_1 \lor l_2
\end{aligned}
$$

Mit diesen Formeln ergeben sich für ϕ_I und ϕ_R:

$$
\phi_I := l_{boot} \land \neg l_0 \land \neg l_1 \land \neg l_2
$$

$$
\phi_R := \left(\begin{array}{l}
(l_{boot} \land \neg l_0 \land \neg l_1 \land \neg l_2 \land Xl_0 \land \neg Xl_1 \land \neg Xl_2 \land \neg Xl_{boot}) \lor\\
(l_0 \land \neg l_1 \land \neg l_2 \land \neg Xl_0 \land Xl_1 \land Xl_2 \land \neg l_{boot} \land \neg Xl_{boot}) \lor\\
((l_1 \lor l_2) \land \neg Xl_1 \land \neg Xl_2 \land \neg Xl_0 \land \neg l_{boot} \land \neg Xl_{boot}) \lor\\
(\neg l_0 \land \neg l_1 \land \neg l_2 \land \neg Xl_1 \land \neg Xl_2 \land \neg Xl_0 \land \neg l_{boot} \land \neg Xl_{boot})
\end{array}\right)
$$

Im Normalfall sind die Formeln, die man durch die Verwendung von *instant*(S), *enter*(S), *move*(S), *terminate*(S) sowie *inside*(S) erhält, sehr umfangreich. Diese Formeln können jedoch noch vereinfacht werden. Es ist daher ratsam, eine Minimierungsfunktion oder eine Programmbibliothek für OBDD's zu verwenden. Die folgende Abbildung zeigt das Zustandsübergangsdiagramm für das Programm P_3. In diesem Diagramm sind jedoch nur die tatsächlich erreichbaren Zustände dargestellt. Der Startzustand ist durch eine zusätzliche Linie und der Terminalzustand durch ein Quadrat gekennzeichnet. Die erzeugten Formeln für ϕ_I und ϕ_R enthalten jedoch auch alle unerreichbaren Zustände.

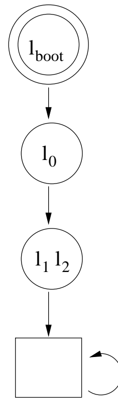

Abbildung 3.14: Kontrollfluss des Programms P_5

3.3.4 Die Berechnung des Datenflusses

Normalerweise enthält der Kontrollfluss auch unerreichbare Zustände, die dadurch zustande kommen, dass bei der Berechnung des Kontrollflusses die Ein- und Ausgaben des Programms ausser Acht gelassen wurden. Daher enthält der Kontrollfluss auch einige unmögliche Ein- und Ausgabekombinationen. Um diese unmöglichen Ein-/Ausgabekombinationen zu entfernen, muss nun der tatsächliche Datenfluss einer Anweisung S berechnet werden. Der Datenfluss wird ohne Berücksichtigung des Kontrollflusses berechnet. Die Kombination aus Kontrollfluss und Datenfluss ergibt dann das tatsächliche Verhalten einer Anweisung S. Die Berechnung des Datenflusses geschieht mit Hilfe einer Liste von guarded commands, die zu einer Anweisung S erzeugt wird. Eine formale Definition dieser guarded commands wird im Folgenden gegeben.

Definition 20 (Guarded Commands) *Seien eine PURR-Signatur Σ, eine aussagenlogische Formel γ aus $V_\Sigma \cup \{Xl | l \in \Sigma_L\}$ und Signale x und y mit typ$_\Sigma(x) = 0$ und typ$_\Sigma(y) > 0$ gegeben. Guarded commands sind dann folgendermaßen definiert:*

- $(\gamma, y := \tau)$
- $(\gamma, y := \tau$ ***after 1***$)$
- $(\gamma,$ ***emit*** $x)$
- $(\gamma,$ ***emit*** x ***after 1***$)$

Ein guarded command (γ, C) wird ausgeführt, wenn der guard γ durch einen Zustand und eine Ein-/Ausgabekombination erfüllt wird. Der guard ist eine aussagenlogische Formel über $V_\Sigma \cup \{Xl | l \in \Sigma_L\}$, der eine Menge von Transitionen in der Anweisung S beschreibt. Ein guarded command kann jedoch auch inkonsistent sein. Zum Beispiel macht das guarded command $(\neg x,$ ***emit*** $x)$ keinen Sinn, da genau dann, wenn x nicht präsent ist, x ausgegeben wird. Wenn x jedoch dann ausgegeben wird, dann ist der guard nicht mehr erfüllt. Dies ist eine Inkonsistenz zwischen dem guard γ und der Anweisung C. Es gibt jedoch auch Inkonsistenzen, die nur in der Anweisung C eines guarded command auftreten. Dies ist z.B. bei (γ, x **:=** $x + 1$) der Fall.

Die Berechnung der Liste der guarded commands einer Anweisung S wird im Folgenden definiert.

Definition 21 *Seien eine PURR-Signatur Σ, eine aussagenlogische Formel φ aus V_Σ und eine Anweisung S aus PStm$_\Sigma$ gegeben. Die Liste der guarded commands guardcmd(φ, S) einer Anweisung S ist dann wie folgt definiert:*

- *guardcmd(φ, **nothing**) := {}*
- *guardcmd(φ, l : **pause**) := {}*
- *guardcmd(φ, **emit** x) := {(φ, **emit** x)}*
- *guardcmd(φ, **emit** x **after 1**) := {(φ, **emit** x **after 1**)}*
- *guardcmd(φ, x **:=** τ) := {(φ, x **:=** τ)}*
- *guardcmd(φ, x **:=** τ **after 1**) := {(φ, x **:=** τ **after 1**)}*
- *guardcmp(φ, **present** x **then** S_1 **else** S_2 **end**) :=*
  guardcmd($\varphi \wedge \sigma$, S_1) $\cup$ guardcmd($\varphi \wedge \neg\sigma$, S_2)
- *guardcmd(φ, S_1 **;** S_2) :=*
  $$\textit{guardcmd}(\varphi, S_1) \cup \textit{guardcmd}\left(\left(\begin{array}{l}(\textit{instant}(S_1) \wedge \varphi) \vee \\ (\neg\textit{instant}(S_1) \wedge \textit{terminate}(S_1))\end{array}\right), S_2\right)$$
- *guardcmd(φ, $S_1 \parallel S_2$) := guardcmd(φ, S_1) $\cup$ guardcmd(φ, S_2)*
- *guardcmd(φ, **loop** S **end**) := guardcmd($\varphi \vee$ terminate(S), S)*
- *guardcmd(φ, **suspend** S **when** σ **end**) :=*
  guardcmd($\varphi \wedge \neg\sigma$, S_1)
- *guardcmd(φ, **weak suspend** S **when** σ **end**) :=*
  guardcmd(φ, S_1)

- *guardcmd*(φ, ***abort*** S ***when*** σ ***end***) := *guardcmd*($\varphi \wedge \neg\sigma$, S_1)
- *guardcmd*(φ, ***weak abort*** S ***when*** σ ***end***) := *guardcmd*(φ, S_1)
- *guardcmd*(φ, ***signal*** x:α ***in*** S ***end***) := *guardcmd*(φ, S_1)
- *guardcmd*(φ, ***while*** σ ***do*** S ***end***) :=
  guardcmd$((\varphi \vee \mathit{terminate}(S)) \vee \sigma$, S)

Bei der Definition der Sequenz ist für den Teil S_2 zwischen zwei Fällen zu unterscheiden. Im ersten Fall ist es möglich, dass S_1 sofort terminiert, so dass der letzte Zustand durch φ beschrieben wurde. Im zweiten Fall ist es möglich, dass S_1 nicht sofort terminiert. In diesem Fall muss der letzte Zustand in S_1 berechnet werden, in dem sich der Kontrollfluss vor dem Verlassen von S_1 befunden hat.

In der Definition wird auch der Unterschied zwischen den starken und den schwachen Varianten der Prozessverdrängung und des Aussetzens von Prozessen deutlich. Während die starken Varianten keinerlei Manipulation der Daten mehr erlauben, wenn die Bedingung σ zu 1 ausgewertet werden kann, erlauben die schwachen Varianten auch dann noch die Manipulation von Daten.

Das Berechnen der guarded commands soll an dem Programm P_6 beispielhaft durchgeführt werden.

P_6:

l_{boot} : ***pause***
present i ***then*** l_1 : ***pause else emit*** x ***end ;***
emit y ***after 1*** ;
l_2 : ***pause***
present y ***then emit*** z ***else emit*** y ***end***

Aus diesem Programm können die folgenden guarded commands berechnet werden:

1. $(l_{boot} \wedge \neg i,$ ***emit*** $x)$
2. $((l_{boot} \wedge \neg i) \vee l_1,$ ***emit*** x ***after 1***$)$
3. $(l_2 \wedge y,$ ***emit*** $z)$
4. $(l_2 \wedge \neg y,$ ***emit*** $y)$

Das letzte guarded command widerspricht sich selbst und kann daher weggelassen werden.

Um die Struktur des Datenflusses aus der Liste der guarded commands zu berechnen, muss berücksichtigt werden, dass Signale mit unterschiedlichen Typen unterschiedlich behandelt werden müssen. Der Wert von Signalen des Typs $\text{typ}_\Sigma(x) = 0$ muss zu jedem Zeitpunkt neu berechnet werden, egal welchen Wert sie im vorhergehenden Zeitpunkt hatten. Im Gegensatz dazu behalten Signale vom Typ $\text{typ}_\Sigma(x) > 0$ ihren Wert so lange bei, bis ihr Wert geändert wird.

Die folgende Definition gibt die Semantik der guarded commands wieder.

Definition 22 *Seien eine PURR-Signatur Σ mit einer Funktion ξ, eine Anweisung $S \in$ PStm$_\Sigma$ und Signale x und y mit typ$_\Sigma(x) = 0$ und typ$_\Sigma(y) > 0$ gegeben. Seien $\{(\alpha_1, \textbf{emit}\ x), \cdots, (\alpha_k, \textbf{emit}\ x)\}$ und $\{(\beta_1, \textbf{emit}\ x\ \textbf{after 1}), \cdots, (\beta_k, \textbf{emit}\ x\ \textbf{after 1})\}$ die Mengen der sofortigen und der verzögerten Ausgaben von Signalen, die in S vorkommen. Weiterhin seien $\{(\gamma_1, y\textbf{:=}\ \tau_1), \cdots, (\gamma_k, y\textbf{:=}\ \tau_m)\}$ und $\{(\delta_1, y\textbf{:=}\ \pi_1\ \textbf{after 1}), \cdots, (\delta_k, y\textbf{:=}\ \pi_n\ \textbf{after 1})\}$ die Mengen der sofortigen und der verzögerten Zuweisungen von Signalen, die in S vorkommen. Dann definiert man für jede Variable x des Typs 0 und jede Variable y des Typs > 0*

- $\phi_I(x, S) := \left[\left(\bigvee_{i=1}^k \alpha_i\right) = x\right]$
- $\phi_R(x, S) := \left(\begin{array}{l} \left[\left(\bigvee_{i=1}^k \alpha_i\right) \to x\right] \wedge \left[\left(\bigvee_{i=1}^l \beta_i\right) \to Xx\right] \wedge \\ \left[Xx \to \left(\bigvee_{i=1}^k X\alpha_i\right) \vee \left(\bigvee_{i=1}^l \beta_i\right)\right] \end{array} \right)$
- $\phi_I(y, S) := \left(\bigwedge_{i=1}^m \left(\gamma_i \to [y = \tau_i]^\xi\right)\right)$
- $\phi_R(y, S) := \left(\begin{array}{l} \left[\bigwedge_{i=1}^m \left(\gamma_i \to [y = \tau_i]^\xi\right)\right] \wedge \left[\bigwedge_{i=1}^n (\delta_i \to [Xx = \pi_i])\right] \wedge \\ \left[\left(\bigwedge_{i=1}^m \neg\gamma_i \wedge \bigwedge_{i=1}^n \neg\delta_i\right) \to [Xy = y]^\xi\right] \end{array} \right)$

Wenn eines der α_i gilt, dann wird x ausgegeben. Ähnliches gilt für die verspätete Ausgabe von x. Wenn eines der β_i gilt, dann wird x zum nächsten Zeitpunkt ausgegeben. Es kann also in einem Zeitpunkt zwei Gründe für die Ausgabe eines Signals x geben: Zum jetzigen Zeitpunkt kann eines der α_i gelten oder zum vorigen Zeitpunkt galt eines der β_i. Im Startzustand brauchen die vorigen Zeitpunkte nicht betrachtet werden. Daher brauchen hier die β_i nicht in die Berechnung mit einbezogen werden. Wenn eines der γ_i gilt, dann muss auch $y = \tau_i$ gelten. Wenn dagegen eines der δ_i gilt, dann muss auch $Xy = \pi_i$ gelten. In einem solchen Fall wird π_i sofort ausgewertet, aber die Zuweisung wird erst zum nächsten Zeitpunkt durchgeführt. Wenn es keine Zuweisung zum momentanen Zeitpunkt gibt, dann muss sichergestellt sein, dass die Variablen ihre momentanen Werte beibehalten.

Die folgende Definition beschreibt die Berechnung des Datenflusses einer Anweisung.

Definition 23 *Seien eine PURR-Signatur $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ und eine Anweisung $S \in PStm_\Sigma$ sowie eine Funktion ξ gegeben. Der Datenfluss von S ist definiert als Kripke Struktur $(I, \wp(V_\Sigma), R, L))$ über V_Σ wobei I und R durch ihre charakteristischen Funktionen definiert sind und L die Markierungsfunktion ist, die einem Zustand $s \in \wp(V_\Sigma)$ die Markierung $L(s) := s$ zuordnet.*

$$\phi_I^D := \bigwedge_{x \in \Sigma_O \cup \Sigma_V} \phi_I(x, S) \quad \phi_R^D := \bigwedge_{x \in \Sigma_O \cup \Sigma_V} \phi_R(x, S)$$

Als Beispiel für die Berechnung des Datenflusses soll nun der Datenfluss des Programms P_5 berechnet werden.

Es ergeben sich: $\phi_I = (l_0 = x) \wedge (l_0 = y)$ und $\phi_R = (\neg l_0 \vee (x \wedge y)) \wedge (Xl_0 \vee (\neg x \wedge \neg y))$

3.3.5 Kombination von Kontroll- und Datenfluss

Nachdem der Kontroll- und der Datenfluss einer Anweisung S berechnet sind, können sie kombiniert werden, um die Kripke Struktur zu erhalten, die die Semantik der Anweisung S repräsentiert.

Definition 24 *Seien eine PURR-Signatur $\Sigma = (V_\Sigma, C_\Sigma, s_\Sigma)$ und eine Anweisung $S \in PStm_\Sigma$ sowie $l_{boot} \in \Sigma_L \setminus labels(S)$ gegeben. Weiterhin seien die Kripke Strukturen $K = (\phi_I^K, \wp(V_\Sigma), \phi_R^K, L)$ und $D = (\phi_I^D, \wp(V_\Sigma), \phi_R^D, L)$ für den Kontroll- und den Datenfluss von S gegeben. Die Semantik von S ist dann definiert durch $\phi_I^K \times \phi_I^D$ und $\phi_R^K \times \phi_R^D$.*

Die Kombination aus Kontroll- und Datenfluss für das Programm P_5 ergibt sich dann zu:

$$\phi_I := l_{boot} \wedge \neg l_0 \wedge \neg l_1 \wedge \neg l_2 \wedge \neg x \wedge \neg y$$

$$\phi_R := \left(\begin{array}{l} (l_{boot} \wedge \neg l_0 \wedge \neg l_1 \wedge \neg l_2 \wedge Xl_0 \wedge \neg Xl_1 \wedge \neg Xl_2 \wedge \neg Xl_{boot}) \vee \\ \left(\begin{array}{l} l_0 \wedge \neg l_1 \wedge \neg l_2 \wedge \neg Xl_0 \wedge Xl_1 \wedge Xl_2 \wedge \neg l_{boot} \wedge \neg Xl_{boot} \wedge \\ x \wedge y \wedge \neg Xx \wedge \neg Xy \end{array} \right) \vee \\ \left(\begin{array}{l} (l_1 \vee l_2) \wedge \neg Xl_1 \wedge \neg Xl_2 \wedge \neg Xl_0 \wedge \neg l_{boot} \wedge \neg Xl_{boot}) \wedge \\ \neg Xx \wedge \neg Xy \wedge (\neg l_0 \vee (x \wedge y)) \end{array} \right) \end{array} \right)$$

Die graphische Darstellung des Verhaltens des Programms P_5 ist in der folgenden Abbildung zu sehen. Hierbei ist der Startzustand durch eine doppelte Linie und der Terminalzustand durch ein Quadrat gekennzeichnet.

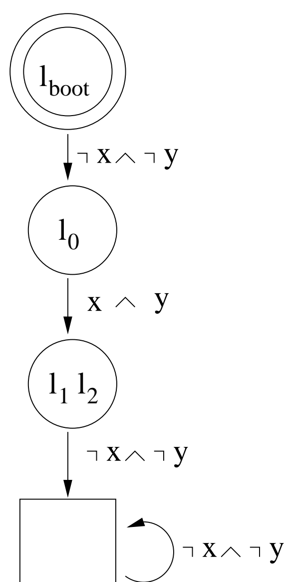

Abbildung 3.15: Verhalten des Programms P_5

3.4 Die Modellprüfung

Nachdem nun die Kripke Struktur des Programms als BDD-Repräsentation vorliegt, kann die Modellprüfung erfolgen. Hierzu werden zuerst alle in der Spezifikation angegebenen Fairnessbedingungen der CTL-Modellprüfung (ohne Fairness) unterzogen. Die jeweiligen Ergebnisse werden in einer Liste gesammelt, die dann der Modellprüfung mit Fairness übergeben wird.

Für die CTL-Modellprüfung wird noch eine Funktion *predecessor* benötigt, die zu einer Menge von Zuständen alle möglichen Vorgängerzustände berechnet. Diese Funktion bekommt als Eingabe eine Menge von Zuständen Q und liefert als Ergebnis die Menge aller Vorgängerzustände.

function $predecessor_{\exists}^{R}(Q)$
$\quad \exists l'_1, \cdots, l'_n \; [Q]_{l_1,\cdots,l_n}^{l'_1,\cdots,l'_n} \wedge R(l_1, \cdots, l_n, l'_1, \cdots, l'_n, x_1, \cdots, x_n)$
end

Bei der Modellprüfung wird der Syntaxbaum der spezifizierten CTL-Formel rekursiv abgearbeitet und es werden mit den in Kapitel 2 vorgestellten Fixpunktgleichungen die jeweiligen Zustandsmengen erzeugt, in denen alle Teilformeln der CTL-Formel gelten. Nachdem die gesamte CTL-Formel abgearbeitet wurde, erhält man als Ergebnis die Menge der Zustände in denen die Formel gilt.

Die rekursive Funktion für die Modellprüfung ohne Fairness ist in der folgenden Abbildung dargestellt, wobei ϕ die zu prüfende CTL-Formel ist.

Die in Abbildung 3.16 verwendeten Funktionen sind in den Abbildungen 3.17 bis 3.24 dargestellt.

function $States_{CTL}(\phi)$
case $\phi\, of$

$isvar(\phi)$	: **return** $BDD(\phi)$;
φ	: **return** $\neg\varphi$;
$\varphi \wedge \psi$	: **return** $States_{CTL}(\varphi) \wedge States_{CTL}(\psi)$;
$\varphi \vee \psi$	: **return** $States_{CTL}(\varphi) \vee States_{CTL}(\psi)$;
$EX\varphi$	: **return** $predecessor_{\exists}^{R}(States_{CTL}(\varphi))$;
$EG\varphi$	: **return** CheckEG(φ);
$EF\varphi$	: **return** CheckEF(φ);
$E[\varphi \text{ U } \psi]$	: **return** CheckEU(φ, ψ);
$E[\varphi \underline{\text{U}} \psi]$	: **return** CheckESU(φ, ψ);
$E[\varphi \text{ B } \psi]$	: **return** CheckEB(φ, ψ);
$E[\varphi \underline{\text{B}} \psi]$	: **return** CheckESB(φ, ψ);
$E[\varphi \text{ W } \psi]$	: **return** CheckEW(φ, ψ);
$E[\varphi \underline{\text{W}} \psi]$	: **return** CheckESW(φ, ψ);
$AX\varphi$	: $\neg EX(\neg\varphi)$;
$AG\varphi$	: $\neg EF(\neg\varphi)$;
$AF\varphi$	: $\neg EG(\neg\varphi)$;
$A[\varphi \text{ U } \psi]$	: $\neg E[(\neg\varphi) \underline{\text{B}} \psi]$;
$A[\varphi \underline{\text{U}} \psi]$	: $\neg E[(\neg\varphi) \text{ B } \psi]$;
$A[\varphi \text{ B } \psi]$	: $\neg E[(\neg\varphi) \underline{\text{U}} \psi]$;
$A[\varphi \underline{\text{B}} \psi]$	: $\neg E[(\neg\varphi) \text{ U } \psi]$;
$A[\varphi \text{ W } \psi]$	: $\neg E[(\neg\varphi) \underline{\text{W}} \psi]$;
$A[\varphi \underline{\text{W}} \psi]$	: $\neg E[(\neg\varphi) \text{ W } \psi]$;

end

Abbildung 3.16: CTL-Modellprüfung ohne Fairness

function CheckEG(φ)
    $Q_1 := \mathit{true}; Q_2 := M_\varphi \wedge predecessor_\exists^R(\mathit{true});$
    while $Q_1 \neq Q_2$ **do**
        $Q_1 := Q_2;$
        $Q_2 := M_\varphi \wedge predecessor_\exists^R(Q_2);$
    end
    return $Q_1;$
end

Abbildung 3.17: Die Funktion CheckEG(φ)

function CheckEF(φ)
    $M_\varphi = States_{CTL}(\varphi);$
    $Q_1 := \mathit{false}; Q_2 := M_\varphi \vee predecessor_\exists^R(\mathit{false});$
    while $Q_1 \neq Q_2$ **do**
        $Q_1 := Q_2;$
        $Q_2 := M_\varphi \vee predecessor_\exists^R(Q_2);$
    end
    return $Q_1;$
end

Abbildung 3.18: Die Funktion CheckEF(φ)

```
function CheckEU(φ, ψ)
    M_φ = States_CTL(φ);
    M_ψ = States_CTL(ψ);
    Q_1 := true; Q_2 := M_ψ ∨ M_φ ∧ predecessor_∃^R(true);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := M_ψ ∨ M_φ ∧ predecessor_∃^R(Q_2);
    end
    return Q_1;
end
```

Abbildung 3.19: Die Funktion CheckEU(φ, ψ)

```
function CheckESU(φ, ψ)
    M_φ = States_CTL(φ);
    M_ψ = States_CTL(ψ);
    Q_1 := false; Q_2 := M_ψ ∨ M_φ ∧ predecessor_∃^R(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := M_ψ ∨ M_φ ∧ predecessor_∃^R(Q_2);
    end
    return Q_1;
end
```

Abbildung 3.20: Die Funktion CheckESU(φ, ψ)

function CheckEB(φ, ψ)
$M_\varphi = States_{CTL}(\varphi)$;
$M_\psi = States_{CTL}(\psi)$;
$Q_1 := \mathit{true}; Q_2 := \neg M_\psi \wedge M_\varphi \vee predecessor^R_\exists(\mathit{true})$;
while $Q_1 \neq Q_2$ **do**
$Q_1 := Q_2$;
$Q_2 := \neg M_\psi \wedge M_\varphi \vee predecessor^R_\exists(Q_2)$;
end
return Q_1;
end

Abbildung 3.21: Die Funktion CheckEB(φ, ψ)

function CheckESB(φ, ψ)
$M_\varphi = States_{CTL}(\varphi)$;
$M_\psi = States_{CTL}(\psi)$;
$Q_1 := \mathit{false}; Q_2 := \neg M_\psi \wedge M_\varphi \vee predecessor^R_\exists(\mathit{false})$;
while $Q_1 \neq Q_2$ **do**
$Q_1 := Q_2$;
$Q_2 := \neg M_\psi \wedge M_\varphi \vee predecessor^R_\exists(Q_2)$;
end
return Q_1;
end

Abbildung 3.22: Die Funktion CheckESB(φ, ψ)

function CheckEW(φ, ψ)
$M_\varphi = States_{CTL}(\varphi)$;
$M_\psi = States_{CTL}(\psi)$;
$Q_1 := \mathit{true}; Q_2 := M_\psi \wedge M_\varphi \vee \neg M_\psi \wedge predecessor_\exists^R(\mathit{true})$;
while $Q_1 \neq Q_2$ **do**
$Q_1 := Q_2$;
$Q_2 := M_\psi \wedge M_\varphi \vee \neg M_\psi \wedge predecessor_\exists^R(Q_2)$;
end
return Q_1;
end

Abbildung 3.23: Die Funktion CheckEW(φ, ψ)

function CheckESW(φ, ψ)
$M_\varphi = States_{CTL}(\varphi)$;
$M_\psi = States_{CTL}(\psi)$;
$Q_1 := \mathit{false}; Q_2 := M_\psi \wedge M_\varphi \vee \neg M_\psi \wedge predecessor_\exists^R(\mathit{false})$;
while $Q_1 \neq Q_2$ **do**
$Q_1 := Q_2$;
$Q_2 := M_\psi \wedge M_\varphi \vee \neg M_\psi \wedge predecessor_\exists^R(Q_2)$;
end
return Q_1;
end

Abbildung 3.24: Die Funktion CheckESW(φ, ψ)

Nachdem die Fairnessbedingungen ausgewertet wurden, wird mit der Prüfung der Spezifikation begonnen. Dies geschieht durch eine weitere rekursive Funktion, die die CTL-Modellprüfung mit Fairness implementiert. Sei $Q = (Q_1, \cdots, Q_n)$ die Liste, die die Ergebnisse der Fairnessbedingungen enthält und $\phi := E_\phi G(\mathit{true})$ sowie ξ die zu prüfende CTL-Formel.

Die in Abbildung 3.25 verwendeten Funktionen sind in den Abbildungen 3.26 bis 3.33 dargestellt.

function $States_{CTL}\phi(\xi)$
case $\xi\ of$

$isvar(\xi)$	: **return** $BDD(\xi)$;
φ	: **return** $\neg\varphi$;
$\varphi \wedge \psi$	: **return** $States_{CTL_\phi}(\varphi) \wedge States_{CTL_\phi}(\psi)$;
$\varphi \vee \psi$	: **return** $States_{CTL_\phi}(\varphi) \vee States_{CTL_\phi}(\psi)$;
$E_\phi X\varphi$	: **return** $predecessor_\exists^R(States_{CTL_\phi}(\varphi) \wedge \phi)$;
$E_\phi G\varphi$	: **return** CheckfairEG (φ);
$E_\phi F\varphi$	: **return** CheckfairEF (φ);
$E_\phi[\varphi\ \mathrm{U}\ \psi]$	: **return** CheckfairEU (φ, ψ);
$E_\phi[\varphi\ \underline{\mathrm{U}}\ \psi]$	: **return** CheckfairESU (φ, ψ);
$E_\phi[\varphi\ \mathrm{B}\ \psi]$	: **return** CheckfairEB (φ, ψ);
$E_\phi[\varphi\ \underline{\mathrm{B}}\ \psi]$	: **return** CheckfairESB (φ, ψ);
$E_\phi[\varphi\ \mathrm{W}\ \psi]$	: **return** CheckfairEW (φ, ψ);
$E_\phi[\varphi\ \underline{\mathrm{W}}\ \psi]$	: **return** CheckfairESW (φ, ψ);
$AX\varphi$	: $\neg E_\phi X(\neg\varphi)$;
$AG\varphi$	: $\neg E_\phi F(\neg\varphi)$;
$AF\varphi$	: $\neg E_\phi G(\neg\varphi)$;
$A[\varphi\ \mathrm{U}\ \psi]$	: $\neg E_\phi[(\neg\varphi)\ \underline{\mathrm{B}}\ \psi]$;
$A[\varphi\ \underline{\mathrm{U}}\ \psi]$	: $\neg E_\phi[(\neg\varphi)\ \mathrm{B}\ \psi]$;
$A[\varphi\ \mathrm{B}\ \psi]$	: $\neg E_\phi[(\neg\varphi)\ \underline{\mathrm{U}}\ \psi]$;
$A[\varphi\ \underline{\mathrm{B}}\ \psi]$	: $\neg E_\phi[(\neg\varphi)\ \mathrm{U}\ \psi]$;
$A[\varphi\ \mathrm{W}\ \psi]$	: $\neg E_\phi[(\neg\varphi)\ \underline{\mathrm{W}}\ \psi]$;
$A[\varphi\ \underline{\mathrm{W}}\ \psi]$	: $\neg E_\phi[(\neg\varphi)\ \mathrm{W}\ \psi]$;

end

Abbildung 3.25: CTL-Modellprüfung mit Fairness

```
function CheckfairEG(φ)
    M_φ = States_{CTL_ϕ}(φ);
    P_1 := true;
    repeat
        P_0 := P_1;
        for i := 1 to n do
            Q_i := Q_i ∧ P_1
            R_{i,1} := Q_i;
            repeat
                R_{i,0} := R_{i,1};
                R_{i,1} := Q_i ∨ (predecessor^R_∃(R_{i,0} ∧ M_φ))
            until R_{i,1} = R_{i,0}
        end
        P_1 := M_φ ∧ ⋀_{i=1}^n predecessor^R_∃(R_{i,0});
    until P_1 = P_0; return P_1;
end
```

Abbildung 3.26: Die Funktion CheckfairEG(φ)

```
function CheckfairEF(φ)
    M_φ = States_{CTL_ϕ}(φ);
    Q_1 := false; Q_2 := M_φ ∧ ϕ ∨ predecessor^R_∃(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := M_φ ∧ ϕ ∨ predecessor^R_∃(Q_2);
    end
    return Q_1;
end
```

Abbildung 3.27: Die Funktion CheckfairEF(φ)

```
function CheckfairEU(φ, ψ)
    M_φ = States_{CTL_ϕ}(φ);
    M_ψ = States_{CTL_ϕ}(ψ);
    Q_1 := false; Q_2 := M_ψ ∧ ϕ ∨ M_φ ∧ predecessor^R_∃(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := M_ψ ∧ ϕ ∨ M_φ ∧ predecessor^R_∃(Q_2);
    end
    return (Q_1 ∨ E_ϕ Gφ);
end
```

Abbildung 3.28: Die Funktion CheckfairEU(φ, ψ)

```
function CheckfairESU(φ, ψ)
    M_φ = States_{CTL_ϕ}(φ);
    M_ψ = States_{CTL_ϕ}(ψ);
    Q_1 := false; Q_2 := M_ψ ∧ ϕ ∨ M_φ ∧ predecessor^R_∃(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := M_ψ ∧ ϕ ∨ M_φ ∧ predecessor^R_∃(Q_2);
    end
    return Q_1;
end
```

Abbildung 3.29: Die Funktion CheckfairESU(φ, ψ)

```
function CheckfairEB(φ, ψ)
    M_φ = States_{CTL_ϕ}(φ);
    M_ψ = States_{CTL_ϕ}(ψ);
    Q_1 := false; Q_2 := ¬M_ψ ∧ M_φ ∧ ϕ ∨ predecessor^R_∃(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := ¬M_ψ ∧ M_φ ∧ ϕ ∨ predecessor^R_∃(Q_2);
    end
    return (Q_1 ∨ E_ϕ G(¬φ ∧ ¬ψ));
end
```

Abbildung 3.30: Die Funktion CheckfairEB(φ, ψ)

```
function CheckfairESB(φ, ψ)
    M_φ = States_{CTL_ϕ}(φ);
    M_ψ = States_{CTL_ϕ}(ψ);
    Q_1 := false; Q_2 := ¬M_ψ ∧ M_φ ∧ ϕ ∨ predecessor^R_∃(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := ¬M_ψ ∧ M_φ ∧ ϕ ∨ predecessor^R_∃(Q_2);
    end
    return Q_1;
end
```

Abbildung 3.31: Die Funktion CheckfairESB(φ, ψ)

```
function CheckfairEW(φ, ψ)
    M_φ = States_{CTL_φ}(φ);
    M_ψ = States_{CTL_φ}(ψ);
    Q_1 := false; Q_2 := M_ψ ∧ φ ∧ M_φ ∨ ¬M_ψ ∧ φ ∧ predecessor_∃^R(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := M_ψ ∧ φ ∧ M_φ ∨ ¬M_ψ ∧ φ ∧ predecessor_∃^R(Q_2);
    end
    return (Q_1 ∨ E_φ G(¬ψ));
end
```

Abbildung 3.32: Die Funktion CheckfairEW(φ, ψ)

```
function CheckfairESW(φ, ψ)
    M_φ = States_{CTL_φ}(φ);
    M_ψ = States_{CTL_φ}(ψ);
    Q_1 := false; Q_2 := M_ψ ∧ φ ∧ M_φ ∨ ¬M_ψ ∧ φ ∧ predecessor_∃^R(false);
    while Q_1 ≠ Q_2 do
        Q_1 := Q_2;
        Q_2 := M_ψ ∧ φ ∧ M_φ ∨ ¬M_ψ ∧ φ ∧ predecessor_∃^R(Q_2);
    end
    return Q_1;
end
```

Abbildung 3.33: Die Funktion CheckfairESW(φ, ψ)

Die Funktion $States_{CTL}\phi$ liefert als Ergebnis eine Menge von Zuständen, in denen die angebene Spezifikation gilt. Als Nächstes wird nun ermittelt, ob sich der Startzustand in dieser Menge befindet. Wenn die Implikation $\phi_I \rightarrow States_{CTL}\phi(\xi)$ als Ergebnis *true* liefert, dann gilt die angegebene CTL-Formel ξ in der Struktur, ansonsten nicht.

3.5 Aufrufsyntax des Werkzeugs

In diesem Abschnitt wird die Aufrufsyntax des in der Arbeit implementierten Werkzeugs beschrieben. Der Aufruf des Werkzeugs geschieht mit der Befehlszeile:

purr_parser -v *Specname* [Optionen] *Eingabedatei*

Eingaben und Optionen:

-v *Specname*	*Specname* ist der Name der zu prüfenden Spezifikation und muss im Spezifikationsteil des Programmtextes vorkommen. Die Angabe einer zu prüfenden Spezifikation ist zum Starten des Werkzeugs erforderlich.
Eingabedatei	Diese Eingabe legt den Namen der Datei fest, in der sich das zu verifizierende PURR-Programm mit den Spezifikationen befindet.
- I	Mit dieser Option wird der BDD für *instant*(S) in die Datei INSTANT gespeichert. S ist hierbei der Modulrumpf des zu verifizierenden PURR-Programms.
- E	Mit dieser Option wird der BDD für *enter*(S) in die Datei ENTER gespeichert. S ist hierbei der Modulrumpf des zu verifizierenden PURR-Programms.
- M	Mit dieser Option wird der BDD für *move*(S) in die Datei MOVE gespeichert. S ist hierbei der Modulrumpf des zu verifizierenden PURR-Programms.
- T	Mit dieser Option wird der BDD für *terminate*(S) in die Datei TERMINATE gespeichert. S ist hierbei der Modulrumpf des zu verifizierenden PURR-Programms.
- R	Mit dieser Option werden alle erfüllenden Belegungen der Transitionsrelation in die Datei RELATION gespeichert. Die erfüllenden Belegungen der Transitionsrelation sind alle Pfade, die von der Wurzel des BDD's der Transitionsrelation zu einem mit *true* markierten Blatt führen.
-smv *Dateiname*	Mit dieser Option wird die Transitionsrelation in eine Datei im SMV-Format geschrieben. *Dateiname* bezeichnet den Namen der Datei, in die die Transitionsrelation geschrieben werden soll.

3.6 Experimente

In diesem Abschnitt werden einige Experimente, die den Aufbau der Transitionsrelation und die Modellprüfung zeigen, behandelt. Die Syntax, die in den Spezifikationen verwendet wird ist:

NOT	NOT
&	AND
|	OR
->	$\rightarrow$
XOR	XOR
ONEPATH	E *(there exists a path)*
ALLPATH	A *(for all paths)*
X	X *(next)*
F	F *(sometimes)*
G	G *(always)*
W	W *(when)*
SW	$\underline{\text{W}}$ *(strong when)*
U	U *(until)*
SU	$\underline{\text{U}}$ *(strong until)*
B	B *before*
SB	$\underline{\text{B}}$ *(strong before)*

Das erste Beispiel ist das Programm P_7.

```
module P7 input i output x,y,z

present i then l1:pause else emit x end present;
emit y after 1;
l2 : pause;
present y then emit z else emit y end present

spec s1 ONEPATH F (NOT(i) -> x) with Fairness TRUE,
spec s2 ONEPATH F (y&z) with Fairness TRUE,
spec s3 ONEPATH F l2 with Fairness TRUE
spec s4 ONEPATH F (i&x) with Fairness TRUE

end module
```

Dieses Programm zeigt folgendes Verhalten:

Im Startzustand l_{boot}, vgl. Definition 19, wird je nachdem, ob i zu 0 ausgewertet werden kann oder nicht, x emittiert. Falls i nicht zu 0 ausgewertet werden kann, vergeht eine Zeiteinheit, bevor die nächste Anweisung ***emit*** y ***after 1*** abgearbeitet wird. Zum nächsten Zeitpunkt werden dann auf jeden Fall y und z emittiert, wonach die Abarbeitung des Programms beendet wird. In der folgenden Abbildung sind der Kontrollfluss sowie die Transitionsrelation des Programms graphisch dargestellt. Der Startzustand ist durch eine zusätzliche Linie und der Terminalzustand durch ein Quadrat gekennzeichnet.

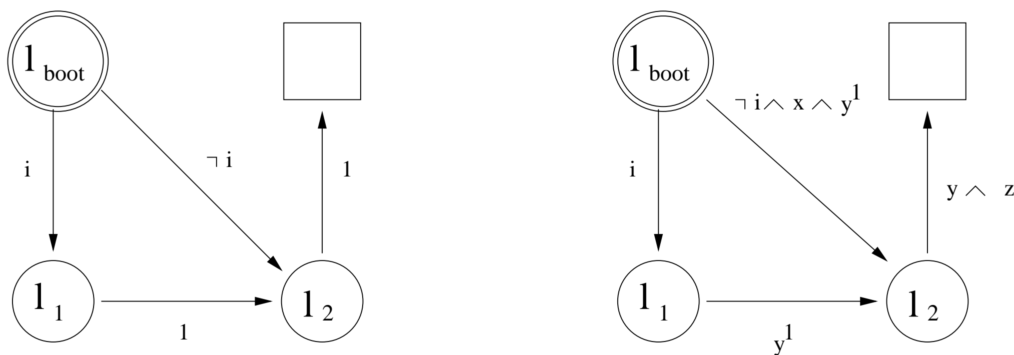

Abbildung 3.34: Kontroll- und Datenfluss des Programms P_7

Die im Programmtext angegebenen Spezifikationen besagen:

s_1 : Es gibt einen Pfad, auf dem irgendwann $\neg\, i$ gilt, woraus folgt, dass x gilt.
s_2 : Es gibt einen Pfad, auf dem irgendwann y und z gelten.
s_3 : Es gibt einen Pfad, auf dem irgendwann der Zustand l_2 erreicht wird.
s_4 : Es gibt einen Pfad, auf dem irgendwann i und x gelten.

Während die ersten drei Spezifikationen wahr sind, kann die vierte niemals erfüllt werden, denn x soll ja genau dann emittiert werden, wenn i nicht gilt. Dies ließ sich auch durch Programmläufe der Implementierung bestätigen.

Als zweites Beispiel soll das bereits in Abschnitt 3.3.1 vorgestellte Programm *ABRO* dienen:

```
module ABRO
input a,b,r
output o
loop
  abort
    (la:await a ||lb: await b);
    emit o;
    lr:await r
  when r end abort
end loop

spec s1 ONEPATH F ((a&b) -> (o)) with Fairness TRUE,
spec s2 ONEPATH F (o -> ONEPATH X lr) with Fairness TRUE,
spec s3 ONEPATH F (r-> (la&lb)) with Fairness TRUE,
spec s4 ONEPATH F (o&lr) with Fairness TRUE

end module
```

Das Verhalten dieses Programms wurde bereits in Abschnitt 3.3.1 beschrieben.

Die im Programmtext angegebenen Spezifikationen besagen:

s_1 : Es gibt einen Pfad, auf dem irgendwann a und b gelten, woraus folgt, dass dann auch o gilt.
s_2 : Es gibt einen Pfad, auf dem irgendwann gilt, dass sich der Kontrollfluss nach der Emission von o im Zustand l_r befindet.
s_3 : Es gibt einen Pfad, auf dem irgendwann gilt, dass sich der Kontrollfluss nach dem Eintreten von r im Zustand $l_a l_b$ befindet.
s_4 : Es gibt einen Pfad, auf dem irgendwann gilt, dass sich der Kontrollfluss im Zustand l_r befindet und o ausgegeben wird.

Während die ersten drei Spezifikationen wahr sind, kann die vierte niemals erfüllt werden, denn wenn sich der Kontrollfluss im Zustand l_r befindet, dann wurde vorher schon o ausgegeben. Auch diese Spezifikationen konnten durch Programmläufe des Modellprüfers bestätigt werden.

Als letztes Beipiel dient nun der Programmcode eines Arbiters: [16]

```
module arbiter
input alpha1, alpha2, arb
output crit1a, crit1b, crit2a, crit2b

signal rho1, rho2, fA in
  (
    (
      (
        loop
          weak abort
            sustain rho1
          when immediate alpha1 end abort;
          emit crit1a;
          pause;
          emit crit1b;
          emit fA
        end loop
      )
      ||
      (
        loop
          weak abort
            sustain rho2
          when immediate alpha2 end abort;
          emit crit2a;
          pause;
          emit crit2b;
          emit fA
        end loop
      )
    )
    ||
    (
      signal t1, t2, p1, p2 in
        (
          (
            (
              (
```

```
      loop
        abort
          sustain t1
        when arb end abort;
        abort
          sustain t2
        when arb end abort
      end loop
    )
    ||
    (
      loop
        await (rho1 & t1);
        weak abort
          sustain p1
        when ~rho1 end abort
      end loop
    )
  )
  ||
  (
    loop
      await (rho2 & t2);
      weak abort
        sustain p2
      when ~rho2 end abort
    end loop
  )
)
||
(
  loop
    await immediate (rho1 | rho2);
    emit arb;
    present (t1 & p1) | (t2 & p2) then
      present
        case rho1 & t1 & p1 do
          emit alpha1
        case rho2 & t2 & p2 do
```

```
                    emit alpha2
                end present
              else
                present
                  case rho1 do
                    emit alpha1
                  case rho2 do
                    emit alpha2
                end present
              end present;
              await immediate fA;
              pause
            end loop
          )
        )
      end signal
    )
  )
end signal

spec s1 ONEPATH F fA with Fairness TRUE

end module
```

Für dieses Programm war eine Verifikation nicht möglich, da die Transitionsrelation dieses Programms so groß wird, dass der Aufbau des entsprechenden BDD's mehr als 1024 MB Speicher benötigt. Das Experiment mit diesem Programm war daher nicht in vernünftiger Zeit durchführbar. Eine Verkleinerung der Transitionsrelation lässt sich möglicherweise durch eine Änderung der Definitionen für *instant*(S), *enter*(S), *move*(S) und *terminate*(S) erreichen.

Kapitel 4

Schlussfolgerungen und Ausblick

4.1 Schlussfolgerungen

In dieser Arbeit wurde gezeigt, wie man für Programme, die in der synchronen Programmiersprache PURR verfasst sind, eine Umsetzung des Programmtextes in eine Kripke Struktur schaffen kann. Hierzu war es notwendig, die Transitionsrelationen für den Kontroll- und den Datenfluss getrennt voneinander zu erzeugen und nach der Erzeugung zu kombinieren. Bei der Erzeugung der Transitionsrelationen entsteht eine große Menge an nicht erreichbaren Zuständen, die die Transitionsrelationen sehr komplex machen, was sich unter anderem auf den benötigten Speicherbedarf auswirkt. Es wurde weiterhin gezeigt, wie man auf der erzeugten Kripke Struktur aufbauend einen Modellprüfer für die temporale Logik CTL implementieren kann, um eine CTL-Modellprüfung für in der Sprache PURR verfasste Programme durchführen zu können. Dieser Modellprüfer ist in der Lage, im Programmtext vorgegebene Spezifikationen, die auch Fairnesseigenschaften enthalten können, zu verifizieren. Neben den oben beschriebenen Zielen der Arbeit gibt es noch einige Verbesserungs- und Erweiterungsmöglichkeiten, die im nächsten Abschnitt beschrieben werden.

4.2 Ausblick und Erweiterungsmöglichkeiten

Ein wichtiger Punkt für eine Erweiterung des Modellprüfers ist die Entwicklung einer Funktion zur Generierung von Gegenbeispielen. In der vorliegenden Implementierung des Modellprüfers bekommt der Anwender lediglich das Ergebnis der Modellprüfung mitgeteilt. Zur Fehlersuche im Programmtext wäre es jedoch hilfreich, wenn der Modellprüfer für den Fall, dass die angegebene Spezifikation nicht wahr ist, ein Gegenmodell erzeugt. Ein solches Gegenmodell enthält dann eine Zustandsmenge, die vom Startzustand aus erreichbar ist, in der die Spezifikation jedoch nicht gilt.

Ein wichtiger Ansatz für Verbesserungen ist das Erzeugen der Transitionsrelationen für den Kontroll- und den Datenfluss eines PURR-Programms. Bei der in dieser Arbeit vorgestellten Variante entstehen bei der Erzeugung des Kontroll- und des Datenflusses nicht erreichbare Zustände, die die Transitionsrelationen unnötig komplex machen. Hier könnte man den Versuch unternehmen, die Definitionen, die zur Erzeugung der BDD's für $\mathit{enter}(S)$, $\mathit{move}(S)$, $\mathit{terminate}(S)$ und $\mathit{instant}(S)$ dienen, so zu verändern, dass zwar die Semantik erhalten bleibt, aber nur noch die Zustände und Transitionen erzeugt werden, die auch tatsächlich vom Startzustand aus erreicht werden können. Dies hätte zur Folge, dass die Transitionsrelationen für den Kontroll- und den Datenfluss weniger Zustände umfassen, was sich in der Größe der benötigten BDD's zur Repräsentation der Transitionsrelationen bemerkbar machen würde. Eine Verkleinerung der Transitionsrelationen hätte auch zur Folge, dass der Speicherbedarf zur Erzeugung der Transitionsrelationen weniger groß wäre und auch die Modellpüfung hätte eine geringere Laufzeit.

Eine zusätzliche sinnvolle Erweiterung wäre die Entwicklung eines Visualisierungswerkzeuges, welches eine graphische Darstellung der Kripke Struktur erzeugen könnte. Diese graphische Darstellung kann die Fehlersuche in der Transitionsrelation, aber auch in dem eingelesenen Programmtext erheblich vereinfachen, da eine solche Darstellung sehr viel einfacher zu lesen ist, als die Ausgaben der meisten BDD-Pakete. Das Problem bei der Erzeugung einer solchen Beschreibung ist jedoch, dass man entweder darauf angewiesen ist, dass die zu visualisierende Transitionensrelation keine unerreichbaren Zustände enthält, oder aber man muss vom Startzustand ausgehend die Transitionsrelation durchlaufen und alle erreichbaren Zustände ermitteln. Auch hierfür wäre es also hilfreich, wenn die Transitionsrelationen keine unerreichbaren Zustände mehr enthielten.

Da der BDD, welcher die Transitionsrelation repräsentiert, das Verhalten des Systems als endlichen Automaten beschreibt, könnte man aus dieser Darstellung eine Beschreibung in einer Hardware-Beschreibungssprache, wie z.B. VHDL oder Verilog, erzeugen. Diese Beschreibung kann dann mit weiteren Werkzeugen weiterverarbeitet werden, um z.B. eine Beschreibung der Schaltung auf Transistorebene zu erhalten.

Das in dieser Arbeit verwendete BDD-Paket bietet die Möglichkeit, einen BDD in eine Datei zu schreiben und zu einem späteren Zeitpunkt wieder einzulesen. Daher wäre es nicht notwendig, die in den oberen beiden Absätzen beschriebenen Erweiterungen noch in den vorliegenden Programmtext mit aufzunehmen. Die beiden beschriebenen Erweiterungen können durchaus durch eigenständige Programme implementiert werden. Es müsste hierzu lediglich eine Funktion zum Speichern des BDD's für die Transitionsrelation in den Programmtext eingefügt werden.

Anhang A

Die Grammatik des Parsers

In diesem Abschnitt befindet sich die Grammatik des Parsers, der zum Einlesen der PURR-Programme verwendet wird.

```
purr_file: module_list
           ;

module_list:   modul
             | module_list modul
             ;

modul:    MODULE variable INPUT declarelist OUTPUT declarelist statespec
          END_MODULE
        | MODULE variable OUTPUT declarelist statespec END_MODULE
        ;

statespec : statement0 speclist
            ;

fairlist :     expra
             | expra PROZENT fairlist
              ;

speclist:      SPEC variable expra WITHF fairlist
             | SPEC variable expra
             | SPEC variable expra COMMA speclist
             | SPEC variable expra WITHF fairlist COMMA speclist
              ;

expra:    IF expra THEN expra ELSE expra
| LET variable EQUIV expra IN expra
| expr1
```

```
expr1:    expr1 ATNEXT expr2
| expr1 SATNEXT expr2
| expr2

expr2:    expr2 BEFORE expr3
| expr2 SBEFORE expr3
| expr3

expr3:    expr3 UNTIL1 expr4
| expr3 SUNTIL expr4
| expr4

expr4:    expr4 WHEN1 expr5
| expr4 SWHEN expr5
| expr5

expr5:    expr5 EQUIV expr6
| expr5 XOR expr6
| expr6

expr6:    expr6 IMPLIES expr7
| expr7

expr7:    expr7 OR expr8
| expr8

expr8:    expr8 AND expr9
| expr9

expr9:    variable
| FALSE
| TRUE
| NEXT expr9
| ALWAYS expr9
| EVENTUAL expr9
| ONEPATH expr9
| ALLPATH expr9
| NEG expr9
| DEL11 expra DEL12
| DEL21 expra DEL22

delay:   variable
       | bool_expr
       | IMMEDIATE bool_expr
       | IMMEDIATE variable
       | number TIMES bool_expr
```

```
        | number TIMES variable
        ;


bool_expr:   bool_expr OR bool_expr1
           | variable OR bool_expr1
           | bool_expr OR variable
           | variable OR variable
           | bool_expr1
           ;

bool_expr1:   bool_expr1 AND bool_expr2
            | variable AND bool_expr2
            | bool_expr1 AND variable
            | variable AND variable
            | bool_expr2
            ;

bool_expr2:   NOT bool_expr2
            | NOT variable
            | DEL11 bool_expr DEL12
            | bitvec_expr LESS bitvec_expr
            | dualdigit LESS bitvec_expr
            | bitvec_expr LESS dualdigit
            | dualdigit LESS dualdigit
            | bitvec_expr GREATER bitvec_expr
            | dualdigit GREATER bitvec_expr
            | bitvec_expr GREATER dualdigit
            | dualdigit GREATER dualdigit
            | bitvec_expr EQUAL bitvec_expr
            | dualdigit EQUAL bitvec_expr
            | bitvec_expr EQUAL dualdigit
            | dualdigit EQUAL dualdigit

variable: IDENTIFIER
    ;


number: DIGIT
          ;


bitvec_expr:   bitvec_expr PLUS bitvec_expr1
             | bitvec_expr PLUS dualdigit
             | dualdigit PLUS bitvec_expr1
             | dualdigit PLUS dualdigit
             | bitvec_expr MINUS dualdigit
```

```
             | dualdigit MINUS dualdigit
             | dualdigit MINUS bitvec_expr1
             | bitvec_expr MINUS bitvec_expr1
             | bitvec_expr1
             ;

bitvec_expr1:   bitvec_expr1 MULT bitvec_expr2
              | bitvec_expr1 MULT dualdigit
              | dualdigit MULT bitvec_expr2
              | dualdigit MULT dualdigit
              | bitvec_expr1 DIV dualdigit
              | bitvec_expr1 DIV bitvec_expr2
              | bitvec_expr1 MOD dualdigit
              | bitvec_expr1 MOD bitvec_expr2
              | bitvec_expr2
              ;

bitvec_expr2:   variable
              | CONCAT DEL11 bitvec_expr COMMA bitvec_expr DEL12
              | CONCAT DEL11 dualdigit COMMA bitvec_expr DEL12
              | CONCAT DEL11 bitvec_expr COMMA dualdigit DEL12
              | CONCAT DEL11 dualdigit COMMA dualdigit DEL12
              | CHOOSE variable bitvec_expr2
              | DEL11 bitvec_expr DEL12
              | variable DEL21 number DEL22
              | variable DEL21 number DOT DOT number DEL22
              ;


dualdigit: DUALDIGIT


expr:   bool_expr
      | bitvec_expr
      | dualdigit
      ;


caselist:   CASE delay DO statement0
          | CASE delay DO statement0 caselist
          | CASE delay DO statement0 ELSE statement0
          ;

pcaselist:  CASE bool_expr DO statement0
          | CASE bool_expr DO statement0 caselist
          | CASE bool_expr DO statement0 ELSE statement0
          | CASE variable DO statement0
```

```
          | CASE variable DO statement0 caselist
          | CASE variable DO statement0 ELSE statement0

;


elselist:   ELSIF expr THEN statement0
          | elselist ELSIF expr THEN statement0
          ;

handlelist:   HANDLE expr DO statement0
            | handlelist HANDLE expr DO statement0
            ;

declarelist:   variable
             | variable COLON number
             | variable COMMA declarelist
             | variable COLON number COMMA declarelist
             ;

var_declarelist:   variable COLON number
                 | variable COLON number COMMA var_declarelist
                 ;

paramlist:   variable variable SLASH variable
           | variable variable SLASH variable COMMA paramlist
           ;

statement0 :   statement0 PARALLEL statement1
             | statement1
             ;

statement1 :  NOTHING
            | HALT
            | PAUSE
            | EXIT variable
            | variable COLON HALT
            | variable COLON PAUSE
            | variable ZUWEIS expr
            | variable ZUWEIS expr AFTER number
            | EMIT variable
            | EMIT variable AFTER number
            | SUSTAIN variable
            | SUSTAIN variable AFTER number
            | SUSTAIN variable bitvec_expr
            | SUSTAIN variable bitvec_expr AFTER number
            | variable COLON SUSTAIN variable
```

```
| variable COLON SUSTAIN variable AFTER number
| variable COLON SUSTAIN variable bitvec_expr
| variable COLON SUSTAIN variable bitvec_expr AFTER number
| LOOP statement1 END_LOOP
| REPEAT number TIMES statement1 END_REPEAT
| REPEAT statement1 UNTIL bool_expr
| POSITIVE_REPEAT number TIMES statement1 END_REPEAT
| PRESENT bool_expr THEN statement1 ELSE statement1 END_PRESENT
| PRESENT bool_expr THEN statement1 END_PRESENT
| PRESENT bool_expr ELSE statement1 END_PRESENT
| PRESENT pcaselist END_PRESENT
| IF bool_expr THEN statement1 ELSE statement1 END_IF
| IF bool_expr THEN statement1 END_IF
| IF bool_expr ELSE statement1 END_IF
| IF bool_expr THEN statement1 elselist END_IF
| AWAIT delay
| AWAIT caselist END_AWAIT
| variable COLON AWAIT delay
| variable COLON AWAIT caselist END_AWAIT
| ABORT statement1 WHEN delay END_ABORT
| ABORT statement1 WHEN delay DO statement1 END_ABORT
| ABORT statement1 WHEN caselist END_ABORT
| WEAK_ABORT statement1 WHEN delay END_ABORT
| WEAK_ABORT statement1 WHEN delay DO statement1 END_ABORT
| WEAK_ABORT statement1 WHEN caselist END_ABORT
| LOOP statement1 EACH variable
| variable COLON LOOP statement1 EACH variable
| EVERY delay DO statement1 END_EVERY
| variable COLON EVERY delay DO statement1 END_EVERY
| SUSPEND statement1 WHEN delay
| WEAK_SUSPEND statement1 WHEN delay
| TRAP variable IN statement1 END_TRAP
| TRAP variable IN statement1 handlelist END_TRAP
| statement1 SEQUENZ statement1
| RUN variable
| DEL11 statement0  DEL12
| RUN variable DEL21 paramlist DEL22
| SIGNAL declarelist IN statement1 END_SIGNAL
| SIGNAL variable COLON number IN statement1 END_SIGNAL
| VAR var_declarelist IN statement1 END_VAR
| VALUE variable
| WHILE bool_expr DO statement0 END_WHILE
;
```

Literaturverzeichnis

[1] BERRY, G. The constructive semantics of pure Esterel, May 1996.

[2] BERRY, G. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 1998.

[3] BRYANT, R. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers C-35*, 8 (August 1986), 677–691.

[4] EDWARDS, S. *The Specification and Execution of Heterogeneous Reactive Systems*. PhD thesis, University of California, Berkeley, 1997. http://ptolemy.eecs.berkeley.edu/papers/97/sedwardsThesis/.

[5] EMERSON, E. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science* (Amsterdam, 1990), J. van Leeuwen, Ed., vol. B, Elsevier Science Publishers, pp. 996–1072.

[6] GORDON, M., AND MELHAM, T. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[7] GUPTA, A. Formal Hardware Verification Methods: A Survey. *Journal of Formal Methods in System Design 1* (1992), 151–238.

[8] KOZEN, D. Results on the propositional μ-calculus. In *International Colloquium on Automata, Languages, and Programming* (1982), pp. 348–359.

[9] KOZEN, D. Results on the propositional mu-calculus. *Theoretical Computer Science 27* (December 1983), 333–354.

[10] KROPF, T., RUF, J., SCHNEIDER, K., AND WILD, M. A synchronous language for modeling and verifying real time and embedded systems. In *GI/ITG/GME Workshop: Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme und Beschreibungssprachen und Modellierung von Schaltungen und Systemen* (1998), HNI-Verlagsschriften, ISBN 3-931466-35-3, pp. 11–20.

[11] KROPF, T., RUF, J., SCHNEIDER, K., AND WILD, M. The synchronous system description language PURR. In *Open Project Workshop on System Design Automation (SDA98)* (Dresden, Germany, 1998).

[12] LAPRIE, J. Dependability: Basic concepts and associated terminology. Tech. Rep. 31, PDCS, May 1990.

[13] OWRE, S., RUSHBY, J., SHANKAR, N., AND SRIVAS, M. A tutorial on using PVS for hardware verification. In *International Conference on Theorem Provers in Circuit Design (TPCD)* (Bad Herrenalb, Germany, September 1994), T. Kropf and R. Kumar, Eds., vol. 901 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 258–279. published 1995.

[14] PRATT, V. A decidable μ-calculus: Preliminary report. In *IEEE Symposium on Foundations of Computer Science* (1982), pp. 421–427.

[15] SANGHAVI, J., RANJAN, R., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. High performance BDD package by exploiting memory hierarchy. In *Design Automation Conference (DAC96)* (Las Vegas NV, June 1996), pp. 635–640.

[16] SCHNEIDER, K., AND SABELFELD, V. Introducing mutual exclusion in Esterel. In *Andrei Ershov Third International Conference Perspectives of Systems Informatics* (Novosibirsk, Akademgorodok, Russia, July 1999), Lecture Notes in Computer Science, Springer Verlag.

[17] SENTOVICH, E. M. A brief study of BDD package performance. In *International Conference on Formal Methods in Computer Aided Design (FMCAD)* (Palo Alto, CA, USA, November 1996), M. Srivas and A. Camilleri, Eds., vol. 1166 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 389–403.

[18] THOMAS, W. Automata on infinite objects. In *Handbook of Theoretical Computer Science* (Amsterdam, 1990), J. van Leeuwen, Ed., vol. B, Elsevier Science Publishers, pp. 133–191.