Design und Implementierung einer Plattform zur Verifikation verteilter Systeme

von
Stephan Tobies
Matrikelnummer 191757

Diplomarbeit

im Fach Informatik

vorgelegt an der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinisch-Westfälischen Technischen Hochschule Aachen
im Februar 1998

Angefertigt am
Lehrstuhl für Informatik II
bei
Univ.–Prof. Dr. Klaus Indermark

Ich danke Herrn Prof. Dr. Klaus Indermark für die Überlassung des Themas, Martin Leucker für die Betreuung sowie meiner Frau und meinen Eltern für moralische und orthographische Unterstützung.

Ich versichere, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Aachen, am 25. Februar 1998

Inhaltsverzeichnis

Einleitung **1**

1. Truth **4**
1.1. Benutzerschnittstelle 5
1.2. CCS und Prozeßalgebra 6
1.3. Semantische Bereiche 7
1.4. Transitionssysteme 7
1.5. Logik und Modelchecking 8
1.6. Simulation 10

2. Implementierungsumgebung **11**
2.1. Haskell 11
2.1.1. Typsystem 11
2.1.2. Algebraische Datentypen und Pattern Matching 12
2.1.3. Monadische Ein– und Ausgabe 12
2.1.4. Standard–Bibliotheken 12
2.2. Der Glasgow Haskell Compiler 12
2.2.1. Zustands–Monaden 13
2.2.2. Unboxed Werte 13
2.2.3. Bibliotheken 14
2.2.4. Eindeutige Bezeichner 16
2.2.5. Weitere Eigenschaften 16
2.3. Happy 17
2.4. DaVinci 17
2.5. Java 18

3. Spezifikationen **19**
3.1. CCS 19
3.1.1. Syntax 20
3.1.2. Semantik 21

3.2. Alternativen 24
3.2.1. Spezifikationssprachen 24
3.2.2. Semantische Bereiche 26
3.3. Implementierung in Truth 27
3.3.1. Die Prozeßalgebra–Schnittstelle 27
3.3.2. Implementierung der CCS–Prozeßalgebra 29
3.3.3. Erzeugung und Speicherung des Transitionssystems 32
3.3.4. Analysen 37
3.3.5. Ausgabe des Transitionssystems 39
3.4. Laufzeit–Messungen 40

4. Logiken für verteilte Systeme 43
4.1. Temporale Logiken 43
4.1.1. Die Logik CTL* 44
4.2. Der μ-Kalkül 46
4.2.1. Syntax von $L\mu$ 46
4.2.2. Semantik von $L\mu$ 47
4.2.3. Übersetzungen von LTL, CTL und CTL* 48
4.3. Modelchecking für $L\mu$ 48
4.3.1. Entscheidbarkeit 49
4.3.2. Komplexität 49
4.3.3. Globales und lokales Modelchecking 51
4.3.4. Tableau–basiertes Modelchecking 52
4.3.5. Effizienzbetrachtungen 58
4.4. Implementierung in TRUTH 59
4.4.1. Modifikationen der Logik 60
4.4.2. Der Datentyp für Formeln 61
4.4.3. Modelchecking 62
4.4.4. Andere Modelchecker 67
4.5. Laufzeit–Messungen 67

5. Simulation von CCS–Prozessen 69
5.1. Prozesse in CCS 70
5.2. Die `Haskell`–Implementierung 71
5.2.1. Datenstrukturen 72
5.2.2. Definition der Schnittstelle 72
5.3. Die Darstellungskomponente 75
5.3.1. Implementierung der Darstellungskomponente 75

6. Ein Beispiel: Das Alternating Bit Protocol **80**

Zusammenfassung und Ausblick **87**

A. Halbordnungen und Verbände **89**

B. Tests **92**

C. Quelldateien **94**

Abbildungsverzeichnis

1.1. Architektur von TRUTH 5
3.1. CCS–Einzelschritt–Relation 22
3.2. Semantik einer CCS–Spezifikation 23
3.3. Eine Beispiel–Ableitung eines CCS–Ausdrucks 23
3.4. Ausgabe eines Transitionssystems durch *daVinci* 39
4.1. Regeln des Tableau–Systems 53
4.2. Abgeleitete Tableau–Regeln 54
4.3. Ein Beispiel–Tableau 55
4.4. Ein naiver Modelchecking Algorithmus 58
4.5. Ein optimiertes Tableau 60
5.1. Grammatik der Initialisierung 73
5.2. Graphische Elemente der Simulation 76
5.3. Die Darstellungskomponente 77
6.1. Eine schematische Übersicht des ABP 81
6.2. Die Spezifikation des ABP 82
6.3. Ausgabe von Transitionssystemen mit *daVinci* 84
6.4. Das Transitionssystem des ABP 85
6.5. Simulation des ABP 86

Einleitung

Die Entwicklung verläßlicher verteilter Systeme ist eine große Herausforderung, der sich die Informatik zur Zeit stellen muß. Die Schwierigkeit dieser Aufgabe hat im Wesentlichen zwei Gründe: Zum einen hat die Verteilung und der Einsatz in Echtzeit–Umgebungen zur Folge, daß die verwendeten Programme mit einem hohen Maß an Nicht–Determinismus umgehen müssen. Zum anderen erhöht die Nebenläufigkeit der Systeme deren Komplexität. Dabei findet man solche Systeme oft in Bereichen, in denen ein fehlerfreies Arbeiten gewährleistet sein muß. Als Beispiel hierfür seien moderne Flugzeugsteuerungen und Kraftwerk–Leitsysteme genannt.

Ausgehend von der Erkenntnis, daß Menschen Fehler machen und daß Tests von Systemen niemals die Abwesenheit von Fehlern garantieren können, besteht eine weitgehende Übereinstimmung, daß formale Methoden bei der Entwicklung solcher Systeme von großem Nutzen sind. Die Größe und Komplexität solcher Systeme macht dabei die Verwendung automatischer Werkzeuge notwendig.

Folgerichtig existiert eine Reihe von Werkzeugen, die die Anwendung formaler Methoden bei der Entwicklung und Verifikation verteilter Systeme unterstützt. Dabei sind zwei wesentliche Strömungen auszumachen [CW96]:

- Theorem–Beweiser verfolgen einen Ansatz, bei dem sowohl das System als auch die gewünschten Eigenschaften in einer geeigneten Logik ausgedrückt werden. Der Nachweis der Korrektheit erfolgt durch den Beweis, daß die Eigenschaften aus den Axiomen des Systems nach festgelegten Schlußregeln abzuleiten sind. Dabei ist oftmals eine Anleitung und Unterstützung des Beweisers durch den Benutzer notwendig, was zu Fehlern im Beweis führen kann. Von Vorteil sind Theorem–Beweiser insbesondere bei der Untersuchung von Systemen mit unendlichen Zustandsräumen.

- Bei Systemen, die einen endlichen Zustandsraum haben oder deren Zustandsraum sich zumindest geeignet endlich repräsentieren läßt, findet die Methode des Modelcheckings Anwendung [CES86]. Modelchecker basieren auf einer endlichen Darstellung des betrachteten Systems und erlauben durch verschiedene Analysen des Zustandsraumes Korrektheitseigenschaften nachzuweisen. Insbesondere die Modellierung von Systemen mit Hilfe von Transitionssystemen und die Formu-

lierung gewünschter Eigenschaften mit Hilfe von Temporal–Logiken hat sich als vielversprechende Methode erwiesen. Dieser Ansatz wird auch in TRUTH verfolgt.

Die Spezifikation eines verteilten Systems erfolgt dabei üblicherweise in einer Spezifikationssprache. Solche haben gegenüber gewöhnlichen Programmiersprachen den Vorteil, daß sie eine Abstraktion des Systems auf die Merkmale zulassen, die für die Korrektheit wesentlich sind. Ziel ist dabei die Reduktion auf eine endliche Struktur. Es gibt eine Reihe von Spezifikationssprachen zu diesem Zweck, die sich für unterschiedliche Problemstellungen eignen, als Beispiel seien hier CCS [Mil89], CSP [Hoa83] und Promela [Hol91] genannt.

Es gibt verschiedene Logiken, die sich dazu eignen, Eigenschaften solcher endlich beschriebenen Systeme auszudrücken [Eme96]. Allen diesen Logiken ist gemein, daß ihr Modelchecking–Problem über Strukturen, wie man sie als Semantik der Spezifikationen erhält, entscheidbar ist. Daher ist eine automatische Verifikation der Spezifikationen möglich. Hierbei zieht eine höhere Aussagekraft der Logik in der Regel eine höhere Komplexität des Modelcheckings nach sich.

Es existieren bereits Werkzeuge, die eine Vorgehensweise, wie sie oben skizziert wurde, unterstützen [Hol91, CPS93, CS96]. Dennoch ist eine weitere Implementierung eines solchen Werkzeugs aus verschiedenen Gründen sinnvoll. Der wichtigste Grund hierfür ist die Größe der semantischen Strukturen, die sich aus Spezifikationen ergeben können. Alle existierenden Werkzeuge erzeugen aus der eingegebenen Spezifikation eine Repräsentation des Systems in Form eines endlichen Automaten. Dabei wird die nicht–synchronisierte Parallelität von Teilprozessen als nicht–deterministische Auswahl aller möglichen Ausführungsreihenfolge dieser Prozesse interpretiert, was zu einer Explosion der Größe des Zustandsraumes führt. Es sind daher andere semantische Bereiche vorgeschlagen worden, die dieses Problem durch eine andere Modellierung von Nebenläufigkeit reduzieren. Dazu gehören zum Beispiel Mazurkiewics–Traces, Petri–Netze oder Ereignisstrukturen [WN93]. Auf die Entwicklung von Werkzeugen zur Verifikation verteilter Systeme haben diese jedoch noch keine große Auswirkung gehabt. Dies liegt vor allem daran, daß bislang keine befriedigenden Modelchecking–Algorithmen für die über diesen Bereichen verwendeten Logiken existieren.

Daraus resultiert der Wunsch nach einem System, das als Versuchsplattform für neue Ideen in diesem Bereich dienen kann. Darüber hinaus ist eine Konfigurierbarkeit des Systems in vielerlei Hinsicht wünschenswert. So werden auch für Transitionssysteme immer noch neue Logiken und für diese auch immer neue Modelchecking–Algorithmen untersucht. Daneben ist auch die Wahl der verwendeten Spezifikationssprache stark von der Art des zu beschreibenden Systems abhängig.

In dieser Hinsicht versagen die bereits existierenden Werkzeuge. Oft sind sie in der Wahl des semantischen Bereichs und selbst der verwendeten Logik festgelegt. Hinzu kommt, daß die Architektur dieser Systeme es schwer macht, irgendwelche Änderungen, wie sie oben angedeutet wurden, mit vertretbarem Aufwand vorzunehmen. Andere Systeme

sind in dieser Hinsicht variabler gestaltet, doch die Wahl der Programmiersprache, in der sie implementiert wurden, erschwert eine Änderung durch Außenstehende unnötig. Von anderen ist der Quelltext erst gar nicht zu erhalten.

Daher ist die Entwicklung eines Werkzeugs, das als Grundlage für neue Untersuchungen dienen kann, ein lohnendes Unterfangen.

Diese Arbeit beschreibt die Architektur und die Implementierung von TRUTH, einem Werkzeug zur Verifikation verteilter Systeme, das im Hinblick auf die oben geforderten Variabilitäten entwickelt wurde. In seiner heutigen Version stellt TRUTH ein vollwertiges Werkzeug dar, das in seiner Leistung mit existierenden Implementierungen vergleichbar ist, ja diese stellenweise übertrifft. Dabei wurde beim Entwurf des Systems besonderer Wert auf dessen Veränderbarkeit gelegt, so daß eine Modifizierung von TRUTH zur Berücksichtigung und Erprobung neuer Techniken leicht möglich ist. Dies wird unterstützt durch die Wahl von `Haskell` als Implementierungssprache. Eine deklarative Programmiersprache hat große Vorteile im Hinblick auf Modifizierbarkeit und Wartbarkeit des Programmes.

Das erste Kapitel gibt eine Übersicht über die Architektur und Funktionalität von TRUTH. Im nächsten Kapitel findet sich eine Beschreibung der Programmiersprachen, Werkzeuge und Bibliotheken, die bei der Implementierung verwendet wurden. Die Informationen sind wesentlich für das Verständnis der Ausführungen über die Details der Implementierung. Das dritte Kapitel beschäftige sich mit dem Bereich der Spezifikation verteilter Systeme. Die Spezifikationssprache CCS wird eingeführt und ihre Semantik mit Hilfe von beschrifteten Transitionssystemen erklärt. Es wird eine Übersicht über die Implementierung dieser Komponenten gegeben, wobei der Gesichtspunkt der Modifizierbarkeit eine wichtige Rolle spielt. Kapitel 4 führt eine Reihe von Logiken ein, die sich eignen, Eigenschaften verteilter Systeme auszudrücken. Die wichtigste Rolle spielt hierbei der μ–Kalkül, für den ein Tableau–basierter Modelchecking–Algorithmus beschrieben wird. Die Implementierung dieses Algorithmus in TRUTH wird ebenfalls in diesem Kapitel beschrieben. Im fünften Kapitel wird ein Verfahren zur interaktiven Simulation von CCS–Spezifikationen angegeben, das zu einem besseren Verständnis der spezifizierten Systeme beitragen kann. Abschließend findet sich im sechsten Kapitel ein größeres Beispiel der Analyse und Verifikation eines verteilten Systems mit TRUTH.

1. Truth

Diese Kapitel gibt zunächst eine kurze Übersicht über die Architektur und die Funktionalität von TRUTH. Die späteren Kapitel befassen sich dann detailierter mit den einzelnen Komponenten des Systems.

TRUTH ist ein System zur Verifikation verteilter Systeme. Es erlaubt die Spezifikation eines verteilten Systems mit Hilfe einer Prozeßalgebra. Eigenschaften, die ein so spezifiziertes System erfüllen soll, lassen sich in einer aussagestarken Modallogik ausdrücken. Die wesentliche Aufgabe von TRUTH ist die Beantwortung der Frage: Erfüllt eine Spezifikation *Spec* die Eigenschaft Φ.

Als Prozeßalgebra fiel die Wahl dabei auf Milners CCS. Dies erlaubt einen einfachen Vergleich mit bestehenden Werkzeugen, da diese auch CCS als Spezifikationssprache unterstützen. Darüber hinaus existiert eine ganze Reihe von Spezifikationen in CCS, die es erlauben, TRUTH anhand realistischer Beispiele zu testen und zu bewerten. Die verwendete Logik ist der μ–Kalkül, eine sehr aussagestarke Logik, die die Formulierung vieler Eigenschaften verteilter Systeme ermöglicht. In Kapitel 4 findet sich eine Beschreibug einger Eigenschaften, welche im μ–Kalkül formulierbar sind.

Anfragen der Form „Erfüllt *Spec* die Eigenschaft Φ“ werden mit Hilfe eines Tableau–basierten Modelcheckers für den μ–Kalkül bestimmt.

Neben dieser Funktionalität verfügt TRUTH noch über einige Funktionen, welche eine zusätzliche Analyse von verteilten Systemen durchführen. Neben Funktionen, die es erlauben, verschiedene Ableitungen von Prozeßspezifikationen zu generieren, ist hierbei vor allem die visuelle Simulation hervorzuheben, die zu einem besseren Verständnis der betrachteten Systeme beitragen kann.

Der größte Teil von TRUTH ist in `Haskell` implementiert worden. Der deklarative Charakter dieser Programmiersprache sowie der völlige Verzicht auf Seiteneffekte führt zu einer Implementierung, die sehr gut dazu geeignet ist, Veränderungen und Erweiterungen an ihr vorzunehmen. Dies war eine der wichtigsten Anforderungen an die Implementierung von TRUTH. Neben der Wahl der Programmiersprache sind viele Designentscheidungen im Hinblick auf die Variabilität von TRUTH getroffen worden.

Dieses Kapitel gibt eine Übersicht über die Funktionalität und den Aufbau des Systems. Dabei wird insbesondere auf die Aspekte eingegangen, die zu der Variabilität von TRUTH beitragen.

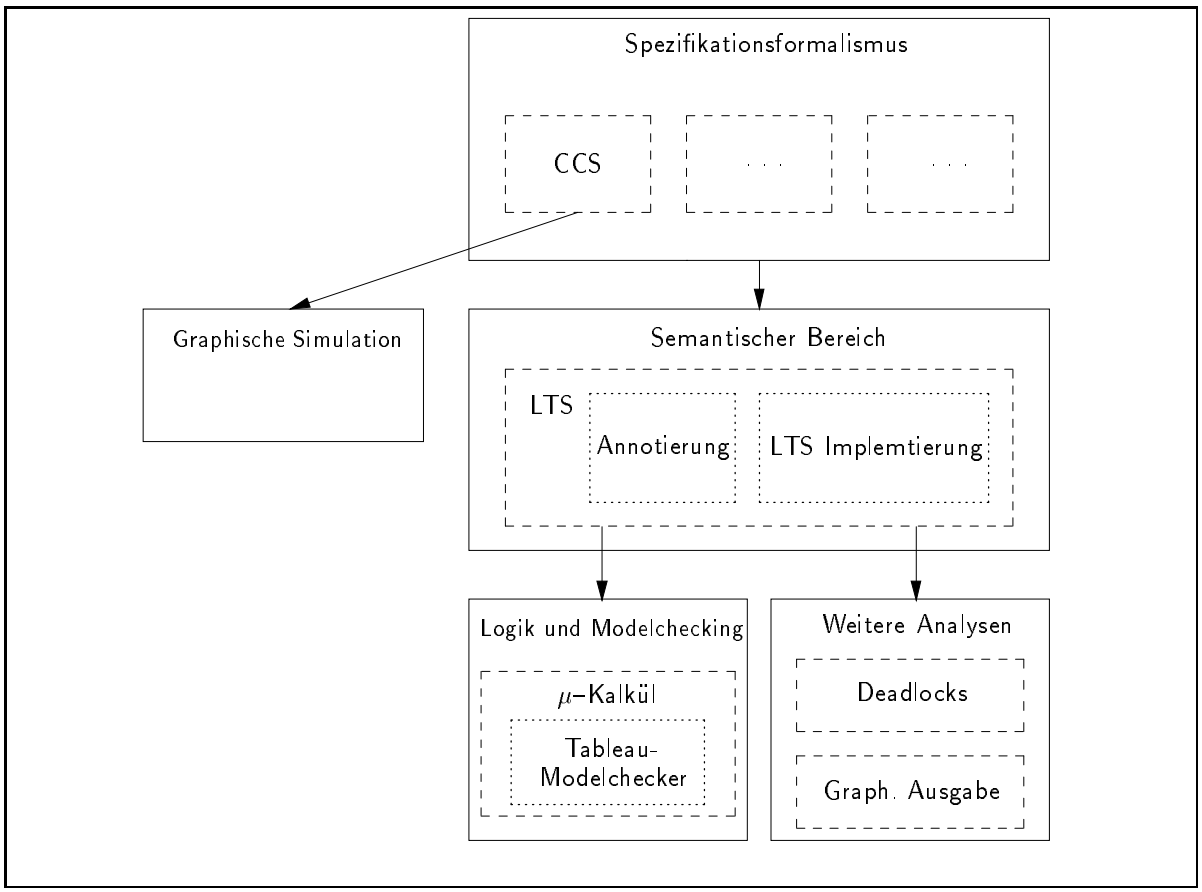


Abbildung 1.1: Architektur von TRUTH

Eine Übersicht über die Komponenten[1], aus denen TRUTH zusammengesetzt ist, findet sich in Abbildung 1.1.

1.1. Benutzerschnittstelle

TRUTH präsentiert sich dem Benutzer nach dem Start mit einer einfachen Textschnittstelle, deren Bedienung sich an die der unter UNIX verbreiteten C–Shell anlehnt. Der Benutzer kann Dateien mit Spezifikationen und Formel einlesen, kann Definitionen ausgeben, ändern und auch wieder in Dateien speichern. Auch die Umleitung der Ausgabe in eine Datei ist möglich, was zum Beispiel dazu benutzt werden kann, um TRUTH in einer Art Batch–Betrieb zu verwenden.

Über eine ganze Reihe von Befehlen kann der Benutzer die eingegebenen Spezifikationen untersuchen. Zum Beispiel ist es möglich, alle Nachfolge–Konfigurationen eines Ausdrucks zu ermitteln. Es sind aber auch Anfragen möglich, die sich auf das gesamte, durch eine Spezifikation induzierte, Transitionssystem beziehen. So ist es möglich,

[1] Wir wählen hier absichtlich nicht den Begriff *Modul*, da es sich bei diesen Komponenten jeweils um eine Reihe von `Haskell`–Modulen handelt, die jedoch logisch eng zusammengehören.

Deadlocks zu suchen, die Größe des Transitionssystems zu bestimmen oder das Transitionssystem graphisch darzustellen. Auch der Aufruf des Modelcheckers sowie der Simulation geschieht aus dieser Ebene heraus.

1.2. CCS und Prozeßalgebra

TRUTH ist in der Lage, Prozeßspezifikationen in Form von CCS–Ausdrücken zu verarbeiten. CCS wurde von Milner zur Beschreibung nebenläufiger, kommunizierender Systeme entwickelt [Mil89]. Alle für die Verarbeitung von CCS benötigten Funktionen sind in der CCS–Komponente zusammen gefaßt. Diese beinhaltet einen Scanner und Parser zum Einlesen von Definitionen des Benutzers. Eingegebene Definitionen werden in einer Umgebung gespeichert, die abgefragt und geändert werden kann.

Den wichtigsten Teil der CCS–Komponente stellen die Funktionen dar, die es ermöglichen, Einzelschritt–Ableitungen von CCS–Ausdrücken, gemäß der in [Mil89] beschriebenen SOS–Regeln für die Übergangsrelation, zu bestimmen. SOS–Regeln wurden von Plotkin als Notation für operationelle Semantiken von Programmformalismen entwickelt [Plo81] und ermöglichen eine sehr übersichtliche Notation.

Die Anforderungen an eine Spezifikationssprache werden sich immer nach dem zu spezifizierenden System richten. Aus diesem Grunde existiert heute eine Reihe von Sprachen, die sich für verschiedene Probleme eignen. Dazu gehören verschiedene Varianten von CCS genauso wie zum Beispiel CSP [Hoa83] oder Promela [Hol91] – Spezifikationssprachen, die sich stärker an existierenden Programmiersprachen orientieren.

Allen diesen Spezifikationssprachen ist jedoch gemein, daß sie über eine Transitionssystem–Semantik auf der Basis von Einzelschritt–Transitionen verfügen. Diese lassen sich zudem oft in der Form von SOS–Regeln angeben.

Die Wahl von CCS als verwendete Spezifikationssprache ist willkürlich und nur durch persönliche Vorlieben begründet. Sie ist darüber hinaus von geringer Relevanz für die Architektur von TRUTH. Dies findet darin Niederschlag, daß zwischen der CCS–Komponente und den meisten anderen Komponenten von TRUTH eine zusätzliche Abstraktionsebene existiert. Dies erlaubt es, CCS auf einfache Art gegen eine andere Spezifikationsprache auszutauschen. Voraussetzung hierfür ist, daß die gewählte Spezifikationssprache über eine Semantik auf der Basis von Transitionssystemen verfügt.

Es existieren zur Zeit bereits Bestrebungen, diesen Austausch zu automatisieren. Vorbild für dieses Vorgehen ist der Prozeßalgebra Compiler der NCSU Concurrency Workbench [CMS95]. Dieser ermöglicht es, aus einer Spezifikation der Syntax und Semantik einer Prozeßalgebra, automatisch die Routinen zu generieren, die die NCSU CWB benötigt, um Spezifikationen in dieser Prozeßalgebra zu verarbeiten.

1.3. Semantische Bereiche

Die meisten Analysen einschließlich des Modelcheckings operieren nicht direkt auf den Spezifikationen, sondern auf semantischen Strukturen, die durch diese Spezifikationen induziert werden. Die einfachste und am besten verstandene Möglichkeit für solche Strukturen bilden beschriftete Transitionssysteme. Diese stellen in TRUTH den semantischen Bereich dar, der bei allen Analysen zugrundegelegt wird.

Wie in Kapitel 3 näher erläutert wird, ist die Wahl von beschrifteten Transitionssystemen als semantischen Bereich sehr problematisch. Dies liegt insbesondere in der Tatsache begründet, daß die aus Spezifikationen resultierenden Transitionssysteme im allgemeinen sehr groß werden können. Es sind daher verschiedene anderere semantische Bereiche vorgeschlagen worden, bei denen dieses Problem nicht in diesem Maße auftritt. Trotzdem finden diese Strukturen bei der Implementierung von Verifikationsplattformen zur Zeit noch keine Anwendung:

- Nur für wenige semantische Bereiche existieren Spezifikationssprachen, die eine entsprechende Semantik besitzen. Es gibt zwar Ausnahmen – so existiert zum Beispiel eine Semantik für CCS–Spezifikationen auf der Basis von Ereignistrukturen – doch sind die resultierenden Strukturen für Anwendungen im Modelchecking nicht gut geeignet.

- Es gibt nur wenige Logiken, die über diesen Strukturen interpretiert werden. Diese sind zum Teil nur sehr schlecht verstanden, insbesondere existieren kaum Modelchecking Algorithmen. Es ist allerdings damit zu rechnen, daß sich dies in näherer Zukunft ändern wird, da auf diesem Gebiet zur Zeit intensiv geforscht wird.

Die verwendeten Logiken unterscheiden sich im Aufbau nur unwesentlich von den Logiken, wie sie über Transitionssystemen gebräuchlich sind. Die Vorgehensweise beim Modelchecking wird sich jedoch wesentlich von der hier implementierten unterscheiden. Auch die Generierung der semantischen Struktur aus der Spezifikation wird ganz anders vorgenommen werden, als dies bei Transitionssystemen der Fall ist. Daher war es nicht möglich, den semantischen Bereich in TRUTH ähnlich variabel zu gestalten, wie dies zum Beispiel mit der verwendeten Spezifikationssprache möglich war. Selbstverständlich ist jedoch eine Implementierung eines neuen semantischen Bereichs auf der Basis von TRUTH möglich.

1.4. Transitionssysteme

Die Implementierung beschrifteter Transitionssysteme bildet gewissermaßen den Kern der Implementierung von TRUTH. Gestützt auf die vom Prozeßalgebra–Modul bereitgestellte Berechnung eines Schritts der Transitionsrelation, ermöglicht diese Komponente

die bedarfsgesteuerte, effiziente und speichersparende Generierung des gesamten Transitionssystems.
Geschwindigkeit und Speicherverbrauch spielten bei der Entwicklung dieses Systems eine entscheidende Rolle, da es sich hier um die zentralen Routinen aller Analysen handelt. Darüber hinaus haben die sich aus der Spezifikation realistischer Systeme ergebenden Transitionssysteme die Eigenschaft, sehr groß zu werden. Je nach betrachtetem System und verwendeter Spezifikationssprache sind Systeme von mehreren Tausend bis hin zu 2^{30} und mehr Zuständen möglich. Während Transitionssysteme mit letzterer Größe die Größenordnung dessen sprengen, was TRUTH zu verarbeiten in der Lage ist, ist die Arbeit mit Transitionssystemen von mehreren 10.000 Zuständen kein Problem.
Über die Generierung und den Zugriff auf Transitionssysteme hinaus unterstützt die vorliegende Implementierung auch eine beliebige Annotation der Transitionssysteme, wie sie für verschiedene Analysen, insbesondere für den implementierten Modelchecking–Algorithmus, notwendig sind.
Direkt aufbauend auf die Implementierung der Transitionssysteme sind einige einfache Analysen realisiert. So ist es möglich, Deadlocks zu suchen, sich Pfade zu allen Zuständen ausgeben zu lassen oder die Größe von Transitionssystemen zu ermitteln. Auch eine graphische Ausgabe der Transitionssysteme mit Hilfe von *daVinci*, einem Werkzeug zur Visualisierung gerichteter Graphen, ist möglich.
Bei der Implementierung der Transitionssysteme ist es sehr wünschenswert, mit verschiedenen Ansätzen experimentieren zu können. Während die vorliegende Implementierung das Transitionssystem konventionell in Form einer Datenstruktur ablegt, wie diese zum Beispiel in [KL95] vorgeschlagen wird, gibt es auch andere Ansätze. Insbesondere ist hier die Speicherung von Transitionssystemen in Form sogenannter BDDs erwähnenswert [Bry86]. Dieses Vorgehen macht es möglich, wesentlich größere Transitionssysteme zu speichern, als dies mit der jetzigen Methode möglich wäre [BCL91].
Eine Implementierung einer BDD–Bibliothek in `Haskell` erscheint wenig sinnvoll, da BDDs bei ihrer Speicherung massiven Gebrauch von der expliziten Repräsentierung im Speicher machen und daher eine explizite Speicherverwaltung notwendig ist. Es existiert aber bereits eine Vielzahl von BDD–Implementierungen in C und C++ [Sto95]. Da der Glasgow Haskell Compiler mit `Greencard` über ein Schnittstelle zur Anbindung an C–Programme besitzt [JNR97], besteht prinzipiell die Möglichkeit zur Implementierung eines solchen Verfahrens.
Zusätzlich wird eine Veränderung von TRUTH in dieser Hinsicht unterstützt durch die Tatsache, daß die Datenstruktur zur Speicherung der beschrifteten Transitionssysteme als abstrakte Datenstruktur implementiert und in einer Zustandsmonade gekapselt ist.

1.5. Logik und Modelchecking

Sicherlich die stärkste Form der Analyse erlaubt der eingebaute Modelchecker, der Formeln des μ–Kalküls auf ihre Gültigkeit über Transitionssystemen, die sich als Semantik

von Prozeßspezikifationen ergeben, überprüft.

Der μ–Kalkül [Koz82] ist eine sehr expressive Modallogik, welche es erlaubt, viele interessante Eigenschaften von verteilten Systemen auszudrücken. Durch die Ausdruckskraft des μ–Kalküls lassen sich so beliebige *Safety*– *Liveness*– und *Fairness*–Eigenschaften überprüfen. Näheres hierzu findet sich in Kapitel 4.

Ähnlich wie dies bei der Spezifikationssprache möglich ist, könnte man auch die verwendete Logik leicht durch eine andere Logik ersetzen, solange diese eine Interpretation über beschrifteten Transitionssystemen besitzt. Es sind dennoch keine Vorrichtungen im Design von TRUTH vorgesehen, die den Austausch der benutzten Logik unterstützen. Der Grund hierfür liegt darin, daß mit dem μ–Kalkül bereits gewissermaßen die ausdruckstärkste der vorgeschlagenen Logiken für die Spezifizierung von Eigenschaften verteilter Systeme implementiert wurde. Darüber hinaus existieren für die anderen gebräuchlichen Logiken, wie LTL, CTL oder CTL*, einfache Übersetzungen in äquivalente Formeln des μ–Kalküls [Eme97], so daß ein System, welches den μ–Kalkül unterstützt, damit auch alle anderen Logiken unterstützt. Hilfreich ist hierbei die Möglichkeit, Formeln mit Hilfe von Makros zu definieren, die vor dem Modelchecking in Formeln des reinen μ–Kalküls übersetzt werden. Dies ist in der Benutzerschnittstelle von TRUTH vorgesehen.

Interessant ist die Betrachtung anderer Logiken jedoch nicht nur unter dem Gesichtspunkt der Expressivität. Auch Effizienz–Betrachtungen müssen eine Rolle spielen, da die Modelchecking–Probleme für diese Logiken unterschiedliche Komplexitäten besitzen. Jedoch ist eine auf dem μ–Kalkül basierende Lösung auch in dieser Hinsicht befriedigend, da die Übersetzungen von den oben genannten Logiken nicht die gesamte (syntaktische) Expressivität des μ–Kalküls nutzen. Für die Fragmente des μ–Kalküls, in denen die Formeln liegen, die man durch die Übersetzungen erhält, sind Modelchecking–Algorithmen bekannt, deren Komplexität derer für die Ausgangslogiken entspricht [CS92, GPVW95, BC96a].

Diese Ausführungen zeigen jedoch auch, daß die Verwendung verschiedener Modelchecking–Algorithmen durchaus sinnvoll sein kann. Aus diesem Grund wird die Einbindung zusätzlicher Modelchecking–Algorithmen in TRUTH besonders unterstützt. Dies geschieht insbesondere durch Funktionen, die eine einfache Generierung des Transitionssystems erlauben. Je nach Anforderung des Algorithmus können so mit einem einzigen Befehl das gesamte Transitionssystem oder auch nur die Nachfolger eines Knotens generiert werden. Darüber hinaus wird eine Annotierung der Knoten des Transitionssystems durch Werte beliebigen Typs unterstützt.

Der zur Zeit implementierte Modelchecking–Algorithmus ist in der Lage, beliebige Formeln des μ–Kalküls zu verarbeiten. Es handelt sich dabei um einen Tableau–basierten Algorithmus gemäß Cleaveland [Cle90]. Dieser Algorithmus erlaubt es, das Transitionssystem nur bedarfsgesteuert zu generieren und erreicht damit oft eine Verbesserung in der Laufzeit gegenüber Algorithmen, die stets das gesamte Transitionssystem betrachten.

1.6. Simulation

TRUTH erlaubt die visuelle Simulation von CCS–Spezifikationen. Ausgehend von einer graphischen Darstellung einer Konfiguration des Systems kann der Benutzer interaktiv den Lauf des Systems beeinflussen. Auf diese Weise ist ein besseres Verständnis des Systems möglich. Dies ist insbesondere dann hilfreich, wenn der Modelchecker eine Eigenschaft als nicht erfüllt nachweist. In diesem Fall kann eine Simulation des Systems helfen, den Grund für die Fehlerhaftigkeit des Systems herauszufinden. Es ist darüber hinaus denkbar, diese Simulationskomponenten weiterzuentwickeln, um Gegenbeispiele zu visualisieren. Diese könnten vom Modelchecker bei Nichterfüllung einer Eigenschaft automatisch generiert werden. Bei der Simulation wird der Benutzer durch eine Reihe von Informationen im Verständnis des Systems unterstützt. Darüber hinaus erlaubt die Benutzung einer graphischen Benutzerschnittstelle eine Darstellung des Verlaufs der Simulation, was zu einer besseren Übersicht über das Verhalten des Systems verhilft.

Die Simulation basiert direkt auf der CCS–Komponente, genauer, auf deren Funktion, Einzelschritte gemäß der Einzelschritt–Semantik zu berechnen. Dies ist zwar generell eine Funktion, wie sie jede gebräuchliche Prozeßalgebra bereitstellt. Darüber hinaus spielt jedoch CCS bei der Aufteilung des Systems in separat zu betrachtende Einheiten ein große Rolle, so daß eine Unabhängigkeit der Simulation von CCS nicht zu realisieren ist.

Dennoch existiert eine starke Abstraktionsbarriere zwischen der Steuerung der Simulation und der graphischen Darstellung. Diese mußte eingeführt werden, da die Implementierung der graphischen Benutzerschnittstelle in `Haskell` nicht sinnvoll erschien. Aus diesem Grunde erfolgte die Programmierung dieser Komponente in `Java`. Diese wird dann über eine Textschnittstelle an TRUTH angebunden.

Da diese Textschnittstelle genau definiert und auch in dieser Arbeit dokumentiert ist, ist eine Neuimplementierung der Benutzeroberfläche in einer effizienteren Programmiersprache möglich. Aber auch die Anbindung der `Java`–Oberfläche über das WWW ist eine denkbare Erweiterung.

2. Implementierungsumgebung

Ziel dieser Diplomarbeit war die Erstellung eines Prototyps einer Verifikationsplattform unter der besonderen Berücksichtigung der eingangs beschriebenen Kriterien. Das dabei entstandene System TRUTH stellt jedoch deutlich mehr als einen Prototyp dar und kann sich in der Leistung mit bestehenden Werkzeugen durchaus messen. Dies wurde möglich durch die Verwendung einer leistungsfähigen Programmiersprache sowie durch Zurückgreifen auf eine ganze Reihe von Werkzeugen und Bibliotheken.

Da diese eine große Rolle bei der Implementierung von TRUTH gespielt haben, sollen sie in diesem Kapitel vorgestellt werden. Darüber hinaus wird erläutert, warum die Wahl auf das jeweilige Werkzeug fiel.

2.1. Haskell

Der weitaus größte Teil von TRUTH ist in `Haskell` geschrieben. Es handelt sich bei `Haskell` um eine nicht–strikte, rein–funktionale Programmiersprache [PH+96].

Neben den Vorteilen, die sich ohnehin aus der Verwendung einer deklarativen Programmiersprache im Hinblick auf Lesbarkeit und Wartbarkeit der Programme ergeben [Hug89], verfügt `Haskell` über weitere Eigenschaften, die sich positiv beim Design und der Implementierung von TRUTH ausgewirkt haben. Diese werden nun im einzelnen vorgestellt.

2.1.1. Typsystem

`Haskell` ist eine stark getypte Sprache, das heißt, daß jedem Ausdruck stets ein Typ zugeordnet werden kann. Während des Compilierens überprüft ein Typchecker das Programm auf Typkorrektheit. Dadurch können manche Programmierfehler, die sonst erst zur Laufzeit bemerkt würden, schon zur Compilezeit entdeckt werden. Dies beschleunigt die Programmentwicklung und mach das erstelle Programm sicherer.

Darüber hinaus verfügt das Typsystem von `Haskell` über sogenannte Typklassen. Damit ist es möglich, Überladung von Funktionen elegant auszudrücken und in das Typsystem zu integrieren. Wie der Ausdruck *Typklasse* bereits suggeriert, wird außerdem

eine gewisse Form der objektorientierten Programmierung unterstützt, was zum Beispiel dazu benutzt wurde, beliebige Annotationen von Transitionssystemen zu ermöglichen.

2.1.2. Algebraische Datentypen und Pattern Matching

`Haskell` unterstützt die Definition algebraischer Datentypen. Diese eignen sich besonders gut zur Repräsentierung von Termstrukturen, wie sie in TRUTH an einer Reihe von Stellen auftreten. Zusammen mit der Möglichkeit des Pattern Matchings ist es möglich, Algorithmen, die auf diesen Datenstrukturen operieren, in klarer und verständlicher Weise auszudrücken.

Weiterhin kann man algebraische Datentypen zur Realisierung abstrakter Datentypen nutzen. Dies ist eine übliche Technik und wird verwendet, um eine Modifizierung von TRUTH zu erleichtern.

2.1.3. Monadische Ein– und Ausgabe

In rein–funktionalen Programmiersprachen ist die Behandlung von Ein– und Ausgabeoperationen problematisch, da diese stets einen Seiteneffekt darstellen. Mit dem Konzept der monadischen Ein–/Ausgabe ist es gelungen, dieses Problem auf elegante Weise zu lösen [PW93]. `Haskell` verwendet dieses Konzept. Integriert in die Ein–/Ausgabe–Bibliothek ist die Fehlerbehandlung mit Hilfe von Exceptions und umfangreiche Funktionen zur Manipulation von Dateien.

2.1.4. Standard–Bibliotheken

Im `Haskell`–Standard ist eine Vielzahl von Funktionen festgelegt, die in jeder `Haskell`–Implementierung vorhanden sein müssen. Neben der Unterstützung einer ganzen Reihe von elementaren Datentypen sind auch verschiedene Funktionen höherer Ordnung vorhanden, die gebräuchliche Abstraktionen, wie die Anwendung einer Funktion auf eine Liste von Elementen, ausdrücken

Darüber hinaus definiert der `Haskell` Library Report weitere Bibliotheken, die ein vollwertiges `Haskell`–System unterstützen muß. In diesen werden zum Beispiel Arrays definiert. Darüber hinaus existiert eine Vielzahl von Funktionen zur Manipulation von Listen, die sich an vielen Stellen als nützlich erweisen. Wir machen auch Gebrauch von der an dieser Stelle definierten Zufallzahlen–Bibliothek.

2.2. Der Glasgow Haskell Compiler

Der Glasgow Haskell Compiler [JHH+93], kurz `GHC`, ist ein optimierender Compiler für `Haskell`, der Maschinencode für viele verschiedene Hardware–Plattformen erzeugen

kann. Diese Vielseitigkeit wird durch die Verwendung von GNU C als Zwischensprache bei der Codeerzeugung erreicht. Dadurch ist es auch möglich, die Codeoptimierungs–Funktionen des GNU C Compilers zu verwenden. Darüber hinaus führt der `GHC` selbst eine ganze Reihe von Optimierungen durch, so daß die übersetzten Programme sehr gute Laufzeit–Eigenschaften haben.

Über die im `Haskell`–Standard geforderten Eigenschaften bietet der `GHC` einige Erweiterungen und zusätzliche Bibliotheken.

2.2.1. Zustands–Monaden

Rein–funktionale Sprachen sind in der Lage, viele Algorithmen sehr prägnant und effizient auszudrücken. Es gibt jedoch auch Fälle, in denen dies nicht der Fall ist. Insbesondere Graph–Algorithmen, bei denen ein explizites Sharing[1] ausgenutzt wird, sind in rein–funktionalen Programmiersprachen nur ineffizient zu implementieren. Gleiches gilt zum Beispiel für Hash–Tabellen. Beide treten in der Implementierung von TRUTH auf.

Durch die Einführung sogenannter Zustands–Monaden [LJ94] ist es jedoch möglich, destruktive Veränderungen an Datenstrukturen im Heap sicher, das heißt, ohne Verlust der referentiellen Transparenz, vorzunehmen. Dabei werden die Algorithmen in Form von Zustandstransformationen aufgebaut, die durch den Aufruf einer speziellen Funktion auf einen gekapselten Zustandsraum angewandt werden.

Zustands–Monaden sind im `GHC` implementiert und finden in TRUTH Anwendung sowohl bei der Speicherung der Transitionssysteme als auch bei der Implementierung von Hash–Tabellen.

2.2.2. Unboxed Werte

Die bei nicht–strikten Programmiersprachen verwendete Speicherverwaltung führt zu Overheads bei der Implementierung numerischer Algorithmen. Dadurch, daß Werte nicht direkt, sondern in Form sogenannter Boxen auf dem Heap abgelegt werden, verbringen Programme einen großen Teil ihrer Laufzeit mit dem Ein– und Auspacken solcher Boxen. Die Auswertung eines Ausdrucks der Form `x = a * (b + c)` geschieht bei einer naiven Implementierung, indem zunächst die Boxen von `b` und `c` ausgepackt werden, um die entsprechenden Werte zu erhalten. Diese werden addiert und das Ergebnis in einer neuen Box auf dem Heap abgelegt. Zur Auswertung des Produktes muß diese Box dann erneut ausgepackt werden, was zu einem unnötigen Aufwand führt.

Compiler für nicht–strikte funktionale Programmiersprachen führen aus diesem Grunde Optimierungen durch, um wiederholtes Ein– und Auspacken zu verhindern. Im `GHC`

[1] Unter Sharing versteht man die Darstellung identischer Teilstrukturen von Datenstrukturen durch einen einzigen Eintrag im Heap.

geschieht dies, indem die Boxen und deren Inhalt explizit in der Sprache sichtbar gemacht werden. Dazu wird `Haskell` um sogenannte *unboxed* Werte erweitert [JL91]. Diese werden im Laufzeit–System nicht mit der Hilfe von Boxen sondern direkt im Speicher realisiert. Aus diesen unboxed Werten werden boxed Werte, indem ihnen eine Datenkonstruktor vorangestellt wird. Diese Sichtweise ermöglicht eine elegante Formulierung und Implementierung der Optimierungen, wie sie oben angedeutet wurden.

Für die Implementierung von TRUTH spielen diese unboxed Werte eine wichtige Rolle, da sie als sogenannte „first class citizens“ im `GHC` auftreten, das heißt, an jeder beliebigen Stelle in einem Programm auftreten dürfen.[2]

So ist zum Beispiel eine Typklasse `Hashable` unter der Verwendung von unboxed Typen folgendermaßen definiert:

```
class (Eq a) => Hashable a where
  hashVal::a->Int#
```

`Int#` ist dabei die unboxed Variante von `Int`. Auf diese Weise ist es möglich, Typen, deren Komponenten Instanzen von `Hashable` sind, auf effiziente Art auch zu Instanzen von `Hashable` zu machen:

```
instance (Hashable a, Hashable b) => Hashable (a,b) where
  hashVal (a,b) = (hashVal a) +# 2# *# (hashVal b)
```

2.2.3. Bibliotheken

Zu den üblichen Bibliotheken beinhaltet die `GHC`–Distribution eine Reihe weiterer, die sich bei der Implementierung von TRUTH als sehr nützlich erwiesen haben.

Posix

Zur Einbindung externer Programme ist ein Zugriff auf die Prozeß–Primitiven von UNIX notwendig. Der `GHC` unterstützt dies durch die Bereitstellung von Funktionen gemäß dem POSIX–Standard. Dadurch ist es möglich, Prozesse abzuspalten, fremde Programme aufzurufen und mit diesen über Pipes zu kommunizieren.

Daneben wird die Einrichtung von Signal Handlern unterstützt, die es erlauben, eine Berechnung von TRUTH jederzeit unterbrechen zu können, ohne daß dies zu einem Verlust der gespeicherten Spezifikationen und Formeln führen würde.

[2] Einzige Ausnahmen hierbei: Unboxed Typen dürfen weder in Tupeln noch als Instanz von Typklassen auftreten.

Mengen und Abbildungen

Das Modul `Set` implementiert einen abstrakten Datentyp zur Speicherung endlicher Mengen. Die Elemente werden dabei in Form balancierter Bäume abgelegt [Ada93]. Dazu ist es notwendig, daß der Typ der Elemente eine Instanz von `Ord` ist, das heißt, daß eine lineare Ordnung auf den Elementen existiert. Durch die Speicherung in Baumform ist ein Zugriff in logarithmischer statt linearer Zeit möglich.

Basierend auf dieser Mengenimplementierung ist der abstrakte Datentyp `FiniteMap` implementiert. Dieser erlaubt die effiziente Speicherung von Assoziationen. Dabei muß der Typ der Schlüssel wiederum Instanz von `Ord` sein.

Pretty Printer

Ein interaktives Programm wie TRUTH muß interne Datenstrukturen, wie zum Beispiel Prozeßterme oder Formeln, übersichtlich darstellen können. Zusätzlich benötigt man zur Anbindung externer Programme über Textschnittstellen die Möglichkeit, Daten strukturiert auszugeben. Der `GHC` verfügt über eine Pretty–Printing Bibliothek [Hug95], die dies unterstützt. Dazu können einige Basistypen wie Strings oder Integer–Werte in Werte vom Typ `Pretty` umgewandelt werden. Für Werte dieses Typs existieren dann eine ganze Reihe von Kombinatoren, die Strukturierungen wie zum Beispiel die horizontale oder vertikale Anordnung oder auch Einrückungen beschreiben. Schließlich existiert eine Funktion, die `Pretty`–Werte wieder in Strings umwandelt, welche bei ihrer Ausgabe das beschriebene Layout erzeugen.

Nach dem Vorbild der Implementierung des `GHC` ist darüber hinaus eine Typklasse `Outputable` zur Unterstützung der Ausgabe von Werten in TRUTH wie folgt definiert:

```
class Outputable a where
  ppr:: PprStyle -> a -> Pretty

data PprStyle
  = PprAll
  | PprDebug
  | PprUser
  | PprDaVinci
  deriving Eq
```

Jeder Typ, dessen Werte ausgegeben werden sollen, implementiert diese Klasse und kann so auf einfache Weise in Bildschirmausgaben dargestellt werden. Über einen zusätzlichen Parameter ist dabei die Art der Ausgabe zu beeinflussen.

2.2.4. Eindeutige Bezeichner

Namen spielen in TRUTH, insbesondere bei der Verarbeitung von Prozeßtermen, eine große Rolle. Bei den Namen von Aktionen ist es bei der Generierung von Transitionssystemen notwendig, diese schnell vergleichen zu können, da dieser Test sehr häufig auftritt. Ein Vergleich der Aktionen auf Basis ihrer Namen ist ineffizient, da es sich dabei oft um lange Zeichenketten handelt, die untersucht werden müssen. Daher bietet sich folgendes Verfahren an:

Beim Einlesen der Prozeßdefinition wird eine Symboltabelle der auftretenden Zeichenkette angelegt. In dieser wird für jeden Eintrag ein eindeutiger Wert abgelegt. Dieser bildet zusammen mit der Zeichenkette dann den Namen, der im Lauf des Programms verwendet wird. Wenn ein solcher Name ausgegeben wird, greift man auf die Zeichenkette zurück; geht es darum, Namen zu vergleichen, benutzt man den eindeutig vergebenen Wert.

Ein Ausschnitt aus den Modul `Id`, welches die Bezeichner bereitstellt, die für Prozeßnamen verwendet werden, sieht wie folgt aus:

```
data Id = MkId Unique
               PackedString
instance Eq Id where
  (==) (MkId uniq1 _) (MkId uniq2 _)  = uniq1 == uniq2
```

Der eindeutige Wert hat hierbei den Typ `Unique`. Die Implementierung von `Unique` ist der Implementierung des `GHC` entnommen. Dazu gehört das Modul `UniqSupply`, welches die Vergabe der Werte organisiert. `Unique` ist dabei ein abstrakter Datentyp, der einen Wert vom Typ `Int#` kapselt. Die Generierung der Werte geschieht mit Hilfe von `UniqSupply`s.

Ein `UniqSupply` kann dabei entweder in einen `Unique` umgewandelt, oder in zwei `UniqSupply`s aufgespalten werden. Die Art der Aufspaltung garantiert, daß die `Unique`s, die aus den resultierenden `UniqSupply`s erzeugt werden, stets unterschiedlich sind.

2.2.5. Weitere Eigenschaften

Der `GHC` verfügt über verschiedene Funktionalitäten und Erweiterungen, die ihn für die Implementierung von TRUTH besonders interessant erscheinen lassen.

Profiling

Zum `GHC` gehört seit der Version 0.15 ein Profiler [SJ95]. Dieser erlaubt eine detaillierte Analyse des Laufzeitverhaltens von `Haskell`–Programmen. Insbesondere werden Informationen über den Zeit– und Speicherplatzverbrauch einzelner Funktionen generiert. Dadurch war es möglich einen wichtigen Aspekt der Implementierung von TRUTH um den Faktor zwei zu beschleunigen.

Anbindung anderer Programmiersprachen

Mit `Greencard` existiert für den `GHC` eine elegante Möglichkeit, C–Code in `Haskell`–Programme einzubinden [JNR97]. Durch die Beschreibung von Schnittstellen durch sogenannte *Data Interface Schemes* ist auch der Austausch komplexer Datenstrukturen möglich. Auf diese Weise kann man bestehende C–Bibliotheken in TRUTH einbinden. Darüber hinaus wir zur Zeit auch noch an anderen Wegen gearbeitet, externe Funktionen in `Haskell` einzubinden.

Parallel Haskell

Es existieren Ansätze zur parallelen Auswertung von `Haskell`–Programmen. Unter Verwendung der Parallel Virtual Machine ist es damit möglich, Programme zum Beispiel auch auf mehreren Workstation parallel auszuwerten. Dies kann benutzt werden um zum Beispiel mit parallelisierbaren Modelchecking–Algorithmen zu experimentieren.

2.3. Happy

Zum Einlesen von Prozeßtermen und Formeln ist ein Parser notwendig. Es gibt eine Reihe von Ansätzen, solche in `Haskell` zu realisieren. Erste Versuche, die Parser in TRUTH mit der Hilfe von Parser–Kombinatoren [HM96] zu realisieren, scheiterten, da die zur Verfügung stehenden Bibliotheken nicht in der Lage waren, mehrdeutige Grammatiken mit Präzedenzen zu verarbeiten. Aus diesem Grund sind die in TRUTH verwendeten Parser mit `Happy` realisiert.

`Happy` [Mar97] stellt für `Haskell` das dar, was YACC für C ist. Aus einer attributierten BNF–Spezifikation einer Grammatik wird eine Funktion generiert, die Eingaben gemäß dieser Grammatik parst und dabei die Attributierung berechnet. Dabei ist `Happy` wesentlich vielseitiger als YACC, da zum Beispiel die Attribute einen beliebigen Typ haben dürfen. Dadurch ist es zum Beispiel möglich, auch eine eingeschränkte Form inheriter Attribute zu verwenden. Dies wird möglich, indem man als synthetische Attribute Funktionen verwendet, die als Parameter die inheriten Attribute erwarten.

Darüber hinaus ist es auch möglich, mehrere `Happy`–Parser gleichzeitig in einem System zu verwenden, was mit YACC–generierten Parsern nicht möglich ist.

2.4. DaVinci

TRUTH ermöglicht die graphische Darstellung beschrifteter Transitionssysteme, welche sich als Semantik von Prozeßspezifikationen ergeben. Die übersichtliche Darstellung von Graphen ist ein schwieriges Problem. Es existieren jedoch einige Werkzeuge, die

es ermöglichen, aus der Beschreibung der Struktur eines Graphen übersichtliche Ausgaben zu generieren. Ein solches Werkzeug ist *daVinci* [FW94]. Ausgehend von einer Termdarstellung des darzustellenden Graphen berechnet *daVinci* ein möglichst kreuzungsfreies Layout der Knoten und Kanten. Da hierzu ein heuristischer Algorithmus benutzt wird, sind die Ausgaben sicherlich nicht immer optimal. Bei kleineren Transitionssystemen kann diese Ausgabe jedoch sehr zum Verständnis des spezifizierten Systems beitragen. Leider ist die Leistungsfähigkeit von *daVinci* in Bezug auf die Größe der darstellbaren Graphen sehr beschränkt.

daVinci verfügt über die Möglichkeit, auch als Eingabe–Schnittstelle für TRUTH zu fungieren. Auf diese Weise wäre es zum Beispiel möglich, *daVinci* zu benutzen, um graphisch Spiele gegen einen Spiel–basierten Modelchecker darzustellen und durchzuführen.

2.5. Java

Die Realisierung der graphischen Simulation erfordert die Programmierung einer dynamischen graphischen Benutzeroberfläche. Mit `Haggis` existiert zwar eine Bibliothek zur Programmierung solcher Oberflächen für den `GHC` [FJ96], erste Versuche haben jedoch gezeigt, daß diese zur Zeit noch fehlerhaft und darüber hinaus nur unvollständig dokumentiert ist.

Aus diesem Grunde war es erforderlich, die Programmierung der graphischen Benutzeroberfläche in einer anderen Programmiersprache vorzunehmen. Die Wahl fiel dafür auf `Java` [GJS97], da diese mit dem Abstract Windowing Toolkit über eine umfangreiche Bibliothek zur Entwicklung graphischer Benutzeroberflächen verfügt [Zuk97].

Darüber hinaus hat `Java` den Vorteil der weitgehenden Plattformunabhängigkeit, da `Java`–Programme nicht direkt in Maschinencode, sondern in Bytecode für eine abstrakte Maschine übersetzt werden. Eine Implementierung dieser abstrakten Maschine existiert für fast alle gebräuchlichen Hardware–Plattformen.

3. Spezifikationen

In TRUTH erfolgt die Spezifikation des zu verifizierenden Systems mit Hilfe einer sogenannten Prozeßalgebra. Solche beschreiben das Verhalten eines Systems in einer kompositionalen Weise. Einfache Systeme werden mit Hilfe von Operatoren zu komplexeren zusammengesetzt. Spezifikationen wird als Semantik ein beschriftetes Transitionssystem zugeordnet, das als Basis für alle weiteren Analysen dient, einschließlich des Modelcheckings.

Dieses Kapitel beschreibt zunächst die verwendete Prozeßalgebra CCS. Es wird definiert, was wir unter einem beschrifteten Transitionssystem verstehen. Auf Basis beschrifteter Transitionssysteme definieren wir die Semantik einer CCS–Spezifikation. Es werden Alternativen zu CCS und beschrifteten Transitionssystemen betrachtet. Den Abschluß dieses Kapitels bildet eine Beschreibung der Aspekte der Implementierung in TRUTH sowie Laufzeitmessungen, die die Leistung von TRUTH in Relation zu der bestehender Werkzeugen setzt.

3.1. CCS

Die jetzige Version von TRUTH verwendet als Spezifikationssprache CCS. CCS wurde 1980 von Milner zur Modellierung verteilter Systeme entwickelt. Es erlaubt die Beschreibung eines verteilten Systems in Form von nebenläufigen Prozessen, die miteinander und mit der Umwelt kommunizieren. Prozesse sind in der Lage, Aktionen durchzuführen und dabei ihren Zustand zu ändern. Dabei stellt jede solche Aktion entweder Kommunikation mit einem benachbarten Prozeß dar oder wird unabhängig von diesen durchgeführt. Diese unabhängige Aktion kann dabei wieder als Kommunikation zwischen Teilprozessen eines Prozesses aufgefaßt werden.

Die Kommunikation zwischen Prozessen ist dabei als synchrone Kommunikation zwischen je zwei Prozessen modelliert, wobei keine Unterscheidung zwischen Sender und Empfänger stattfindet.

CCS verwendet bei der Spezifikation verteilter Systeme einen algebraischen Ansatz. Atomare Prozesse können durch die Anwendung verschiedener Operatoren zu neuen verknüpft werden.

3.1.1. Syntax

In Anlehnung an Milners Definition folgt hier nun die Definition der Syntax von CCS, wie wir sie im weiteren verwenden werden.

Wir definieren zunächst die Aktionen, die CCS–Prozesse ausführen können. Aktionen treten dabei in Paaren wechselseitig komplementärer Aktionen auf. Diese können zur Kommunikation zwischen Prozessen genutzt werden. Eine Aktion spielt eine Sonderrolle. Sie wird mit τ bezeichnet und heißt *stille* oder auch *perfekte* Aktion.

Definition 3.1.1
Es bezeichne $\mathcal{A}$ eine abzählbare Menge von Namen. $\overline{\mathcal{A}} := \{\overline{a} \mid a \in \mathcal{A}\}$ *heißt Menge der* Co–Namen, *wobei wir $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$ voraussetzen. Wir definieren $\mathcal{L}$ durch $\mathcal{L} := \mathcal{A} \cup \overline{\mathcal{A}}$ und nennen diese Menge die* Menge der Kanal–Namen. *Die Menge der* Aktionen *Act ist definiert als $Act := \mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$. τ heißt* stille Aktion. *a und $\overline{a}$ heißen komplementäre Aktionen. Wir erweitern den Definitionsbereich von $\bar{\cdot}$ indem wir festlegen, daß $\overline{\overline{a}} := a$ gelten soll.*

Auch bei der Spezifikation eines Systems möchte man Teile einer Spezifikation wiederverwenden können, um so eine übersichtliche und verständliche Beschreibung des Systems zu erhalten. Zu diesem Zweck führen wir Umbenennungen ein.

Definition 3.1.2
Die Menge aller Abbildungen $f : Act \to Act$ für die gilt

$$f(\alpha) = \begin{cases} \overline{f(\overline{\alpha})} & \text{falls } \alpha \neq \tau, \\ \tau & \text{sonst} \end{cases}$$

heißt Menge der Umbenennungen *und wird im folgenden mit Rel bezeichnet.*

Definition 3.1.3
Es sei Pid eine abzählbare Menge von Prozeßbezeichnern, disjunkt zur Menge der Aktionen. Die Menge der CCS–Ausdrücke *$\mathcal{E}$ ist die kleinste Menge mit $Pid \subset \mathcal{E}$ welche erfüllt: Mit $E, E_1, E_2 \in \mathcal{E}$ gilt*

$0 \in \mathcal{E}$		*Null–Prozeß*
$\alpha.E \in \mathcal{E}$	$(\alpha \in Act)$	*Präfixing*
$E_1 + E_2 \in \mathcal{E}$		*Summe*
$E_1 \mid E_2 \in \mathcal{E}$		*Komposition*
$E \setminus L \in \mathcal{E}$	$(L \subset \mathcal{L})$	*Restriktion*
$E[f] \in \mathcal{E}$	$(f \in Rel)$	*Umbenennung*

Beispiele von CCS–Ausdrücken sind $\mathtt{in}.\overline{\mathtt{out}}.B$ oder auch $B_1 + ((\mathtt{a}.B_2) \mid (\overline{\mathtt{b}}.(P \setminus \{\mathtt{c}\})))$.

Zur Vereinfachung der Notation legen wir eine Präzedenz der Operatoren fest: Umbenennung und Restriktion haben die höchste Priorität, dann folgen Präfixing, Komposition und Summe in absteigender Reihenfolge. Das zweite Beispiel läßt sich mit diesen Konventionen dann auch schreiben als $B_1 + \mathtt{a}.B_2 \mid \overline{\mathtt{b}}.P \setminus \{\mathtt{c}\}$.

Milner führt in seiner Definition die Summe nicht als binäre Operation ein, sondern erlaubt auch die Summation über unendliche Mengen von Prozessen. Während diese Definition eine einfache Einführung wertbehafteter Kommunikation in den Kalkül erlaubt, ist sie natürlich für die Implementierung einer Spezifikationssprache ungeeignet. Aus diesem Grund führen wir die Summe hier nur in der obigen Form ein.

Definition 3.1.4
Eine CCS–Spezifikation *ist ein Paar* (E_0, Eq). *Dabei sei* $E_0 \in \mathcal{E}$ *und* Eq *eine endliche Menge von Gleichungen der Form* $P \stackrel{def}{=} E$ *mit* $P \in Pid$ *und* $E \in \mathcal{E}$, *wobei*

- *jedes* P *nur einmal als linke Seite einer Gleichung auftritt, und*
- *jedes in einer rechten Seite oder in* E_0 *auftretende* P *als linke Seite einer Gleichung in* Eq *auftritt.*

Wir bezeichnen E_0 *als die Anfangskonfiguration der Spezifikation.*

3.1.2. Semantik

Wir geben nun die Semantik von CCS–Spezifikationen an. Diese hat die Form eines beschrifteten Transitionssystems.

Definition 3.1.5 (Beschriftete Transitionssysteme)
Ein beschriftetes Transitionssystem *ist ein Tupel der Form* $\langle S, T, \rightarrow \rangle$, *wobei* S *eine Menge von Zuständen,* T *die Menge der Transitionsbeschriftungen und* $\rightarrow$ *eine Relation über* $S \times T \times S$ *ist.*

Wie üblich schreiben wir $s_1 \stackrel{t}{\rightarrow} s_2$ *für* $(s_1, t, s_2) \in \rightarrow$. *Wir nennen ein beschriftetes Transitionssystem endlich, wenn seine Zustandsmenge endlich ist. Darüber hinaus verwenden wir in dieser Arbeit „Transitionssystem“ als Synonym „für beschriftetes Transitionssystem“.*

Definition 3.1.6 (Einzelschritt–Ableitungen)
Es sei $\langle E_0, Eq \rangle$ *eine CCS–Spezifikation. Für alle* $E, E_i \in \mathcal{E}$ *und* $\alpha \in Act$ *ist* $\stackrel{\alpha}{\rightarrow}$ *durch die Regeln in Abbildung 3.1 induktiv über die Syntax von CCS definiert.*

Wir erhalten nun die Transitionssystem–Semantik einer Spezifikation, indem wir jeden Prozeß–Ausdruck als Zustand eines beschrifteten Transitionssystems auffassen und die Relation $\rightarrow$ gemäß

der Einzelschritt–Ableitungen definieren.

Definition 3.1.7 (Transitionssystem–Semantik)
$\Rightarrow$ *ist definiert durch* $\Rightarrow := \{(E_1, E_2) \mid \exists \alpha \in Act : E_1 \stackrel{\alpha}{\rightarrow} E_2\}$, $\Rightarrow^*$ *bezeichne die reflexive, transitive Hülle von* $\Rightarrow$.

$$\text{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \qquad \text{Sum}_1\ \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 + E_2 \xrightarrow{\alpha} E_1'}$$

$$\text{Sum}_2\ \frac{E_2 \xrightarrow{\alpha} E_2'}{E_1 + E_2 \xrightarrow{\alpha} E_2'} \qquad \text{Par}_1\ \frac{E_1 \xrightarrow{\alpha} E_1'}{E_1 \mid E_2 \xrightarrow{\alpha} E_1' \mid E_2}$$

$$\text{Par}_2\ \frac{E_2 \xrightarrow{\alpha} E_2'}{E_1 \mid E_2 \xrightarrow{\alpha} E_1 \mid E_2'} \qquad \text{Com}\ \frac{E_1 \xrightarrow{\alpha} E_1' \quad E_2 \xrightarrow{\overline{\alpha}} E_2'}{E_1 \mid E_2 \xrightarrow{\tau} E_1' \mid E_2'}$$

$$\text{Res}\ \frac{E \xrightarrow{\alpha} E'}{E \setminus L \xrightarrow{\alpha} E' \setminus L}\quad \alpha \notin L \qquad \text{Rel}\ \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\text{Eq}\ \frac{E \xrightarrow{\alpha} E'}{P \xrightarrow{\alpha} E'}\quad P \in Pid, P = E \in Eq$$

Abbildung 3.1: CCS–Einzelschritt–Relation

Als Semantik einer CCS–Spezifikation $\langle E_0, Eq\rangle$ definieren wir das beschriftete Transitionssystem $\langle S, Act, \rightarrow|_S\rangle$, wobei $S := \{E \in \mathcal{E} \mid (E_0, E) \in \Rightarrow^*\}$ die Menge der von E_0 erreichbaren Ausdrücke ist. $\rightarrow|_S$ ist die Einschränkung von $\rightarrow$ auf die Zustandsmenge S. Wir bezeichnen die Semantik von $\langle E_0, Eq\rangle$ mit $[\![E_0]\!]_{Eq}$.

Wie geben hier ein einfaches Beispiel für eine CCS–Spezifikation und das sich daraus ergebende Transitionssystem. Ein größeres Beispiel folgt in Kapitel 6.

Eine einfache Gleichung für einen Puffer mit der Kapazität eins, der ein Element aufnimmt, dann wieder ausgibt und in den Anfangszustand zurückkehrt, sieht folgendermaßen aus:

$$B \stackrel{\text{def}}{=} \texttt{in}.\overline{\texttt{out}}.B$$

Durch geeignetes Verbinden zweier solcher Puffer ergibt sich ein Puffer mit Kapazität zwei.

$$B_2 \stackrel{\text{def}}{=} (B[\texttt{int/out}] \mid B[\texttt{int/in}]) \setminus \texttt{int}$$

Als Semantik dieser Spezifikation mit Anfangskonfiguration B_2 ergibt sich das in 3.2 dargestellte Transitionssystem. An diesem Beispiel läßt sich auch gut die Berechnung der Einzelschritte gemäß der SOS–Regeln darstellen. So ergibt sich der mit $(*)$ beschriebene Schritt gemäß der Ableitung in Abbildung 3.3.

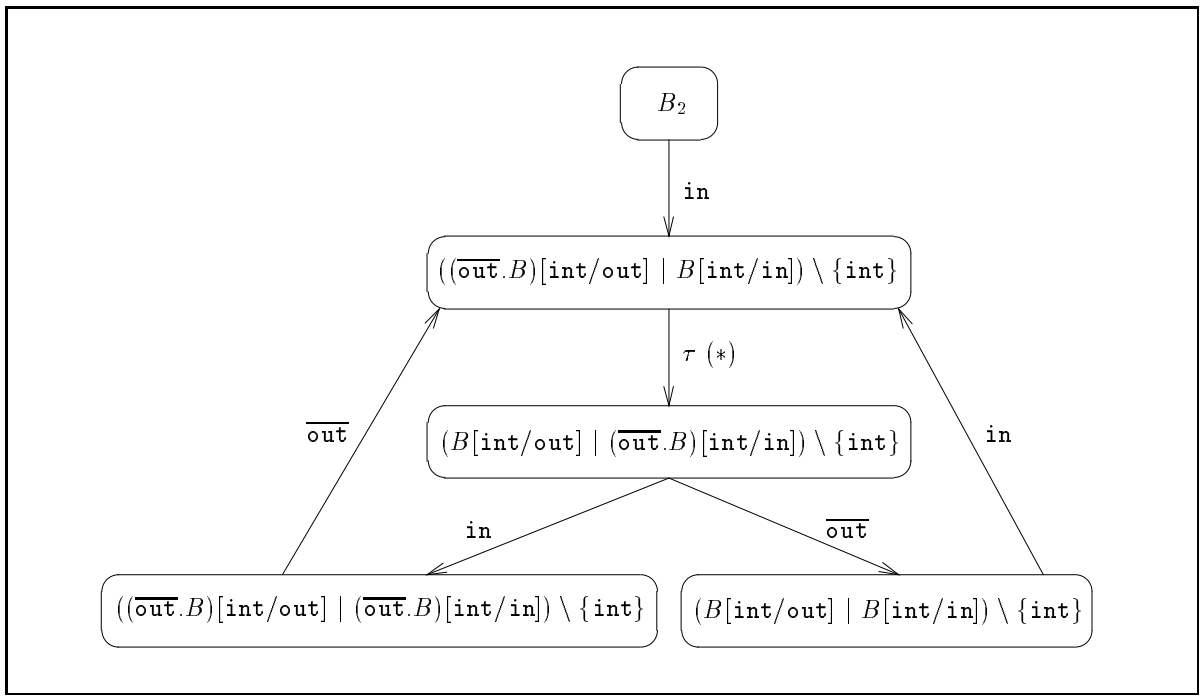


Abbildung 3.2: Semantik einer CCS–Spezifikation

Dieses Beispiel benutzt einige vereinfachenden Schreibweisen, die an dieser Stelle vereinbart werden: Für einelementige Mengen schreiben wir „$E \setminus a$“ anstelle von „$E \setminus \{a\}$“. Darüber hinaus geben wir Umbenennungen in der Form

$$[Bild_1/Urbild_1, \ldots, Bild_n/Urbild_n]$$

an, wobei wir die Abbildung implizit auf die komplementären Aktionen erweitern.

Für die Entscheidbarkeit des Modelcheckings für den μ–Kalkül, wie wir es betrachten, ist es im allgemeinen notwendig, daß das sich aus einer Spezifikation ergebende Transitionssystem endlich ist. Es gibt eine Reihe syntaktischer Einschränkungen von CCS die dies garantieren [DB95]. Für uneingeschränktes CCS gilt dies jedoch nicht, wie das folgende einfache Beispiel zeigt:

$$P \stackrel{\text{def}}{=} \texttt{a}.(P \mid P)$$

$$\dfrac{\dfrac{\overline{\texttt{out}}.B \xrightarrow{\overline{\texttt{out}}} B}{(\overline{\texttt{out}}.B)[\texttt{int/out}] \xrightarrow{\overline{\texttt{int}}} B[\texttt{int/out}]} \quad \dfrac{\dfrac{\texttt{in}.\overline{\texttt{out}}.B \xrightarrow{\texttt{in}} \overline{\texttt{out}}.B}{B \xrightarrow{\texttt{in}} (\overline{\texttt{out}}.B)}}{B[\texttt{int/in}] \xrightarrow{\texttt{int}} (\overline{\texttt{out}}.B)[\texttt{int/in}]}}{((\overline{\texttt{out}}.B)[\texttt{int/out}] \mid B[\texttt{int/in}]) \xrightarrow{\tau} (B[\texttt{int/out}] \mid (\overline{\texttt{out}}.B)[\texttt{int/in}])}}{((\overline{\texttt{out}}.B)[\texttt{int/out}] \mid B[\texttt{int/in}]) \setminus \{\texttt{int}\} \xrightarrow{\tau} (B[\texttt{int/out}] \mid (\overline{\texttt{out}}.B)[\texttt{int/in}]) \setminus \{\texttt{int}\}}$$

Abbildung 3.3: Eine Beispiel–Ableitung eines CCS–Ausdrucks

Als Semantik dieser Spezifikation erhält man einen unendlichen Baum.

Darüber hinaus ist die Eigenschaft einer CCS–Spezifikation, ein endliches Transitionssystem als Semantik zu besitzen, unentscheidbar. Dies ergibt sich aus dem folgenden Satz, der in [Mil89] als Übung auftaucht.

Satz 3.1.8
Es existiert eine rekursive Funktion, welche jede Turingmaschine $\mathcal{M}$ auf eine CCS–Spezifikation $\langle M_0, Eq_{\mathcal{M}}\rangle$ abbildet, so daß gilt: $[\![E_0]\!]_{Eq}$ ist endlich, genau dann, wenn $\mathcal{M}$ auf dem leeren Wort nach endlich vielen Schritten hält.

Beweisidee: Für ein beliebiges endliches Alphabet Σ läßt sich auf einfache Weise eine Spezifikation $\langle Empty, Eq_\Sigma\rangle$ für einen Stack angeben:

$$
\begin{aligned}
Empty &= \sum_{\sigma\in\Sigma} \mathtt{push}_\sigma.C_\sigma \\
C_\sigma &= \mathtt{pop}_\sigma.Empty + \sum_{\sigma'\in\Sigma} (\mathtt{push}_{\sigma'}.D_{\sigma'} \mid \overline{\mathtt{int}}.\mathtt{pop}_\sigma.Empty) \setminus \{\mathtt{int}\} \\
D_\sigma &= \mathtt{pop}_\sigma.\mathtt{int}.0 + \sum_{\sigma'\in\mathcal{A}} (\mathtt{push}_{\sigma'}.D_{\sigma'} \mid \overline{\mathtt{int}}.\mathtt{pop}_\sigma.0) \setminus \{\mathtt{int}\}
\end{aligned}
$$

Mit der Hilfe zweier solcher Stacks läßt sich dann auf einfache Weise das Verhalten von $\mathcal{M}$ modellieren. □

Aus der Unentscheidbarkeit des speziellen Halteproblems für Turing–Maschinen folgt unmittelbar:

Korollar 3.1.9 *Es ist unentscheidbar, ob für eine CCS–Spezifikation $\langle P_0, Eq\rangle$ gilt: „$[\![P_0]\!]_{Eq}$ ist endlich“.*

3.2. Alternativen

Sowohl die Wahl von CCS zur Spezifikation als auch die der beschrifteten Transitionssysteme zur Modellierung der Semantik lassen sich in Frage stellen. Da dies einen Einfluß auf die Implementierung des jetzigen Systems gehabt hat, sollen hier einige alternative Ansätze betrachtet werden.

3.2.1. Spezifikationssprachen

Es existiert eine ganze Reihe von Spezifikationssprachen für verteilte Systeme. Dazu gehören verschiedene Varianten von CCS, in denen der hier vorgestellte Kalkül um eine Reihe weiterer Operatoren erweitert oder Aktionen um Prioritäten angereichert werden. Es gibt aber auch eine ganze Reihe von Formalismen, die sich nicht als Nachfolger von

CCS begreifen, sondern andere Schwerpunkte setzen. Hierzu gehören zum Beispiel CSP [Hoa83], der π–Kalkül [MPW91] oder auch Promela [Hol91].

Jeder dieser Formalismen hat seine Existenzgrundlage, und die Auswahl einer Spezifikationssprache wird sich stets am betrachteten Problem orientieren.

So unterschiedlich diese Ansätze auch sein mögen, sie haben auch eine Reihe wesentlicher Gemeinsamkeiten:

- Man kann allen erwähnten Spezifikationssprachen eine Semantik auf Basis beschrifteter Transitionssysteme zuordnen.
- Die Nachfolger einer Konfiguration lassen sich lokal anhand fester Regeln bestimmen.
- Der μ–Kalkül ist eine geeignete Logik, um Eigenschaften der jeweiligen Spezifikation zu beschreiben und durch Modelchecking zu verifizieren.

Diese Beobachtung legt nahe, mit einem Werkzeug nicht eine einzelne Spezifikationssprache, sondern mehrere zu unterstützen. Durch eine geeignete Architektur des Verifikations–Werkzeugs sind dafür nur wenige Änderungen vorzunehmen. Diesen Ansatz verfolgt die NCSU Concurrency Workbench, für die Frontends für CCS, PCCS und Basic Lotos existieren.

Darüber hinaus existiert ein Werkzeug, welches die sprachabhängigen Funktionen automatisch generiert [CMS95]. Diese müssen dann nur noch in die NCSU CWB eingebunden und das System neu kompiliert werden, um eine Unterstützung der Spezifikationssprache zu integrieren.

Die generierten Funktionen übernehmen die syntaktische Verarbeitung von Eingaben in der jeweiligen Spezifikationssprache. Darüber hinaus wird eine Funktion erzeugt, die die Berechnung der Einzelschritt–Ableitungen durchführt. Alle weiteren benötigten Funktionen sind sprachunabhängig ausgelegt und gehören zum Gerüst der CWB.

Grundlage für die Erzeugung eines neuen Frontends bildetz dabei eine syntaktische und semantische Beschreibung der Spezifikationssprache. Erstere ist mehr oder weniger eine YACC–Spezifikation und definiert die lexikalischen Klassen sowie die kontextfreie Grammatik der Spezifikationssprache. Letztere ist eine Formalisierung der Einzelschritt–Ableitungen mit Hilfe von SOS–Regeln. Zusätzlich können ML–Prädikate und Funktionen eingefügt werden.

Für TRUTH ist die Entwicklung eines ähnlichen Werkzeuges geplant, wobei darüber hinaus versucht werden soll, die Rewriting Logic [Mes90] als Rahmen zur Implementierung verschiedener Semantiken zu verwenden. Diesem Umstand mußte bei der Entwicklung Rechnung getragen werden. Es führte zur Errichtung einer Abstraktionsbarriere zwischen der CCS–Komponente und dem restlichen System, die einen leichten Austausch der Spezifikationssprache unterstützt.

3.2.2. Semantische Bereiche

Transitionssysteme haben sich als semantische Bereiche bei der Verifikation verteilter Systeme weitgehend durchgesetzt. Dies liegt vor allem daran, daß beschriftete Transitionssysteme in einem gewissen Sinne die einfachste Struktur darstellen, die man als Semantik für verteilte Systeme verwenden kann. Die Modellierung eines Systems als Menge von Zuständen, zwischen denen beschriftete Übergänge bestehen, ist eine naheliegende Sichtweise.

Daher existieren auch für die meisten Spezifikationssprachen einfache Übersetzungen in beschriftete Transitionssysteme. Es gibt eine reiche Theorie der beschrifteten Transitionssysteme, insbesondere existieren verschiedene Logiken, die sich eignen, deren Eigenschaften zu beschreiben.

Neben allen diesen Vorteilen haben beschriftete Transitionssysteme, in der Form, in der sie üblicherweise genutzt werden, für die Anwendung des Modelcheckings einen gravierenden Nachteil: Die Art der Modellierung von Nebenläufigkeit durch Interleaving und Nicht–Determinismus, wie sie bei Transitionssystemen notwendig ist, führt zu der sogenannten Zustands–Explosion (state explosion). Damit ist die Tatsache gemeint, daß die Größe eines Transitionssystems sich im schlimmsten Fall exponentiell in der Anzahl der parallelen Prozesse eines Systems verhalten kann. Ein System aus 100 Prozessen, die alle jeweils nur 10 Zustände annehmen können, kann so im schlimmsten Fall 10^{100} Zustände umfassen.

Aus diesem Grund ist eine ganze Reihe semantischer Bereiche vorgeschlagen worden, in denen dieses Problem durch die Anwendung fortgeschrittener Methoden bei der Modellierung von Nebenläufigkeit umgangen werden kann. Solche semantischen Strukturen sind zum Beispiel Traces, Ereignisstrukturen und auch Chu–Spaces.

Diese lassen sich auch als Transitionssysteme auffassen, doch verfügen die Zustände in diesem Fall über eine zusätzliche Struktur, die es erlaubt, Semantiken besser und speichersparender zu beschreiben. Daher eignet sich die jetzige Implementierung von TRUTH auch als Grundstock für die Implementierung solcher, fortgeschritteneren Ansätze.

Diese semantischen Bereiche spielen beim Einsatz in Modelcheckern zur Zeit noch keine wichtige Rolle. Dies liegt vor allem daran, daß Logiken, die Eigenschaften solcher Strukturen ausdrücken können, bisher noch nicht gut verstanden sind. Dies schlägt sich vor allem im Fehlen praktikabler Modelchecking–Algorithmen nieder, so daß ein Einsatz in TRUTH zum jetzigen Zeitpunkt nicht sinnvoll ist. Darüber hinaus existieren wenige Ansätze, solche Strukturen aus Spezifikationen mit üblichen Formalismen zu erhalten, was einen Einsatz zusätzlich erschwert.

3.3. Implementierung in Truth

TRUTH implementiert eine Variante der oben beschriebenen Prozeßalgebra zur Spezifikation verteilter Systeme. Eingaben werden auf ihre syntaktische Korrektheit überprüft und in einer Umgebung abgelegt. Diese läßt sich ausgeben und modifizieren. Aufbauend auf eine Berechnung der Einzelschritt–Transitionen von CCS–Ausdrücken ist es möglich, das sich als Semantik der Spezifikation ergebende Transitionssystem zu berechnen. Dieses dient als Basis weiterer Analysen und des Modelcheckings.

Besonderer Wert wird auf die zeit– und platzeffiziente Konstruktion des Transitionssystems gelegt, da es sich hier um die Kern–Funktion von TRUTH handelt.

Wie im vorigen Abschnitt angedeutet, existiert eine Schnittstelle zwischen der CCS–Komponente und der Transitionssystem–Komponente. Diese ist in einem Modul namens `ProcAlg.lhs` zusammengefaßt. Dieses Modul bildet den geeigneten Einstiegspunkt zur Beschreibung der Implementierung.

3.3.1. Die Prozeßalgebra–Schnittstelle

Das Typsystem von `Haskell` erlaubt zur Zeit keine Multiparameter–Klassen. Dadurch ist es nicht möglich, eine Prozeßalgebra–Klasse einzuführen. Stattdessen behelfen wir uns mit einem Modul, in dem alle benötigten Typen und Funktionen eingeführt und durch konkrete Typen und Funktionen einer speziellen Prozeßalgebra definiert werden. Einige Typsynonyme, insbesonere `Process` und `Action` werden verwendet, um von den zur Implementierung verwendeten Typen zu abstrahieren. Ähnlich geschieht die Vereinbarung der Funktionen durch einfache Zuweisung der konkreten Funktionen zu dort vereinbarten Namen.

Die Implementierung des Modelcheckers und des Transitionssystem basiert auf den so eingeführten Typsynonymen und Namen und ist somit unabhängig von der konkreten Implementierung.

Die Einführung einer neuen Prozeßalgebra erfordert dann eine „Neu–Instaziierung“ dieses Moduls durch die entsprechenden Typen und Funktionen der neuer Prozeßalgebra. Dafür ist allerdings eine Neukompilation des gesamten Systems notwendig.

Dadurch ist es jedoch nicht mehr möglich, mehrere Prozeßalgebren in *einer* Version von TRUTH zu unterstützen. Vielmehr muß für jede Prozeßalgebra eine eigene Binär–Datei erstellt werden.

Im Modul `ProcAlg.lhs` sind alle Typen und Funktionen zusammengefaßt, die zur Verarbeitung von Spezifikationen in einer Prozeßalgebra notwendig sind. Diese werden im folgenden dargestellt. In der jetzigen Version von TRUTH werden diese Typen und Funktionen durch Typen und Funktionen der CCS–Komponente realisiert.

Typdefinitionen

Es existieren vier wichtige Typen, die eine Prozeßalgebra unterstützen muß: `Process`, `Action`, `ProcEnv` und `ActEnv`.

`Process` und `Action` bilden die elementaren Bausteine einer Prozeßalgebra und der Transitionssystem–Semantik. Die Typen, die verwendet werden, um diese zu modellieren unterliegen nur wenigen Einschränkungen. `Process` muß Instanz von `Eq`, `Hashable` und `Outputable` sein. Das heißt, es muß ein Test auf Gleichheit, die Berechnung eines Hash–Wertes sowie eine Ausgabe mit Hilfe des Pretty–Printers für diesen Typ möglich sein. `Action` muß darüber hinaus noch Instanz von `Ord` sein, da Aktionen als Schlüssel in `FiniteMap`s auftauchen und dies die Existenz einer linearen Ordnung auf dem Typ voraussetzt..

Spezifikationen werden in einer Umgebung vom Typ `ProcEnv` abgelegt. Diese Umgebung bildet die Basis der meisten Operationen, die in diesem Modul deklariert werden. Dazu gehört das Einlesen von Definitionen und die Berechnung der Einzelschritt–Transitionen.

Eine Umgebung vom Typ `ActEnv` wird benutzt, beim Parsen die Vergabe von `Unique`s für Aktionen zu koordinieren.

Bei der Funktion, die die Berechnung der Einzelschritt–Ableitungen durchführt, treten zwei weitere Typen auf:

```
type ProcSuccs     = [(Action,[Process])]
type ProcSuccsHT s = HashTable s Process (FiniteMap Action [Process])
```

`ProcSuccs` dient zur Rückgabe der berechneten Nachfolger. In einer Liste werden dabei Paare von Aktionen und der mittels dieser Aktion erreichbaren Prozesse zurückgegeben.

Zur Optimierung der Berechnung der Nachfolger wird eine Hash–Tabelle benutzt, in der zu Prozessen bereits berechnete Nachfolger abgelegt werden. Diese hat den Typ `ProcSuccsHT`. Die Anordnung der Nachfolger geschieht hier nicht in Form einer Liste sondern mit Hilfe einer Abbildung vom Typ `FiniteMap Action [Process]`. Diese Modellierung hat Vorteile bei der induktiven Berechnung der Nachfolger, wie weiter unten beschrieben wird.

Funktionen

Die Implementierung einer Prozeßalgebra muß Funktionen zur syntaktischen Behandlung von Spezifikationen bereitstellen. Darüber hinaus müssen verschiedene Zugriffe auf die `ProcEnv`–Umgebung möglich sein. Schließlich muß eine Funktion zur Berechnung der Nachfolger eines Prozesses vorhanden sein:

```
transitions:: ProcEnv -> Process -> ProcSuccsHT s -> SST s ProcSuccs
```

Die Berechnung ist dabei eingebettet in eine Zustands–Monade, um die Verwendung der Hash–Tabelle zu ermöglichen.

Zwei Funktionen dienen dazu, Zugriff auf den verwendeten Parser zu ermöglichen.

```
runEqParser::   ProcEnv -> IdEnv -> ActEnv -> UniqSupply -> String ->
                Either (ProcEnv,IdEnv,ActEnv) String
runProcParser:: ProcEnv -> IdEnv -> ActEnv -> UniqSupply -> Bool -> String ->
                Either (ProcEnv,IdEnv,ActEnv,Process) String
```

Die erste dient dazu, neue Definitionen in die Umgebung `ProcEnv` einzutragen. Zur Vergabe der eindeutigen Bezeichner sind darüber hinaus noch zwei zusätzliche Umgebungen vorhanden. Schließlich wird ein zusätzlicher `UniqSupply` bereitgestellt, der für ähnliche Zwecke verwendet werden kann. Das Ergebnis dieser Funktion sind entweder die modifizierten Umgebungen oder, im Falle eines Parser–Fehlers, eine Fehlermeldung.

Die zweite Funktion dient dazu, einen einzelnen Prozeßterm der Prozeßalgebra zu parsen und liefert diesen dann zusammen mit den modifizierten Umgebungen, bzw. wiederum eine Fehlermeldung.

Schließlich existieren weitere Funktionen, die dazu dienen, Umgebungen zu erzeugen und abzufragen:

```
newProcEnv::        UniqSupply -> ProcEnv
newActEnv::         UniqSupply -> ActEnv

printBinding::      PprStyle -> ProcEnv -> Id -> Pretty
printAllBindings:: PprStyle -> ProcEnv -> Pretty
isDefinedProcId::  ProcEnv -> Id -> Bool
```

3.3.2. Implementierung der CCS–Prozeßalgebra

Als Prozeßalgebra, die diese abstrakte Schnittstelle ausfüllt, wird in TRUTH zur Zeit CCS verwendet. Es folgt eine Beschreibung der Implementierung der in `ProcAlg.lhs` festgelegten Typen und Funktionen für CCS.

Der Datentyp, der benutzt wird, um CCS–Ausdrücke zu repräsentieren, orientiert sich direkt an der rekursiven Definition der Syntax der Ausdrücke. Allerdings wird die Syntax dabei etwas erweitert, um auch Bezeichner für Restriktionen und Umbenennungen zuzulassen.

```
data CCSProcess
  = CCSNil
  | CCSNamedProc Id
  | CCSPref CCSAction CCSProcess
```

```
  | CCSSum CCSProcess CCSProcess
  | CCSPar CCSProcess CCSProcess
  | CCSRel CCSProcess CCSRelabeling
  | CCSRes CCSProcess CCSRestriction
  | CCSNamedRel CCSProcess Id
  | CCSNamedRes CCSProcess Id deriving Eq
data CCSRelabeling
  = MkCCSRel (FiniteMap CCSAction CCSAction) Unique
data CCSRestriction
  = MkCCSRes (Set CCSAction) Unique
```

`CCSRelabeling` und `CCSRestriction` beinhalten die zur Berechnung der Einzelschritt–Ableitungen benötigten Informationen. Zusätzlich werden `Unique`s abgelegt, um Vergleich und Berechnung der Hashing–Funktion zu beschleunigen.

Beide sind als abstrakte Datentypen implementiert, so daß eine Änderung der Realisierung leicht möglich ist.

Ein mittels `Happy` generierter Parser wird benutzt, um Gleichungen der Form $Id = Rhs$ einzulesen. Beim Einlesen wird jedem neuen Bezeichner ein eindeutiger Wert vom Typ `Unique` zugewiesen, der als Basis für Vergleiche und die Berechnung der Hashing–Funktion verwendet wird. Das gleiche gilt für alle Aktionen. Die CCS–Definitionen werden dann in einer Umgebung vom Typ `CCSEnv` abgelegt. Diese speichert die Zuordnungen von Bezeichnern zu Prozessen, Restriktionen und Umbenennungen.

```
data CCSEnv     = MkCCSEnv CCSProcEnv CCSResEnv CCSRelEnv
type CCSProcEnv = FiniteMap Id CCSProcess
type CCSResEnv  = FiniteMap Id CCSRestriction
type CCSRelEnv  = FiniteMap Id CCSRelabeling
```

Mit verschiedenen Funktionen können dieser Umgebung Definitionen hinzugefügt, Definitionen für Bezeichner abgefragt und alle Umgebungen gelöscht werden.

Der Parser verarbeitet CCS–Ausdrücke gemäß der oben vereinbarten Präzedenzen der Operatoren. Zwar unterstützt `Happy` keine Präzedenzen der Regeln, jedoch kann man die Grammatik einfach so modifizieren, daß der Parser das gewünschte Verhalten zeigt.

Umgekehrt ist auch ein Pretty–Printer implementiert, der unter Berücksichtigung der Präzedenzen und Assoziativität der Operatoren `+` und `|` eine Ausgabe von CCS–Ausdrücken mit möglichst wenigen Klammern ermöglicht.[1]

Das Modul `CCSSOS.lhs` implementiert die Berechnung der Einzelschritt–Ableitungen eines CCS–Ausdrucks unter Berücksichtigung der in der Umgebung gespeicherten Definitionen. Die Speicherung der Nachfolger eines Prozesses geschieht dabei mit Hilfe des

[1] Diese Operatoren sind nicht wirklich assoziativ. Die enstehenden Transitionssysteme lassen sich jedoch weder durch die implementierten Analysen noch durch Formeln des μ–Kalküls unterscheiden.

Typs `FastSuccs = FiniteMap CCSAction [CCSProcess]`. Dieser erlaubt eine effizientere Verknüpfung der Nachfolger zusammengesetzter Prozesse. Dies ist notwendig, da die Berechnung induktiv gemäß der Regeln in Abbildung 3.1 durchgeführt wird, wobei zunächst stets versucht wird, den zu berechnenden Prozeß in der Hash–Tabelle nachzuschlagen. Eine Menge der bereits betrachteten Prozeßbezeichner wird dabei benutzt, um endlose Rekursion der Prozeßbezeichner zu erkennen.

```
getCCSSuccs'::CCSEnv -> Set Id -> CCSProcess -> SOS_M s FastSuccs
getCCSSuccs' ccsenv seen_ids proc
  = lookupHTblSOS proc `thenSOS` \ maybe_result ->
    case maybe_result of
      Just result -> returnSOS result -- already calculated
      Nothing -> calcCCSSuccs ccsenv seen_ids proc `thenSOS` \ result ->
                 addToHTblSOS proc result `seqSOS` -- store in htable
                 returnSOS  result
```

Auf diese Weise wird eine mehrfache Berechnung der Nachfolger des gleichen Ausdrucks vermieden.

Am Beispiel der Berechnung der Nachfolger einer Komposition von Prozessen sieht man leicht, wie die Berechnung induktiv über den Aufbau definiert ist, und wie die Ergebnisse kombiniert werden.

```
calcCCSSuccs:: CCSEnv -> Set Id -> CCSProcess -> SOS_M s FastSuccs
calcCCSSuccs ccsenv seen_ids (CCSPar proc1 proc2)
  = getCCSSuccs' ccsenv seen_ids proc1 `thenSOS` \ succs1 ->
    getCCSSuccs' ccsenv seen_ids proc2 `thenSOS` \ succs2 ->
    let
      alt1 = mapFM (\ _ procs -> map (\ p -> CCSPar p proc2) procs) succs1
      alt2 = mapFM (\ _ procs -> map (\ p -> CCSPar proc1 p) procs) succs2
      nonpar_succs = plusFM_C (++) alt1 alt2
      par_succs = combineSuccsPar (CCSPar) succs1 succs2
    in returnSOS (plusFM_C (++) nonpar_succs par_succs)
```

Durch den Aufruf von `getCCSSuccs'` wird auch für Teilausdrücke zunächst die Hash–Tabelle konsultiert.

Effizienz

Die Berechnung von Nachfolgern von Ausdrücken ist die zentrale Funktion bei der Konstruktion der Transitionssystem–Semantik. Daher wurde großen Wert auf eine effiziente Implementierung gelegt. Platzeffizienz erweist sich hierbei als wichtiger als Zeiteffizienz, wobei es in der vorliegenden Implementierung gelungen ist, beides gegenüber einer naiven Implementierung, die sich aus einer direkten Implementierung der SOS Regeln ergibt, zu verbessern.

Kern dieser Optimierung ist, wie oben deutlich wurde, eine Hash–Tabelle, die CCS–Ausdrücke auf ihre Nachfolger abbildet. Mit Hilfe einer solchen Tabelle werden zwei Ziele verfolgt.

- Der Aufwand zur Berechnung von Nachfolgern steigt in der Größe der Ausdrücke. Komplexe Spezifikationen bringen so eine hohe Komplexität schon in der Berechnung der Einzelschritt–Nachfolger mit sich. Durch die Verwendung der Hash–Tabelle wird es möglich, eine Neuberechnung zu vermeiden, wenn die Nachfolger für einen Ausdruck bereits bestimmt wurden.
- Während der Konstruktion des Transitionssystems muß die Beschriftung jedes Zustandes im Speicher vorhanden sein, um Zykel im Transitionssystem erkennen zu können. Daher ist auch der Speicherbedarf dieser Ausdrücke von Bedeutung. Die Verwendung der Hash–Tabelle macht es möglich, das Sharing in der Termdarstellung der Beschriftungen zu vergrößern und so Speicherplatz zu sparen.

Eine Übersicht über das Laufzeit–Verhalten von TRUTH im Hinblick auf die Behandlung von Transitionssystemen findet sich am Ende dieses Kapitels.

3.3.3. Erzeugung und Speicherung des Transitionssystems

Kernstück von TRUTH ist die Konstruktion von und der Zugriff auf beschriftete Transitionssysteme. Sowohl die verschiedenen Analysen wie Suche nach Deadlocks als auch das Modelchecking sind Verfahren, die auf Transitionssystemen operieren.

Hieraus ergeben sich verschiedene Anforderungen an die Datenstrukturen, die verwendet werden, um Transitionssysteme darzustellen.

Es handelt sich bei einem solchen Transitionssystem in der Regel um eine zyklische Struktur. Eine solche läßt sich in einer rein–funktionalen Sprache nicht direkt, das heißt unter Verwendung von Zeigern, repräsentieren. Statt dessen ist man gezwungen, die Verkettung durch symbolische Zeiger, zum Beispiel durch Indizes in einem Array auszudrücken. Prozeßterme eignen sich nicht als Indexmenge für eine Array. Das liegt daran, daß für Typen, die als Indexmenge von Arrays auftreten sollen, die Typklasse `Ix` implementiert sein muß. Die dort auftretende Funktion `range`, die zwei Werte des Typs auf eine endliche Liste der sich dazwischen befindenden Werte abbildet, läßt sich für eine Termstruktur nicht sinnvoll implementieren.

Bei der Konstruktion wächst das Transitionssystem durch die Hinzunahme neuer Zustände, die sich als Folgezustände bereits gespeicherter Zustände ergeben. Darüber hinaus müssen bereits vorhandene Zustände erkannt werden, so daß Zykel im Transitionssystem erkannt und als solche repräsentiert werden. Da Zustände nicht mehr mit Prozessen identifiziert werden, ist es daher notwendig, eine Zuordnung von Prozeßtermen zu den Zuständen vorzunehmen.

Bei den verschiedenen Analysen und beim Modelchecking ist es notwendig, das Transitionssystem auf unterschiedliche Weise zu annotieren und so Informationen im Transitionssystem bei den verschiedenen Zuständen abzulegen. Die Art dieser Informationen unterscheidet sich je nach Art der Analyse. Daneben ist es notwendig, von einem Zustand auf alle über eine bestimmte Aktion erreichbaren Folgezustände zuzugreifen. Daneben ist es wünschenswert, das Transitionssystem bedarfsgesteuert zu erzeugen, so daß nur Zustände expandiert werden, deren Nachfolger für die verschiedenen Analysen von Interesse sind.

Damit eine effiziente Änderbarkeit einer großen Datenstruktur möglich ist, muß man auf destruktive Updates zurückgreifen, die eine Änderung der Datenstruktur ermöglichen, ohne daß Kopien großer Teile der Struktur erzeugt werden müssen. Dies ist möglich durch Verwendung einer Zustands–Monade.

Um die Implementierung möglichst variabel zu gestalten, sollte der Typ eines Transitionssystems darüber hinaus polymorph in der Wahl der Beschriftung der Kanten und Knoten sein.

Schnittstelle der abstrakten Datenstruktur

Diese Anforderungen lassen sich in einer Schnittstellendefinition zusammenfassen. Der Typ für das Transitionssystem verfügt über vier Typvariablen. Die erste ist der Zustands–Typ, der von der Zustands–Monade benötigt wird, die anderen drei entsprechen den Wahlmöglichkeiten, die oben beschrieben wurden.

```
type MutLTS s label action annotation = ...
```

Zur Erzeugung eines neuen Transitionssystems müssen Anfangsbelegungen für die Annotationen festgelegt werden. Darüber hinaus erscheint es sinnvoll zu vereinbaren, in welchen Schritten das oben geforderte Wachstum eines Transitionssystem stattfinden soll. Dies geschieht mit einem Parameter vom Typ `Int`.

```
newMutLTS:: (Ord b) => Int -> a -> c -> SST s (MutLTS s a b c)
```

Die Anforderung, daß der Typ der Kantenbeschriftung eine Instanz von `Ord` ist, wird für die jetzige Implementierung der Datenstruktur benötigt. Dies ist aber keine wirkliche Einschränkung, da Aktionen in der Regel ohnehin über einen Wert vom Typ `Unique` verfügen werden, der als Basis der Anordnung dienen kann.

Es sind Funktionen nötig, um die Struktur des Transitionssystems zu ändern und abzufragen:

```
getLabelMutLTS:: MutLTS s a b c -> LTSState -> SST s a
setLabelMutLTS:: MutLTS s a b c -> LTSState -> a -> SST s ()
```

```
setSuccsMutLTS:: (Ord b) => MutLTS s a b c -> LTSState -> b -> [LTSState] ->
                 SST s ()
getSuccsMutLTS:: (Ord b) => MutLTS s a b c -> LTSState -> b ->
                 SST s [LTSState]
addSuccMutLTS::  (Ord b) => MutLTS s a b c -> LTSState -> b -> LTSState ->
                 SST s ()
addSuccsMutLTS:: (Ord b) => MutLTS s a b c -> LTSState -> b -> [LTSState] ->
                 SST s ()
getAddLabelMutLTS::MutLTS s a b c -> LTSState -> SST s c
setAddLabelMutLTS::MutLTS s a b c-> LTSState -> c -> SST s ()
```

Schließlich wird noch eine Funktion benötigt, die die Vergabe der Zustände des Transitionssystems regelt.

```
getFreshStateMutLTS:: (Ord b) => MutLTS s a b c -> SST s LTSState
```

Die Datenstruktur

Eine Möglichkeit zur Implementierung einer solchen Datenstruktur in `Haskell` wird hier nun vorgestellt.

Die Zuordnung der Zustände auf ihre Nachfolger geschieht durch ein veränderbares Array vom Typ `MutAdjArr`. Dieses bildet jeden Zustand auf einen Wert vom Typ `FiniteMap tsLbl [LTSState]` ab. So ist über einen Zustand und eine Aktion ein Zugriff auf alle von diesem Zustand über diese Aktion erreichbaren Zustände möglich.

```
type MutAdjArr s tsLbl
  = (MutableArray s LTSState (FiniteMap tsLbl [LTSState]))
```

Die Modellierung mit einer `FiniteMap` hat den Vorteil, daß bei einer großen Menge von auftretenden Aktionen nur Speicherplatz für die Aktionen verbraucht wird, die wirklich von einem Zustand aus möglich sind. Bei einer Modellierung mit einem Array würde hier unter Umständen viel Speicher verschwendet.

Zusätzlich zu der Übergangsrelation soll auch noch eine Beschriftung der Knoten realisiert werden. Alle diese Tabellen sind in einem `MutLTSChunk` zusammengefaßt.

```
data MutLTSChunk s stLbl tsLbl addLbl
  = MkMutLTSChunk (LTSState,LTSState)              -- state range
                  (MutableArray s LTSState stLbl)  -- process labels
                  (MutAdjArr s tsLbl)              -- the successors
                  (MutableArray s LTSState addLbl) -- additional labels
```

Zusätzlich ist der Bereich der in diesem Teil gespeicherten Zustände abgelegt. Diese Aufteilung des Transitionssystems in mehrere Teile resultiert aus der Anforderung, daß ein Transitionssystem in der Lage sein muß, nach Bedarf zu wachsen. Die im `GHC` implementierten veränderbaren Arrays bieten keine Möglichkeit, ihre Größe dynamisch anzupassen. Daher wird das Transitionssystem als eine Liste solcher Teile modelliert, so daß das Transitionssystem durch Hinzunahme weiterer Teile beliebig wachsen kann.

Die vollständige Datenstruktur enthält nun diese Liste zusammen mit einigen Verwaltungsinformationen:

```
data MutLTS s stLbl tsLbl addLbl
  = MkMutLTS (MutableVar s LTSState) -- the next fresh state
             (MutableVar s LTSState) -- the last state we can allocate
             Int -- the size for a chunk
             stLbl -- what to initialize the labels with
             addLbl -- what to initialize the additional labels with
             (MutableVar s [MutLTSChunk s stLbl tsLbl addLbl])
```

Alle Werte, die bei der Benutzung des Transitionssystems eine Veränderung erfahren können, sind dabei als veränderliche Variablen angelegt. Dies ermöglicht eine komfortablere Benutzung der Datenstruktur.

Auf diese Weise sind fast alle Anforderungen an das Transitionssystem, wie sie oben dargestellt wurden, erfüllt. Einzig das Auffinden eines Prozeßterms im Transitionssystem, wie es zur Erkennung von Zykeln bei der Generierung eines Transitionssystems notwendig ist, kann mit dieser Datenstruktur nur auf unbefriedigende Weise gelöst werden.

Wir lösen diese Problem, indem wir eine weitere Hash–Tabelle vom Typ

```
type ProcStateHT s = HashTable s Process LTSState
```

einführen, in der wir zu jedem im Transitionssystem eingetragenen Ausdruck den Wert des zugehörigen Zustands ablegen.

Die Zustands–Monade

Wir fassen das Transitionssystem mit den beiden Hash–Tabellen in einer Monade `LTS_M` zusammen, um eine einfachere Benutzung zu gewährleisten. Diese Monade stellt alle Funktionen zum Zugriff auf das Transitionssystem und die beiden Hash–Tabellen zur Verfügung.

```
type LTS_M s l r = LTSDown s l -> SST s r
data LTSDown s a
  = LTSDown (ProcSuccsHT s)
            (ProcStateHT s)
            (MLTS s a)
type MLTS s addLbl = MutLTS s Process Action addLbl
```

Die zu einer Zustands–Monade gehörende Funktion, die eine Auswertung der Zustandstransformationen bewirkt, ist definiert durch:

```
runLTS::(Int,Int,Int)->l->LTS_M RealWorld l b->b
runLTS (chunksize,sizeht1,sizeht2) defaddlbl do_this
  = runSST (
      newHashTable sizeht1 `thenSST` \ stsuccsht ->
      newHashTable sizeht2 `thenSST` \ stateht ->
      newMutLTS chunksize dummyProcess defaddlbl `thenSST` \ mutlts ->
      do_this (initDown stsuccsht stateht mutlts)
    )
```

Diese erwartet neben der Zustandstransformation eine Reihe weiterer Parameter. Insbesondere ist es notwendig, die Größe der beiden Hash–Tabellen festzulegen. Darüber hinaus muß festgelegt werden, in welchen Abschnitten Speicher für das Transitionssystem alloziert werden soll. Schließlich muß eine anfängliche Beschriftung des Transitionssystems vereinbart werden.

`runLTS` initialisiert die Datenstrukturen entsprechend und führt dann die Zustandstransformation durch.

Annotationen

Im Modul `PAToLTS.lhs` sind verschiedene Funktionen definiert, die den Umgang mit einem Transitionssystem erheblich erleichtern. Insbesondere existiert eine Funktion, die einen Zustand des Transitionssystems expandiert, das heißt alle Nachfolger dieses Zustandes berechnet und in das Transitionssystem einträgt.

```
expandLTSState:: (LTSLabeling l)=>ProcEnv->LTSState->LTS_M s l ProcSuccsRef
```

Um eine wiederholte Expandierung eines Zustandes zu verhindern, wird dabei vorausgesetzt, daß der Typ der Annotation eine Instanz von `LTSLabeling` ist. Diese Typklasse verfügt über die folgenden Methoden:

```
class LTSLabeling a where
  isExpanded::    a -> Bool
  markExpanded::  a -> a
  defLabeling::   a
```

Diese Methoden unterstützen die bedarfsgesteuerte Generierung des Transitionssystems, so daß alle Analysen, die diese benötigen, zur Annotation eine Instanz von `LTSLabeling` verwenden müssen. Dabei wird vereinbart, daß gelten muß:

```
isExpanded defLabeling == False
```

Ist keine Annotation des Transitionssystems für eine Analyse notwendig – dies ist zum Beispiel bei der Suche nach Deadlocks der Fall – reicht es aus, zur Beschriftung `Bool` zu verwenden, um die bedarfsgesteuerte Generierung zu ermöglichen.

```
instance LTSLabeling Bool where
  isExpanded b = b
  markExpanded b = True
  defLabeling = False
```

3.3.4. Analysen

Mit Hilfe dieser Funktionen ist eine ganze Reihe einfacher Analysen für Transitionssysteme implementiert worden. Am Beispiel der Suche nach Deadlocks soll hier noch einmal die Verwendung der Datenstrukturen und Funktionen erläutert werden.

Es ist möglich, logische Formeln zu formulieren, die die Deadlockfreiheit eines Systems fordern. Die Überprüfung des Systems kann dann auch durch den Modelchecker erfolgen, so daß keine explizite Analyse hierfür implementiert werden muß. Im Falle der Suche nach Deadlocks ist man aber neben dem Wissen um die Existenz von Deadlocks auch an den Zuständen interessiert, an denen solche auftreten. Eine derartige Information könnte zwar vom Modelchecker geliefert werden, der in der Lage ist, Gegenbeispiele zu generieren, doch ist so etwas zur Zeit noch nicht implementiert. Daher ist ein Algorithmus, der Deadlocks sucht und die entsprechenden Zustände ausgibt, eine sinnvolle Funktionalität von TRUTH.

Die Funktion, die die Suche nach Deadlocks übernimmt, hat die folgende Signatur:

```
deadlocks:: (Int,Int,Int) -> ProcEnv -> Process -> [Derivative]
```

Sie erwartet die Größe der Tabellen und Abschnitte des Transitionsystems sowie die Umgebung mit der Spezifikation und die Anfangskonfiguration des Systems. Ergebnis ist eine Liste von Werten vom Typ `Derivative`.

```
type Derivative = ([Action],Process)
```

Diese Liste enthält die Deadlock–Zustände sowie die Beschriftung eines Pfades zu jedem solchen Zustand.

Die Berechnung der Zustände ist als Zustandstransformation implementiert, die durch `runLTS` auf ein leeres Transitionssystem und leere Hash–Tabellen angewandt wird. Zunächst wird ein freier Zustand des Transitionssystems mit der Anfangskonfiguration beschriftet und diese Zuordnung in der entsprechenden Hash–Tabelle vermerkt. Die eigentliche Berechnung führt dann die Funktion `buildAndFindDeadlocks` durch, die als Parameter die Umgebung, den zu betrachtenden Zustand und eine Liste mit der Beschriftung eines Pfades zum aktuellen Zustand erhält.

```
deadlocks tablesizes penv proc
  = runLTS tablesizes (defLabeling::Bool)
           ( getFreshStateLTS          `thenLTS` \ st ->
             setLabelLTS st proc       `seqLTS`
             addToStateHTblLTS proc st `seqLTS`
             buildAndFindDeadlocks penv [] st
           )
```

Da wir das Transitionssystem bedarfsgesteuert generieren wollen, müssen wir zur Annotation eine Instanz von `LTSLabeling` wählen. Da wir neben der Information, welche Knoten bereits expandiert wurden, keine weiteren Informationen ablegen müssen, reicht `Bool` für diesen Zweck aus.

`buildAndFindDeadlocks` untersucht für den übergebenen Zustand, ob dieser bereits expandiert wurde. Dies ist in der Annotation der Knoten vermerkt. Ist dies der Fall, wird er nicht weiter betrachtet und die leere Liste zurückgegeben, da keine weiteren Deadlocks gefunden wurden.

Ist der Knoten noch nicht expandiert, so geschieht dies, indem `expandLTSState` aufgerufen wird. Diese Funktion liefert als Ergebnis eine Liste der erreichbaren Zustände und der die Transitionen beschriftenden Aktionen. Ist diese leer, so existieren keine Nachfolger,also handelt es sich bei dem Zustand um einen Deadlock–Zustand. Dieser wird mit der Beschriftung des Pfades zurückgegeben. Der Akkumulator mit den Aktionen wird zunächst umgekehrt, damit die Aktionen in der richtigen Reihenfolge darin stehen, und dann als Pfad zum Zustand mit zurückgegeben.

Sind Nachfolger vorhanden, so erfolgt ein rekursiver Aufruf für die erreichbaren Zustände, wobei der Akkumulator um die entsprechende Aktion erweitert wird. Die Ergebnisse werden zu einer Liste zusammengefügt.

```
buildAndFindDeadlocks::ProcEnv->[Action]->LTSState->LTS_M s Bool [Derivative]
buildAndFindDeadlocks penv accu st
  = ifLTS
     ( getAddLabelLTS st )
     ( returnLTS [] )
     ( expandLTSState penv st `thenLTS` \ succs ->
         case succs of
```

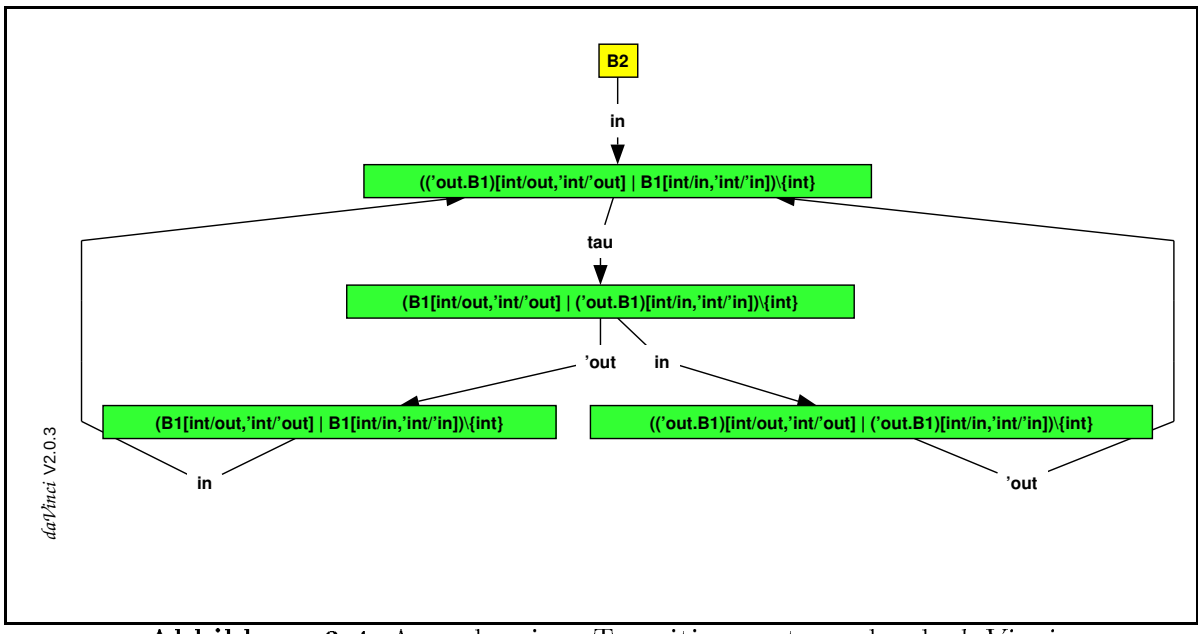


Abbildung 3.4: Ausgabe eines Transitionssystems durch *daVinci*

```
        [] -> getLabelLTS st `thenLTS` \ proc ->
              returnLTS [(reverse accu,proc)]
      somesuccs ->
        let
          next_step (act,states)
            = mapConcatLTS (buildAndFindDeadlocks penv (act:accu))
                states
        in
        mapConcatLTS next_step somesuccs
)
```

3.3.5. Ausgabe des Transitionssystems

Zum Verständnis einer Spezifikation ist es oft hilfreich, eine graphische Darstellung des Transitionssystems zur Verfügung zu haben. TRUTH bietet hierzu die Möglichkeit. Durch Verwendung des Graphvisualisierungs–Werkzeuges *daVinci* ist es möglich, qualitativ hochwertige Ausgaben von Transitionssystemen mit maximal 200 Zuständen zu erhalten. *DaVinci* ist leider nicht in der Lage, größere Systeme in vertretbarer Zeit übersichtlich darzustellen, doch es ist ohnehin fraglich, ob die Darstellung eines so großen Systems für den Benutzer überhaupt verständlich ist. Bei kleineren Systemen kann *daVinci* hingegen sehr gute Ergebnisse erzielen.

Eine Darstellung des Transitionssystemes, wie es bereits in Abbildung 3.2 dargestellt war, findet sich in Abbildung 3.3.5 in einer durch *daVinci* erzeugten Ausgabe. Ein größeres Beispiel befindet sich in Kapitel 6.

3.4. Laufzeit–Messungen

Zum Vergleich der verschiedenen Methoden zur Generierung des Transitionssystems wurden einige Laufzeit–Messungen durchgeführt. Bei den betrachteten Systemen handelt es sich um eine Reihe von realistischen Beispielen, die der Distribution der NCSU Concurrency Workbench beiliegen. Diese werden im Anhang B kurz beschrieben.

Geschwindigkeit

Es werden die Zeiten verglichen, die die veschiedenen Versionen zur Bestimmung der Größe eines Transitionssystems benötigen. Dazu ist im allgemeinen eine vollständige Generierung des Transitionssystems notwendig, die im Falle von TRUTH auch durchgeführt wird. Die verschiedenen Versionen, die verglichen werden, sind:

Naiv	Eine naive Implementierung des SOS–Regeln
Hash	Eine Implementierung, die bereits berechnete Nachfolger in einer Hash–Tabelle speichert.
Hash/Flat	Zusätzlich zum Hashing wird der Vergleich von Restriktion und Relabelings durch Vergabe von `Unique`–Werten beschleunigt.
NCSU	Die NCSU Concurrency Workbench.
CWB	Die Edinburgh Concurrency Workbench in der Version 7.1beta.

Die Messungen fanden auf einer Sun UltraSparc mit 512 MB Hauptspeicher statt.

	Naiv	Hash	H/F	NCSU	CWB
Knuth	0.28 s	0.27 s	0.28 s	0.33 s	0.80 s
802	0.58 s	0.47 s	0.43 s	0.63 s	2.20 s
Mailer	3.31 s	3.35 s	1.60 s	3.48 s	4.24 s
ATM	4.84 s	13.15 s	1.41 s	3.54 s	2.50 s
3LN	131.43 s	94.21 s	52.22 s	59.07 s	74.50 s

Wie man deutlich sehen kann, ist die Version von TRUTH die Hashing und beschleunigte Vergleiche von Restriktionen und Relabelings verwendet, den anderen Versionen deutlich überlegen. Was zunächst erstaunte, war die Verschlechterung der Leistung durch Hinzunahme von Hashing bei der ATM–Spezifikation. Eine Untersuchung mit dem Profiler ergab, daß TRUTH den weitaus größten Teil seiner Zeit mit dem Vergleich von Prozeßtermen verbrachte. Diese Vergleiche sind nötig, um bei der Verwendung von Hashing Kollisionen zu erkennen.

Ausgehend von dieser Beobachtung wurde der Vergleich von CCS–Ausdrücken gezielt optimiert, was zu einer ganz erheblichen Beschleuningung des Systems führte. Die entsprechenden Laufzeiten finden sich unter der Überschrift „Hash/Flat“.

Es war darüber hinaus erstaunlich, festzustellen, daß TRUTH in seiner Geschwindigkeit bei allen betrachteten Beispielen über der der NCSU Concurrency Workbench lag, da diese in ML implementiert ist, einer strikten funktionalen Programmiersprache, die darüber hinaus destruktive Veränderungen von Datenstrukturen erlaubt. Dies kann bei der Implementierung benutzt werden, um zu effizienteren Algorithmen zu gelangen. Dennoch ist TRUTH schneller als NCSU und CWB.

Speicherverbrauch

Noch interessanter als die Geschwindigkeit ist sicherlich der Speicherplatzverbrauch. Auch unter diesem Gesichtspunkt wurden Messungen durchgeführt. Leider ist eine sinnvolle Messung des Speicherplatzverbrauchs der beiden Concurrency Workbenches nicht möglich, da das ML–Laufzeitsystem hierfür keine Vorkehrungen trifft . Die Werte für NCSU und CWB sind Angaben über den maximalen Speicherbedarf des Prozesses bei der Generierung des Transitionssystems. Da dieser niemals unter 13 MB lag, war keine Ermittlung des Speicherbedarfs für die meisten Spezifikationen möglich. Der Speicherplatzverbrauch für die beiden Implementierungen des Hashings in TRUTH unterscheiden sich nicht wesentlich, daher werden diese hier nicht gesondert aufgeführt. Dafür kommt mit **Hash/Clean** eine weitere Implementierung hinzu. Die Angaben beziehen sich auf die maximale Speicherbelegung, die nach eine Garbage Collection auftritt.

	Naiv	Hash	Hash/Clean	NCSU	CWB
Knuth	0.5 MB	1.12 MB	0.51 MB	-	-
802	0.56 MB	1.34 MB	0.55 MB	-	-
Mailer	1.16 MB	3.14 MB	1.00 MB	-	-
ATM	1.47 MB	2.25 MB	1.10 MB	-	-
3LN	15.04 MB	30.78 MB	13.00 MB	47.8 MB	148.5 MB

Diese Zahlen sprechen zunächst gegen den Ansatz, eine Hash–Tabelle zur Beschleunigung der Generierung durch eine Hash–Tabelle zu erreichen, da sich der Speicherplatz–Verbrauch auf ungefähr das Doppelte steigert.

Dieser Effekt kommt dadurch zustande, daß für jeden bei der Berechnung auftretenden Teilausdruck die berechneten Nachfolger in der Hash–Tabelle gespeichert werden, und so diese Hash–Tabelle sehr viel Speicherplatz verbraucht. Unter der Überschrift „Hash/Clean“ sind die Werte für den Speicherplatz angegeben, der sich ergibt, wenn man den Speicherplatz für die Hash–Tabelle nicht mitrechnet. Diese wurden ermittelt, indem nach der Generierung des Transitionssystems die Hash–Tabelle gelöscht und dann eine Garbage Collection erzwungen wurde. Die Werte für die Heap–Größe nach dieser Garbage Collection liegen jeweils um ungefähr 10 Prozent unter denen der Version, die ganz ohne Hash–Tabelle auskommt. Dies liegt an der Tatsache, daß durch die

Wiederverwendung von Berechnungen ein größeres Sharing bei der Speicherdarstellung der Terme erzeugt wird.

Verzichtet man auf die bedarfsgesteuerte Generierung der Transitionssysteme, so bietet sich das obige Verfahren an, um eine möglichst kompakte Darstellung des Transitionssystems zu erhalten. Der Speicher, der durch die Löschung der Hash–Tabelle frei wird, kann so zum Beispiel vom Modelchecker genutzt werden. Für manche Anwendungen ist es sogar denkbar, die zweite Hash–Tabelle sowie die Beschriftung der Zustände durch Ausdrücke der Prozeßalgebra zu löschen, um so weiteren Speicher freizumachen.

4. Logiken für verteilte Systeme

Bisher haben wir uns mit Prozeßspezifikationen und deren Semantik beschäftigt. Das Ziel von TRUTH ist, eine Möglichkeit zur Verifikation verteilter Systeme zu schaffen. Dazu müssen wir in der Lage sein, Eigenschaften eines Systems zu beschreiben, die für dessen Korrektheit notwendig erfüllt sein müssen. Dabei muß die Beschreibung dieser Eigenschaften möglichst „unscharf" möglich sein, um verschiedene Implementierungen dieser Eigenschaften zuzulassen.

Es gibt unterschiedliche Ansätze zur Formulierung solcher Eigenschaften. So ist es möglich, verschiedene Äquivalenzen von Transitionssystemen zu definieren und den Begriff von Korrektheit über diesen Ansatz zu konkretisieren. Es existieren darüber hinaus verschiedene andere Ansätze.

In diesem Kapitel wird der in TRUTH gewählte Ansatz der Beschreibung von Eigenschaften mit Hilfe von Logiken vorgestellt [CES86]. Wir beschreiben verschiedene Logiken, die sich zu diesem Zweck eignen. Darüber hinaus stellen wir einen lokalen Modelchecking–Algorithmus von Cleaveland vor, der in TRUTH implementiert wurde. Wir beschreiben einige Aspekte dieser Implementierung und schließen mit Betrachtungen zum Laufzeitverhalten der Implementierung.

4.1. Temporale Logiken

Ein inzwischen sehr verbreiteter Ansatz zur Formulierung von Eigenschaften verteilter Systeme bedient sich verschiedener Logiken, die in der Lage sind, Eigenschaften von Transitionssystemen auszudrücken. Eine einfache Logik die sich dazu eignet, ist die Hennessy–Milner Logik [HM80]. Diese erlaubt es jedoch nur, Aussagen über endliche Pfade zu treffen.

Aus diesem Grund benutzt man andere Logiken, die die Hennessy–Milner Logik um temporale Operatoren erweitern, wodurch auch Aussagen über unendliche Pfade möglich werden.

Es existieren drei Temporal–Logiken, die sich auf diesem Gebiet als Quasi–Standard herausgebildet haben: LTL, CTL, CTL* und der μ–Kalkül. Diese unterscheiden sich in der Ausdrucksstärke ihrer Temporal–Operatoren. LTL und CTL lassen sich dabei als Fragmente von CTL* auffassen, wobei LTL eine Temporalzeit–Logik ist, das

heißt, nur Aussagen über Läufe des Systems möglich sind. CTL und CTL* hingegen sind Baumzeit–Logiken, die auch Aussagen über Verzweigungen im Lauf des Systems möglich machen.

4.1.1. Die Logik CTL*

Wir erklären nun zunächst die Syntax von CTL* sowie von CTL und LTL als Einschränkungen von CTL* in Anlehnung an [Eme96].

Definition 4.1.1 (CTL*, CTL, LTL)
Es bezeichne $\mathcal{P}$ eine abzählbare Menge von atomaren Propositionen. CTL ist die Menge aller Zustandsformeln, die induktiv gemäß der folgenden Regeln gebildet sind:*

S1 Jede Proposition $P \in \mathcal{P}$ ist eine Zustandsformel.

S2 Wenn p, q Zustandsformeln sind, so auch $p \wedge q, \neg p$.

S3 Wenn p eine Pfadformel ist, so sind Ep, Ap Zustandsformeln.

P1 Jede Zustandsformel ist eine Pfadformel.

P2 Wenn p, q Pfadformeln sind, so auch $p \wedge q, \neg p$.

P3 Wenn p, q Pfadformeln sind, so auch Xp, pUq.

Wir erhalten CTL, indem wir keine Schachtelung der Operatoren X und U zulassen. Formal ersetzen wir dafür P1-3 durch:

P0 Wenn p, q Zustandsformeln sind, so sind Xp, pUq Pfadformeln.

Schließlich definieren wird LTL als die Menge der Pfadformeln, die sich gemäß der Regeln S1,P1-3 ergeben.

Wie üblich führt man weitere Operatoren als abkürzende Schreibweisen ein: Wir schreiben $p \vee q$ anstelle von $\neg(\neg p \wedge \neg q)$, sowie $p \Rightarrow q$ für $\neg p \vee q$. Darüber hinaus stehe Fp für $trueUp$ und Gp für $\neg F \neg p$. Schließlich schreiben wir $\overset{\infty}{F}p$ für GFp und $\overset{\infty}{G}p$ für FGp.

Wir interpretieren Formeln von CTL* über einem beschrifteten Transitionssystem. Dazu muß natürlich eine Interpretation der atomaren Propositionen vorgenommen werden.

Definition 4.1.2 (Kripkestruktur)
Ein beschriftetes Transitionssystem $\mathcal{M} = \langle S, Act, \rightarrow \rangle$ zusammen mit einer Interpretation $L : \mathcal{P} \rightarrow 2^S$ heißt Kripkestruktur.

Die Abbildung L ordnet dabei jeder atomaren Proposition die Menge der Zustände zu, an denen sie erfüllt ist.

Zur Interpretation von Formeln aus CTL* benötigen wir noch den Begriff eines Pfades in einem Transitionssystem.

Definition 4.1.3
Ein Pfad in $\mathcal{M}$ ist eine (möglicherweise unendliche) Folge $(s_0, s_1, s_2, \dots)$, wobei stets $s_i \Rightarrow s_{i+1}$ gelten muß. $\Rightarrow$ ist dabei wie in Definition 3.1.7 als „Vereinigung“ von $\rightarrow$ erklärt. Mit x^i bezeichnen wir den Suffix–Pfad von x, der mit s_i beginnt.

Wir erklären nun, was wir darunter verstehen wollen, daß eine Kripkestruktur $\mathcal{M}$ eine CTL*–Formel an einem Zustand s_0 erfüllt.

Definition 4.1.4 (Semantik von CTL*)
Wir definieren $\models$ induktiv über die Syntax von CTL wie folgt:*

S1 $\mathcal{M}, s_0 \models P$, *gdw.* $s_0 \in L(P)$.

S2 $\mathcal{M}, s_0 \models p \wedge q$, *gdw.* $\mathcal{M}, s_0 \models p$ *und* $\mathcal{M}, s_0 \models q$.

$\mathcal{M}, s_0 \models \neg p$, *gdw. nicht* $\mathcal{M}, s_0 \models p$.

S3 $\mathcal{M}, s_0 \models Ep$, *gdw. es existiert ein Pfad* $x = (s_0, s_1, s_2, \dots)$ *mit* $\mathcal{M}, x \models p$.

$\mathcal{M}, s_0 \models Ap$, *gdw. für alle Pfade* $x = (s_0, s_1, s_2, \dots)$ *mit* $\mathcal{M}, x \models p$.

P1 $\mathcal{M}, x \models p$, *gdw.* $x = (s_0, s_1, s_2, \dots)$ *und* $\mathcal{M}, s_0 \models p$.

P2 $\mathcal{M}, x \models p \wedge q$, *gdw.* $\mathcal{M}, x \models p$ *und* $\mathcal{M}, x \models q$.

$\mathcal{M}, x \models \neg p$, *gdw. nicht* $\mathcal{M}, x \models p$.

P3 $\mathcal{M}, x \models pUq$, *gdw.* $\exists i\ [\mathcal{M}, x^i \models q$ *und* $\forall j\ (j < i \Rightarrow \mathcal{M}, x^j \models p)]$.

$\mathcal{M}, x \models Xp$, *gdw.* x *hat mindestens die Länge 2 und* $\mathcal{M}, x^1 \models p$.

Die Semantik von CTL– und LTL–Formeln ergibt sich durch die Anwendung der entsprechenden Regeln. Eine Formel von LTL ist dabei eine reine Pfadformel und macht Aussagen über einen Pfad des Systems. Wir definieren die Modelbeziehung für eine LTL–Formeln p und einen Zustand s indem wir die Formel für allen von s ausgehenden Pfaden betrachten.

$$\mathcal{M}, s \models p, \text{ gdw. für alle Pfade } x = (s, s_1, s_2, \dots) \text{ gilt } \mathcal{M}, x \models p$$

Ausdrucksstärke

Mit diesen Logiken ist es möglich, eine Reihe interessanter Eigenschaften auszudrücken, dazu gehören *Safety–*, *Liveness–* und *Fairness–*Eigenschaften [Eme96].

Eine Safety–Eigenschaft beschreibt, daß eine Eigenschaft p niemals verletzt. Dies läßt sich mit der Formel AGp ausdrücken. „Für alle Pfade gilt, daß immer p gilt.“ Damit

lassen sich Safety–Eigenschaften in allen drei Logiken ausdrücken. Für CTL sieht man dies, indem man AGp äquivalent in $\neg E(trueU\neg p)$ umformt.

Eine Liveness–Eigenschaft beschreibt, daß eine Eigenschaft p immer irgendwann eintreten kann. Dies ist zum Beispiel durch die Formel $AGEFp$ beschrieben. „Für alle Pfade gilt, daß immer gilt, daß es einen Pfad gibt, auf dem irgendwann p gilt." Diese Formel ist zwar in CTL, jedoch nicht mehr in LTL.

Es gibt verschiedene Fairness-Eigenschaften. Eine unbedingte Fairness–Eigenschaft fordert zum Beispiel, daß p auf allen Pfaden unendlich oft auftritt. Dies läßt sich durch die Formel $A\overset{\infty}{F}p$ ausdrücken, die zwar in LTL, jedoch nicht in CTL liegt.Es läßt sich sogar zeigen, daß es unmöglich ist, eine solche Eigenschaft in CTL auszudrücken [EC81, EC86].

4.2. Der μ-Kalkül

Unter den Modallogiken zur Verifikation verteilter Systeme nimmt der μ–Kalkül [Koz82] eine Sonderstellung ein. Dieses ist durch zwei Tatsachen begründet. Er verfügt über keine temporalen Konstrukte, sondern erweitert die Logik um Fixpunktkonstrukte, die eine induktive Definition von Mengen erlauben. Darüber hinaus ist es die stärkste der hier betrachteten Logiken in dem Sinne, daß für alle LTL–, CTL– und auch CTL*–Formeln äquivalente Formeln des μ–Kalküls existieren. Darüber hinaus ist diese Übersetzung effektiv berechenbar.

4.2.1. Syntax von Lμ

Zunächst geben wir die Syntax des μ–Kalküls an. Wir wählen hier eine minimale Variante und erklären die restliche Operatoren als abkürzende Schreibweisen.

Definition 4.2.1 (Syntax von $L\mu$)
Es sei $\mathcal{P}$ eine Menge atomarer Propositionen, $\mathcal{X}$ eine Menge von Variablen und Act eine Menge von Aktionen. Die Menge $L\mu$ der Formeln des μ–Kalküls ist induktiv definiert durch:

$$\begin{aligned} \mathrm{tt} &\in L\mu \\ P &\in L\mu \quad \textit{falls } P \in \mathcal{P} \\ X &\in L\mu \quad \textit{falls } X \in \mathcal{X} \end{aligned}$$

Sind Φ und Ψ Formeln aus $L\mu$ so gilt:

$$\begin{aligned} \neg\Phi &\in L\mu \\ \Phi \vee \Psi &\in L\mu \\ \langle\alpha\rangle\Phi &\in L\mu \quad \textit{falls } \alpha \in Act \\ \nu X.\Phi &\in L\mu \quad \textit{falls } X \textit{ in } \Phi \textit{ nur positiv auftritt} \end{aligned}$$

Eine Variable tritt dabei positiv in einer Formel auf, wenn sie sich im Geltungsbereich einer geraden Anzahl von Negationszeichen befindet, andernfalls tritt sie negativ auf.

Die Fixpunktoperatoren binden die Variablen in üblicher Weise. Ungebundene Variablen nennen wir auch freie Variablen. *Eine Formel ohne freie Variablen heißt geschlossene Formel oder* Satz.

Eine Formel die mit einem Fixpunktoperator beginnt, nennen wir rekursive *Formel.*

Mit $\prec$ bezeichnen wir die strikte Unterformel–Relation auf $L\mu$.

In üblicher Weise benutzen wir $\Phi \wedge \Psi$ als Abkürzung für $\neg(\neg\Phi \vee \neg\Psi)$, ff für $\neg\mathrm{tt}$, $[\alpha]\Phi$ für $\neg\langle\alpha\rangle\neg\Phi$ und $\mu X.\Phi$ für $\neg\nu X.\neg(\Phi[\neg X/X])$.

4.2.2. Semantik von Lμ

Wir interpretieren Formeln aus $L\mu$ über Kripkestrukturen. Da wir die Semantik induktiv anhand der Syntax erklären, müssen wir mit freien Variablen umgehen. Wir tun dies mit der Hilfe von Umgebungen, welcher jeder Variablen eine Menge von Zuständen zuordnet. Für Sätze wird die Wahl dieser Abbildung keine Rolle spielen, da sie keine freien Variablen beinhalten.

Definition 4.2.2 (Semantik von $L\mu$)
Es sei $\mathcal{M} = \langle S, Act, \rightarrow\rangle$ ein beschriftetes Transitionssystem und $e : \mathcal{X} \to 2^S$ sowie $L : \mathcal{P} \to 2^S$ Abbildungen. Die die Semantik $[\![-]\!]e$ für Formeln aus $L\mu$ ist induktiv definiert durch:

$$
\begin{aligned}
[\![\mathrm{tt}]\!]e &:= S \\
[\![P]\!]e &:= L(P) \quad \textit{für } P \in \mathcal{P} \\
[\![X]\!]e &:= e(X) \quad \textit{für } X \in \mathcal{X} \\
[\![\neg\Phi]\!]e &:= S \setminus [\![\Phi]\!]e \\
[\![\Phi \vee \Psi]\!]e &:= [\![\Phi]\!]e \cup [\![\Psi]\!]e \\
[\![\langle\alpha\rangle\Phi]\!]e &:= \pi_\alpha([\![\Phi]\!]e) \quad \textit{mit} \quad \pi_\alpha(A) = \{s' \mid \exists s \in A : s' \xrightarrow{\alpha} s\} \\
[\![\nu X.\Phi]\!]e &:= \bigcup\{A \subseteq S \mid A \subseteq [\![\Phi]\!]e[X \mapsto A]\}
\end{aligned}
$$

Für Sätze ist die Wahl der ursprünglichen Umgebung nicht von Bedeutung, in diesem Fall schreiben wir auch $[\![\Phi]\!]$ und lassen die Umgebung weg.

Für $s \in [\![\Phi]\!]$ vereinbaren wir als alternative Schreibweise $\mathcal{M}, s \models \Phi$.

Natürlich ist die semantische Funktion auch abhängig von der gewählten Kripkestruktur. Es wird jedoch immer aus dem Kontext klar sein, welche Kripkestruktur gerade betrachtet wird und deshalb verzichten wir auf eine explizite Angabe aus Gründen notationeller Klarheit.

4.2.3. Übersetzungen von LTL, CTL und CTL*

Wie bereits erwähnt ist es möglich, zu jeder CTL*–Formel, und damit auch zu jeder Formel von LTL und CTL, effektiv eine äquivalente Formel des μ–Kalküls anzugeben. Da CTL* keine beschriftete Übergangsrelation $\rightarrow$ verwendet, modifizieren wir die Syntax des μ–Kalküls zu diesem Zweck: Wir entfernen die parametrisierte Modalität $\langle\alpha\rangle\Phi$ und ersetzen sie durch eine Modalität $\langle -\rangle\Phi$, die wir mit Hilfe der unbeschrifteten Übergangsrelation $\Rightarrow$ interpretieren.

Nun können zum Beispiel CTL–Formeln nach dem folgenden Schema übersetzt werden:

$$\begin{aligned} EFP &\equiv \mu Z.P \vee \langle -\rangle Z \\ AGP &\equiv \nu Z.P \wedge [-]Z \\ AFP &\equiv \mu Z.P \vee [-]Z \\ EGP &\equiv \nu Z.P \wedge \langle -\rangle Z \\ A(PUQ) &\equiv \mu Z.Q \vee (P \wedge [-]Z) \\ E(PUQ) &\equiv \mu Z.Q \vee (P \wedge \langle -\rangle Z) \end{aligned}$$

Dies sind sehr naheliegende Fixpunkt–Charakterisierungen, die sich leicht als korrekt nachweisen lassen [EC81]. Für CTL* läßt sich keine so einfache Übersetzung angeben. Es existiert jedoch ein effektives Verfahren, das die Zuordnung durchführt [EC86].

Es ist also möglich, im μ–Kalkül viele interessante Eigenschaften auszudrücken. Die Aussagekraft des μ–Kalküls geht sogar über die Aussagekraft von CTL* hinaus. Dies zeigt sich darin, daß alle Übersetzungen von Formeln aus CTL* nur das Fragment $L\mu_2$ benötigen, das in Definition 4.3.5 eingeführt wird. In [Bra96] wird jedoch gezeigt, daß die Alternierungstiefe des μ–Kalküls unbeschränkt ist. Also gibt es Formeln des μ–Kalküls, die sich nicht durch Formeln in $L\mu_2$ und damit auch nicht in CTL* ausdrücken lassen. Darüber hinaus lassen sich verschiedene Eigenschaften im μ–Kalkül sehr elegant beschreiben. All dies macht den μ–Kalkül zu einer geeigneten Logik zur Beschreibung von Eigenschaften verteilter Systeme.

4.3. Modelchecking für Lμ

Modelchecking ist die Technik, mit der wir nun entscheiden wollen, ob eine Spezifikation eine Formel erfüllt.

Definition 4.3.1 (Modelchecking)
Die Menge

$$\{(\mathcal{T}, s, \Phi) \mid \mathcal{T} \models_s \Phi\},$$

wobei $\Phi \in L\mu$ ein Satz, $\mathcal{T} = \langle S, Act, \rightarrow\rangle$ ein endliches beschriftetes Transitionssystem und $s \in S$ ist, heißt das Modelchecking–Problem *des μ–Kalküls.*

4.3.1. Entscheidbarkeit

Um die Entscheidbarkeit des Modelcheckings für den μ–Kalkül nachzuweisen, benötigen wir elementare Halbordnungs– und Verbands–Theorie. Diese sind in Anhang A dargestellt.

Lemma 4.3.2 *Die durch eine Formel Φ von $L\mu$ über dem vollständigen Verband $\langle 2^S, \subseteq, \cup, \cap \rangle$ folgendermaßen induzierte Funktion*

$$\begin{aligned} \varphi : 2^S &\to 2^S \\ A &\mapsto [\![\Phi]\!]e[X \mapsto A] \end{aligned}$$

ist monoton.

Der Beweis für dieses Lemma ist eine einfache Induktion über den Aufbau von Formeln aus $L\mu$. Für den Fall des Fixpunktoperators benötigt man die Einschränkung, daß die gebundene Variable nur syntaktisch positiv auftreten darf.

Damit ergibt sich die Semantik einer Fixpunktformel aus $L\mu$ als maximaler Fixpunkt der induzierten Funktion. Dieser ist nach dem Satz von Tarski über endlichen Verbänden effektiv berechenbar durch eine Iteration der Funktion ausgehend vom größten Element des Verbandes.

Aus diesen Betrachtungen ergibt sich nun unmittelbar die Entscheidbarkeit des Modelcheckings.

Satz 4.3.3 (Entscheidbarkeit des Modelcheckings)
Für den μ–Kalkül ist Modelchecking über endlichen Kripkestrukturen entscheidbar.

Beweis: Für eine endliche Kripkestruktur sind alle Mengenoperationen, die zur Berechnung der Semantik eines Satzes aus $L\mu$ notwendig sind, effektiv berechenbar. Der Satz von Tarski liefert hierbei die effektive Berechenbarkeit der Semantik von Fixpunktformeln als größte Fixpunkte einer monotonen Funktion über einem vollständigen Verband. □

Da die Übersetzungen von LTL, CTL und CTL* in den μ–Kalkül effektiv berechenbar waren, haben wir damit auch gezeigt, daß Modelchecking über endlichen Kripkestrukturen für diese Logiken entscheidbar ist.

4.3.2. Komplexität

Neben der Entscheidbarkeit ist man natürlich für die Anwendung des Modelcheckings zur Verifikation auch an dessen Zeit– und Platzkomplexität interessiert.

Für die Schwierigkeit des Modelcheckings spielt neben der Größe des betrachteten Transitionssystems auch die Struktur des Satzes ein wesentliche Rolle, insbesondere ist die Alternierungstiefe einer Formel des μ–Kalküls von Bedeutung.

Definition 4.3.4
Eine Formel $\Phi \in L\mu$ ist in positiver Normalform *(PNF), wenn sie Negationen höchstens vor atomaren Propositionen enthält und jede Variable nur von einem Fixpunkt–Operator gebunden ist.*

Offensichtlich läßt sich jede Formel durch Umbenennung von Variablen und „nach innen schieben" der Negationszeichen in PNF bringen. Allerdings kommt man in diesem Fall nicht mehr mit der reduzierten Menge von Junktoren aus, sondern benötigt alle Junktoren. Wir benötigen die PNF nur für die folgende Definition.

Definition 4.3.5 (Alternierungstiefe)
Die Alternierungstiefe $ad(\Phi)$ *einer Formel $\Phi \in L\mu$ in positiver Normalform ist induktiv definiert durch:*

$$ad(\mathrm{tt}) = ad(\mathrm{ff}] = ad(X) = ad(P) = 0$$
$$ad(\neg\Phi) = ad(\Phi) = ad(\langle\alpha\rangle\Phi) = ad([\alpha]\Phi) = ad(\Phi)$$
$$ad(\Phi \vee \Psi) = ad(\Phi \wedge \Psi) = \max\{ad(\Phi), ad(\Psi)\}$$
$$ad(\nu X.\Phi) = 1 + \max\{ad(\mu Y.\Psi) \mid \mu Y.\Psi \prec \Phi \textit{ und } Y \textit{ tritt frei in } \Psi \textit{ auf}\}$$
$$ad(\mu X.\Phi) = 1 + \max\{ad(\nu Y.\Psi) \mid \nu Y.\Psi \prec \Phi \textit{ und } Y \textit{ tritt frei in } \Psi \textit{ auf}\}$$

$L\mu_k$ bezeichnet die Menge aller Formeln aus $L\mu$ mit einer Alternierungstiefe von höchstens k.

Die Alternierungstiefe einer Formel des μ–Kalküls ist ein wichtiges (syntaktisches) Maß für die Aussagekraft der Formel. So ist zum Beispiel die Alternierungshierarchie des μ–Kalküls unbeschränkt [Bra96]. Das heißt, daß sich durch Formeln mit höherer Alternierungstiefe immer neue Eigenschaften beschreiben lassen.

Der Modelchecking–Algorithmus, wie er in den Ausführungen zur Entscheidbarkeit des Modelcheckings angedeutet wurde, hat einen großen Nachteil. Er ist unnötig ineffizient, da bei geschachtelten Fixpunkten, der innere Fixpunkt neu berechnet wird, egal, ob dies notwendig ist oder nicht. Dadurch ergibt sich ein Algorithmus, dessen Zeitaufwand zwar polynomial in der Größe des betrachteten Transitionssystems, dagegen jedoch exponentiell in der Schachtelungstiefe der zu überprüfenden Formel ist.

Mit besseren Algorithmen gelingt es, die Zeitkomplexität zu drücken, so daß sie immer noch polynomial in der Größe des Transitionssystems aber nur noch exponentiell in der Alternierungstiefe der Formel ist. Dabei ist bekannt, daß Modelchecking für den μ–Kalkül in NP $\cap$ co–NP liegt, was die Existenz eines polynomialen Algorithmus nahelegt. Dennoch sind bisher keine Algorithmen bekannt, die über eine solche Komplexität verfügen. In [Eme97] findet man eine Darstellung verschiedener Modelchecking–Algorithmen für $L\mu$.

Eine Tabelle mit der Angabe der Komplexitätsklasse und einer Abschätzung für die verschiedenen Logiken findet sich in in der folgenden Tabelle nach [Eme96]. $|\mathcal{M}|$ bezeichnet

die Größe der Kripkestruktur während $|\Phi|$ für die Länge der zu überprüfenden Formel steht. Der Zeitaufand bezieht sich dabei auf die der besten bekannten Algorithmen.

Logik	Komplexitätsklasse	Zeitaufwand
LTL	PSPACE–vollständig	$O(\|\mathcal{M}\| \cdot \exp(\|\Phi\|))$
CTL	P-vollständig	$O(\|\mathcal{M}\| \cdot \|\Phi\|)$
CTL*	PSPACE–vollständig	$O(\|\mathcal{M}\| \cdot \exp(\|\Phi\|))$
$L\mu_k$	NP $\cap$ co-NP	$(\|\mathcal{M}\| \cdot \|\Phi\|)^{O(k)}$

Obwohl die Zeitkomplexität also bei den meisten Logiken im schlimmsten Fall exponentiell in der Größe der Formel wächst, bietet Modelchecking ein geeignetes Verfahren zur Verifikation verteilter Systeme. Dies liegt an einer Beobachtung, die oft auch als die „These von Lichtenstein–Pnueli“ bezeichnet wird:

> Obwohl die Komplexität des Modelchecking exponentiell in der Größe der betrachteten Formel wachsen kann, ist das lineare Wachstum in der Größe der betrachteten Struktur von größerer Bedeutung. Dies liegt daran, daß die betrachteten Formeln in Anwendungen relativ einfach sind, während die zugrundeliegenden Strukturen sehr groß werden können.

Eine ähnliche Aussage läßt sich für den μ–Kalkül treffen.

> Obwohl die Alternierungshierarchie des μ–Kalküls unbeschränkt ist, ist Modelchecking für den μ–Kalkül ein praktisch verwendbares Verfahren, da die in der Praxis auftretenden Formeln in $L\mu_1$ und $L\mu_2$ liegen.

Diese These wird gestützt durch die Tatsache, daß die Formeln, die sich als Übersetzungen von CTL und CTL* ergeben, in $L\mu_1$ bzw. $L\mu_2$ liegen.

4.3.3. Globales und lokales Modelchecking

Die bisher betrachteten Modelchecking–Algorithmen sind sogenannte *globale* Algorithmen. Das heißt, sie berechnen die Menge aller Zustände, an denen eine Formel erfüllt ist.

Diese Vorgehensweise zeichnet sich durch recht einfach zu verstehende Algorithmen aus, da es sich hier bei letztlich um mehr oder weniger gute Algorithmen zur Berechnung wechselseitig abhängiger Fixpunkte handelt. Globales Modelchecking hat jedoch für ein System wie TRUTH einen entscheidenden Nachteil: Es wird stets das gesamte Transitionssystem in die Berechnungen mit einbezogen. Dies ist ein sinnvolles Vorgehen, wenn das Transitionssystem in der Anfrage explizit gegeben ist. In unserem Fall ist jedoch das zu betrachtende Transitionssystem nicht explizit sondern implizit in Form einer Spezifikation in einer Prozeßalgebra gegeben. Daraus muß dann zunächst das gesamte Transitionssystem bestimmt werden, um Modelchecking möglich zu machen. Dies hat für die praktische Anwendung einige Nachteile:

- Oftmals ist es möglich zu entscheiden, ob $\mathcal{M}, s \models \Phi$ gilt, ohne das gesamte Transitionssystem zu kennen, da die Gültigkeit am Zustand s nur von einem kleinen Teil des Transitionssystems abhängt.

- Ein System wie TRUTH wird nicht nur zur Verifikation einer Spezifikation benutzt, sondern auch, um eine solche Spezifikation zu entwickeln. Dabei wird eine anfängliche Spezifikation so lange weiterentwickelt, bis sie die gewünschten Eigenschaften erfüllt. Im Rahmen dieses Prozesses wird wiederholt die Gültigkeit einer Formel überprüft. Insbesondere bei Safety–Formeln ist es dabei möglich, daß ein Zustand, an dem die entsprechende Eigenschaft verletzt ist, gefunden wird, ohne das gesamte Transitionssystem zu erzeugen

Lokales Modelchecking zeichnet sich durch eine andere Vorgehensweise aus. Ausgehend von der Anfrage wird versucht, einen Beweis oder eine Widerlegung für $\mathcal{M}, s \models \Phi$ zu finden. Beginnend am Anfangszustand wird dabei nur soviel vom Transitionssystem betrachtet, wie es für die Beantwortung der Anfrage notwendig ist. Dies wird in TRUTH ausgenutzt, indem das Transitionssystem bei Modelchecking bedarfsgesteuert erzeugt wird und somit oftmals der aufwendige Aufbau des gesamten Transitionssystems vermieden wird. Natürlich kann es auch in diesem Fall notwendig werden, das gesamte Transitionssystem zu betrachten.

4.3.4. Tableau–basiertes Modelchecking

In TRUTH ist eine Version des Tableau–basierten Modelchecking–Algorithmus implementiert, der in [Cle90] dargestellt wird. Es handelt sich hierbei um ein lokales Modelchecking Verfahren. Es wird versucht, gesteuert von der Struktur des betrachteten Satzes und ausgehend vom betrachteten Zustand, einen Beweis oder eine Widerlegung für die Anfrage „Gilt $\mathcal{M}, s \models \Phi$?“ zu finden.

Das Tableau–System arbeitet mit Sequenzen der Form $H \vdash_{\mathcal{M}} s \in \Phi$. Dabei ist

- $\mathcal{M}$ die zugrundeliegende Kripkestruktur,

- s ein Zustand aus $\mathcal{M}$,

- Φ eine Formel aus $L\mu$ und

- H eine Menge von Hypothesen der Form $s' : \Gamma$, wobei s' ein Zustand von $\mathcal{M}$ und Γ eine geschlossene rekursive Formel aus $L\mu$.

Wir werden aus Gründen der notationellen Klarheit auf die Angabe der betrachteten Kripkestruktur verzichten, da diese sich aus dem Kontext ergibt.

$$\text{R1}\ \frac{H \vdash s \in \neg\neg\Phi}{H \vdash s \in \Phi} \qquad \text{R2}\ \frac{H \vdash s \in \Phi \vee \Psi}{H \vdash s \in \Phi}$$

$$\text{R3}\ \frac{H \vdash s \in \Phi \vee \Psi}{H \vdash s \in \Psi} \qquad \text{R4}\ \frac{H \vdash s \in \neg(\Phi \vee \Psi)}{H \vdash s \in \neg\Phi, H \vdash s \in \neg\Psi}$$

$$\text{R5}\ \frac{H \vdash s \in \langle\alpha\rangle\Phi}{H \vdash s' \in \Phi} \quad (s' \in \{t \mid s \xrightarrow{\alpha} t\})$$

$$\text{R6}\ \frac{H \vdash s \in \neg\langle\alpha\rangle\Phi}{H \vdash s_1 \in \neg\Phi, H \vdash s_2 \in \neg\Phi, \ldots} \quad (\{s_1, s_2, \ldots\} = \{s' \mid s \xrightarrow{\alpha} s'\})$$

$$\text{R7}\ \frac{H \vdash s \in \nu X.\Phi}{H' \cup \{s : \nu X.\Phi\} \vdash s \in \Phi[\nu X.\Phi/X]} \quad (s : \nu X.\Phi \notin H)$$

$$\text{R8}\ \frac{H \vdash s \in \neg\nu X.\Phi}{H' \cup \{s : \nu X.\Phi\} \vdash s \in \neg\Phi[\nu X.\Phi/X]} \quad (s : \nu X.\Phi \notin H)$$

$$\text{mit } H' = H \setminus \{s' : \Gamma \mid \nu X.\Phi \prec \Gamma\}.$$

Abbildung 4.1: Regeln des Tableau–Systems

Definition 4.3.6
Ein Tableau für eine Sequenz $\sigma = H \vdash s \in \Phi$ ist ein maximaler Beweisbaum, dessen Wurzel σ ist, und der gemäß der Regeln in Abbildung 4.1 aufgebaut ist.

Ein Blatt $\sigma = H \vdash s \in \Phi$ eines Tableaus heißt erfolgreich, *wenn es eine der folgenden Bedingungen erfüllt:*

1. $\Phi = \mathrm{tt}$
2. $\Phi = P \wedge s \in L(P)$ *für* $P \in \mathcal{P}$
3. $\Phi = \neg\langle\alpha\rangle\Psi$ *für beliebiges* α *und* Ψ
4. $\Phi = \nu X.\Psi$ *für beliebiges* X *und* Ψ

Ein Tableau heißt erfolgreich, *wenn alle seine Blätter erfolgreich sind.*

Man beachte, daß $H \vdash s \in \neg\langle\alpha\rangle\Psi$ nur dann ein Blatt eines Tableaus ist, wenn s keine α–Nachfolger besitzt. $H \vdash s \in \nu X.\Psi$ ist nur dann Blatt eines Tableaus, wenn $s : \nu X.\Psi \in H$ gilt.

Bemerkenswert ist die Tatsache, daß in den Tableau–Regeln $R7$ und $R8$ die Hypothesenmengen um gewisse Hypothesen reduziert werden. Dies geschieht, damit das Verfahren

$$\text{DR1}\ \frac{H \vdash s \in \Phi \wedge \Psi}{H \vdash s \in \Phi, H \vdash s \in \Psi}$$

$$\text{DR2}\ \frac{H \vdash s \in [\alpha]\Phi}{H \vdash s_1 \in \Phi, H \vdash s_2 \in \Phi, \ldots} \quad (\{s_1, s_2, \ldots\} = \{s' \mid s \xrightarrow{\alpha} s'\})$$

$$\text{DR3}\ \frac{H \vdash s \in \mu X.\Phi}{H' \cup \{s : \neg\mu X.\Phi\} \vdash s \in \Phi[\mu X.\Phi/X]} \quad (s : \neg\mu X.\Phi \notin H)$$

$$\text{mit } H' = H \setminus \{s' : \Gamma \mid \mu X.\Phi \prec \Gamma\}$$

Abbildung 4.2: Abgeleitete Tableau–Regeln

auch bei alternierenden Fixpunkten korrekt arbeitet. In einem solchen Fall muß die Berechnung eines Fixpunktes, der von einem äußeren Fixpunkt abhängt, für jede Iteration der Berechnung des äußeren Fixpunkts neu vorgenommen werden. Im Tableau–System geschieht dies durch die Verkleinerung der Hypothesenmenge um gewisse Hypothesen.

Für die als Abkürzungen eingeführten Junktoren ergeben sich analoge Tableauregeln aus den bereits genannten. Diese sind in Abbildung 4.2 dargestellt.

Zusätzlich müssen wir die Definition eines erfolgreichen Blattes um eine weitere Bedingung erweitern.

5. $\Phi = [\alpha]\Psi$ für beliebiges α und Ψ

Wiederum ist $H \vdash s \in [\alpha]\Psi$ nur dann Blatt eines Tableaus, wenn s keinen α–Nachfolger besitzt.

Ein erfolgreiches Tableau für einen Beispiel–Satz ist in Abbildung 4.3 dargestellt. Die mit $(*)$ bezeichneten Schritte sind solche, in denen die Hypothesen–Menge reduziert wird.

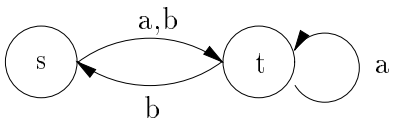
Entscheidend für die Anwendbarkeit des Tableau–Verfahrens ist der folgende Satz.

Satz 4.3.7
Für eine endliche Kripkestruktur $\mathcal{M}$ ist jedes Tableau für jede beliebige Sequenz $\sigma = H \vdash s \in \Phi$ endlich.

Beweisidee: Es ist möglich, eine Ordnung für Sequenzen zu definieren, so daß sich eine Sequenz in einem Tableau–Schritt echt vergrößert. Außerdem zeigt man für diese Ordnung, daß in ihr über endlichen Kripkestrukturen keine unendlichen aufsteigenden Ketten existieren. Details finden sich in [Cle90]. □

Neben der Termination des Algorithmus sind natürlich seine Korrektheit und Vollständigkeit von wesentlicher Bedeutung. Zur Beantwortung dieser Fragen benötigen wir

Das Transitionssystem:



Wir benutzen die folgenden Abkürzungen für auftretende Formeln. Dabei ist B die Abwicklung von A sowie D die Abwicklung von C.

$$A \equiv \nu X.(\langle a\rangle A \wedge \nu Y.\langle b\rangle Y)$$
$$B \equiv \langle a\rangle A \wedge \nu Y.\langle b\rangle Y$$
$$C \equiv \nu Y.\langle b\rangle Y$$
$$D \equiv \langle b\rangle C$$

$$\dfrac{\dfrac{\dfrac{

2. *$x \in \nu\varphi$ genau dann, wenn $x \in \nu\varphi_{\{x\}}$.*

3. *Unter der Annahme $x \in \nu\varphi$ gilt $\nu\varphi = \nu\varphi_{\{x\}}$.*

Beweis: Die erste Aussage folgt trivial aus der Definition von Monotonie und von $\varphi_{S'}$. Die zweite Aussage beweisen wir, indem wir beide Implikationen zeigen.

„$\Rightarrow$“:

$$\begin{aligned} x \in \nu\varphi &\Leftrightarrow x \in \bigcup\{A \subseteq S \mid A \subseteq \varphi(A)\} \\ &\Leftrightarrow \exists S' \subseteq S : x \in S' \wedge S' \subseteq \varphi(S') \end{aligned}$$

Mit $x \in S'$ gilt $S' = \{x\} \cup S'$, und nach Definition $\varphi(S') = \varphi_{\{x\}}(S')$. Also gilt $S' \subseteq \varphi_{\{x\}}(S')$. Damit folgt $x \in \nu\varphi_{\{x\}}$.

„$\Leftarrow$“:

$$\begin{aligned} x \in \nu\varphi_{\{x\}} &\Leftrightarrow x \in \bigcup\{A \subseteq S \mid A \subseteq \varphi_{\{x\}}(A)\} \\ &\Leftrightarrow \exists S' \subseteq S : x \in S' \wedge S' \subseteq \varphi_{\{x\}}(S') \end{aligned}$$

Mit $\varphi_{\{x\}}(A) = \varphi(A \cup \{x\})$ folgt $\varphi_{\{x\}}(A) = \varphi(A)$ und damit $x \in \nu\varphi$.

Um die dritte Aussage zu beweisen, zeigen wir die wechselseitige Inklusion der Mengen, indem wir zeigen, daß es sich jeweils um einen Fixpunkt der anderen Funktion handelt. Da wir maximale Fixpunkte betrachten, ergibt sich die Inklusion und damit die Gleichheit der Menge.

„$\subseteq$“:

$$\begin{gathered} \nu\varphi = \varphi(\nu\varphi) = \varphi(\nu\varphi \cup \{x\}) = \varphi_{\{x\}}(\nu\varphi) \\ \text{also ist } \nu\varphi \subseteq \nu\varphi_{\{x\}}. \end{gathered}$$

„$\supseteq$“:

$$\begin{aligned} \nu\varphi_{\{x\}} &= \varphi_{\{x\}}(\nu\varphi_{\{x\}}) \\ &= \varphi(\nu\varphi_{\{x\}} \cup \{x\}) \\ &= \varphi(\nu\varphi_{\{x\}}) \quad \text{da } x \in \nu\varphi_{\{x\}} \text{ wegen (2).} \end{aligned}$$

□

Definition 4.3.10
Es sei H eine Hypothesen–Menge und $\Phi \in L\mu$ ein Formel. Die Spur *von H bezüglich Φ ist definiert durch*

$$H\lceil\Phi := \{s \mid s : \Phi \in H\}.$$

Mit Hilfe dieser Definition können wir nun die relativierte Semantik definieren.

Definition 4.3.11
Es sei $\mathcal{M} = \langle S, Act, \rightarrow \rangle$ eine Kripkestruktur mit Interpretationsfunktion L. Die relativierte Semantik *einer Formel $\Phi \in L\mu$ über $\mathcal{M}$ bezüglich einer Hypothesen–Menge H ist induktiv definiert durch*

$$\begin{aligned}
[\![\mathrm{tt}]\!]^H e &:= S \\
[\![P]\!]^H e &:= L(P) \quad \textit{für } P \in \mathcal{P} \\
[\![X]\!]^H e &:= e(X) \quad \textit{für } X \in \mathcal{X} \\
[\![\neg\Phi]\!]^H e &:= S \setminus [\![\Phi]\!]^H e \\
[\![\Phi \vee \Psi]\!]^H e &:= [\![\Phi]\!]^H e \cup [\![\Psi]\!]^H e \\
[\![\langle\alpha\rangle\Phi]\!]^H e &:= \pi_\alpha([\![\Phi]\!]^H e) \quad \textit{mit} \quad \pi_\alpha(A) = \{s' \mid \exists s \in A : s' \xrightarrow{\alpha} s\} \\
[\![\nu X.\Phi]\!]^H e &:= (\nu\varphi_{S'}) \cup S'
\end{aligned}$$

mit

$$\begin{aligned}
\varphi(S) &:= [\![\Phi]\!]^H e[X \mapsto S] \\
S' &:= H\lceil \nu X.\Phi
\end{aligned}$$

Offensichtlich gilt für einen beliebigen Satz $\Phi \in L\mu$: $[\![\Phi]\!]^\emptyset = [\![\Phi]\!]$.
Mit Hilfe der relativierten Semantik können wir nun konkretisieren, was unter Korrektheit und Vollständigkeit der Tableau–Methode verstanden wird.

Definition 4.3.12 (Gültige Sequenzen)
Eine Sequenz $\sigma = H \vdash s \in \Phi$ heißt gültig, *wenn gilt $s \in [\![\Phi]\!]^H$.*

Satz 4.3.13 (Korrektheit)
Wenn $\sigma = H \vdash s \in \Phi$ über einem endlichen Transitionssystem ein erfolgreiches Tableau besitzt, so ist σ gültig.

Beweisidee: Jedes Tableau für σ ist endlich. Induktiv über die Höhe des Tableaus zeigt man, daß die Wurzel eines erfolgreichen Tableaus gültig ist. Für den Induktionsanfang sieht man leicht, daß ein erfolgreiches Blatt immer gültig ist. Für den Induktionsschluß beweist man für jede Tableau–Regel, daß sich die Gültigkeit aller Tochterknoten auf den Vaterknoten überträgt. Details finden sich in [Cle90]. □
Die Vollständigkeit liefert uns der folgende Satz.

Satz 4.3.14 (Vollständigkeit)
$H \vdash s \in \Phi$ besitzt genau dann ein erfolgreiches Tableau, wenn $H \vdash s \in \neg\Phi$ kein erfolgreiches Tableau besitzt.

Beweisidee: Induktion über die Höhe des Tableaus. Details wiederum in [Cle90]. □
Damit können wir einen ersten, naiven Tableau–Algorithmus definieren, wie er in Abbildung 4.4 dargestellt ist. Dieser beginnt mit einer leeren Hypothesen–Menge und führt das Tableau–Verfahren in einer deterministischen Weise durch.

```
check s Φ = check' ∅ ⊢ s ∈ Φ
check' H ⊢ s ∈ Φ =
  case Φ of
  tt                          →  True
  ff                          →  False
  P | P ∈ 𝒫                   →  s ∈ L(P)
  X | X ∈ 𝒳                   →  error
  ¬Φ'                         →  not check' H ⊢ s ∈ Φ'
  Φ1 ∨ Φ2                     →  check' H ⊢ s ∈ Φ1 || check' H ⊢ s ∈ Φ2
  Φ1 ∧ Φ2                     →  check' H ⊢ s ∈ Φ1 && check' H ⊢ s ∈ Φ2
  ⟨α⟩Φ'                       →  or [check' H ⊢ s' ∈ Φ' | s' <- {t | s -α-> t}]
  νX.Φ' | s : νX.Φ' ∈ H       →  True
  νX.Φ' | s : νX.Φ' ∉ H       →  let H' = {s' : Γ | Φ ⋠ Γ}
                                 in check' H' ∪ {s : Φ} ⊢ s ∈ Φ'[Φ/X]
```

Abbildung 4.4: Ein naiver Modelchecking Algorithmus

4.3.5. Effizienzbetrachtungen

Der Algorithmus in Abbildung 4.4 hat einige Nachteile, die seinen Ansatz in einem System wie TRUTH verbieten. Der gravierenste davon ist sicherlich seine Ineffizienz, denn selbst bei Formeln ohne geschachtelte Fixpunkte kann dieser Algorithmus exponentielles Laufzeitverhalten zeigen.

Einer der Gründe für die Ineffizienz des Algorithmus ist dabei, daß keinerlei Informationen, die in einem früheren Abschnitt der Berechnung ermittelt wurden, zu einem späteren Zeitpunkt genutzt werden. Ein solches Vorgehen ist aber für die Effizienz des Algorithmus von großer Wichtigkeit. Die theoretische Rechtfertigung für ein solches Vorgehen liefert das nächste Lemma.

Lemma 4.3.15 *$H \vdash s \in \nu X.\Phi$ habe ein erfolgreiches Tableau. Es gilt: $H \cup \{s : \nu X.\Phi\} \vdash s' \in \Gamma$ besitzt ein erfolgreiches Tableau genau dann, wenn $H \vdash s' \in \Gamma$ ein erfolgreiches Tableau besitzt.*

Beweis: Da wir die Korrektheit und Vollständigkeit des Tableau–Systems zugrundelegen können, genügt es zu zeigen, daß gilt

$$[\![\Gamma]\!]^H e = [\![\Gamma]\!]^{H \cup \{s:\nu X.\Phi\}} e$$

wenn man zusätzlich annimmt, daß $s \in [\![\nu X.\Phi]\!]^H e$ gilt. Gemäß der Definition der relativierten Semantik gilt

$$[\![\nu X.\Phi]\!]^H e = \varphi_{S'} \cup S',$$

mit $\varphi(S) = [\![\Phi]\!]^H e[X \mapsto S]$ und $S' = H\lceil \nu X.\Phi$. Für $s : \nu X.\Phi \in H$ gilt die Aussage offensichtlich wegen $H \cup s : \nu X.\Phi = H$. Sei also $s : \nu X.\Phi \notin H$. Dann gilt $s \in \nu\varphi_{S'}$ und

gemäß Lemma 4.3.9 folgt

$$\nu\varphi_{S'} = \nu\varphi_{S' \cup \{s\}}$$

woraus unmittelbar folgt

$$[\![\nu X.\Phi]\!]^H e = [\![\nu X.\Phi]\!]^{H \cup \{s:\nu X.\Phi\}} e$$

Die Behauptung ergibt sich nun als einfache Induktion über den Aufbau von Γ. □

Dieses Ergebnis kann man benutzen, um die Geschwindigkeit des Modelcheckings zu verbessern. Hat man $H \cup \{s : \nu X.\Phi\} \vdash s' \in \Gamma$ bewiesen (widerlegt), und im weiteren $H \vdash s \in \nu X.\Phi$ bewiesen, so kann damit $H \vdash s' \in \Gamma$ beweisen (widerlegen), ohne dafür ein Tableau konstruieren zu müssen.

Darüber hinaus hilft das folgende elementare Lemma bei der Beschleunigung des Algorithmus.

Lemma 4.3.16 *$\sigma = H \vdash s \in \nu X.\Phi$ besitzt ein erfolgreiches Tableau genau dann, wenn $\sigma' = H' \vdash s \in \nu X.\Phi$ mit $H' = H \setminus \{s' : \Gamma \mid \nu X.\Phi \prec \Gamma\}$ ein erfolgreiches Tableau besitzt.*

Beweis: Es gibt zwei Möglichkeiten. Ist $s : \nu X.\Phi \in H$ so gilt auch $s : \nu X.\Phi \in H'$ und somit folgt die Behauptung. Gilt $s : \nu X.\Phi \notin H$ so auch $s : \nu X.\Phi \notin H'$. Die Anwendung der Tableau–Regel (R7) führt aber nun zur gleichen Sequenz. □

In Abbildung 4.5 sieht man gut, welche Einsparungen bei der Konstruktion des Tableaus möglich werden. Es handelt sich hier um ein Tableau für das gleiche Bespiel wie in Abbildung 4.3, in dem die Konstruktion eines Untertableaus an der Stelle (∗∗) durch die Anwendung von Lemma 4.3.15 und Lemma 4.3.16 vermieden werden konnte.

Die vorgeschlagenen Optimierungen beziehen sich nur auf Formeln mit größten Fixpunkten. Dabei handelt es sich aber nicht um eine wirkliche Einschränkung, da Formeln, die kleinste Fixpunkte enthalten, leicht in äquivalente Formeln umgewandelt werden können, die nur größte Fixpunkte verwenden.

Es handelt sich bei dieser Methode zur Beschleunigung des Modelcheckings um einen typischen Zeit/Platz Trade–Off. Daher ist es kaum möglich, alle Informationen über bewiesene Sequenzen zu speichern. Dennoch benutzen wir in TRUTH einen Modelchekking–Algorithmus, der einen Teil der bewiesenen Sequenzen speichert, um schneller zu einem Ergebnis zu kommen.

4.4. Implementierung in Truth

TRUTH unterstützt die Verifikation von reaktiven Systemen mit Hilfe eines Modelchecking–Algorithmus für den vollen μ–Kalkül.

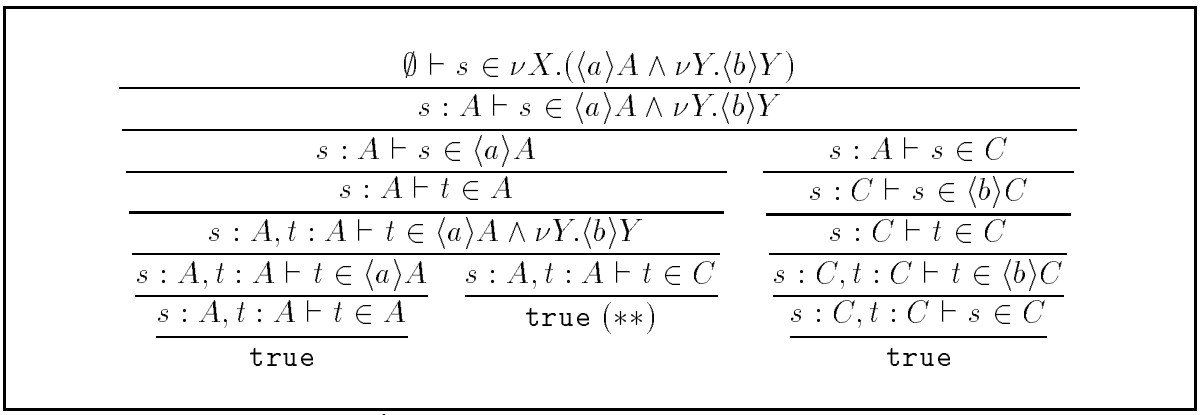


Abbildung 4.5: Ein optimiertes Tableau

Ähnlich wie für CCS–Spezifikationen existiert ein `Happy`–generierter Parser für Formeln des μ–Kalküls. Diese werden in einer Umgebung abgelegt, die durch verschiedene Funktionen ausgegeben und geändert werden kann. Dazu existiert auch ein Pretty–Printer, der eine Darstellung der Formeln unter Berücksichtigung der üblichen Präzedenzen und Assoziativitäten erzeugt.

4.4.1. Modifikationen der Logik

Wir verwenden den μ–Kalkül hier nicht ganz in der Form, wie er zuvor vorgestellt wurde, sondern ändern ihn an einigen Stellen leicht ab.

Wie zuvor erwähnt wurde, existieren für CTL*–Formeln Übersetzungen nach $L\mu$. Für CTL sind diese Übersetzungen sogar sehr einfach in Form von Makros möglich. Beispiele dazu finden sich auf Seite 48.

TRUTH unterstützt diese Art der Übersetzung durch die Einführung von Formelmakros. Diese werden vor dem Modelchecking aufgelöst, so daß der Modelchecker nur einfache Formeln des μ–Kalküls behandeln muß. Aus diesem Grund ist es nicht erlaubt, rekursive Definitionen anzugeben. Ein Beispiel für ein solches Makro sieht folgendermaßen aus:

```
AG(P) = max X.(P && [-]X)
```

In diesem Beispiel tritt eine weitere Erweiterung auf: Es ist möglich, bei den Modalitäten nicht nur Aktionen sondern auch Aktionenmengen anzugeben. Die Semantik der sich ergebenden Formeln ist eine naheliegende Erweiterung der Semantik des μ–Kalküls.

$$[\![\langle \alpha_1, \alpha_2, \ldots, \alpha_n\rangle \Phi]\!]e := \bigcup_{\alpha \in \{\alpha_1, \alpha_2, \ldots, \alpha_n\}} [\![\langle \alpha\rangle \Phi]\!]e$$

Für endliche Aktionenalphabete *Act* macht es darüber hinaus Sinn, Komplemente von Aktionenmengen zuzulassen. Ein „`-`“ vor der Aktionenmenge in der Modalität zeigt an, daß statt der angegebenen Menge deren Komplement betrachtet werden soll. Die

Semantik ergibt sich entsprechend. Ein „`-`“ anstelle irgendwelcher Aktionen bezeichnet das Komplement der leeren Aktionenmenge, also die Menge aller Aktionen. Diese Schreibweise findet sich in der obigen Definition von `AG(P)` wieder.

Wir haben bisher den μ–Kalkül mit sogenannten *starken* Modalitäten betrachtet, das heißt, daß die stille Aktion τ bei der Interpretation der Modalitäten keine gesonderte Rolle spielt, sondern genau wie die anderen Aktionen behandelt wird.

TRUTH stellt zu diesen Modalitäten zwei weitere zur Verfügung, welche der Sonderrolle von τ Rechnung tragen, dies sind die Modalitäten $\langle\langle\alpha\rangle\rangle\Phi$ und die dazu duale $[[\alpha]]\Phi$. Man bezeichnet diese auch als *schwache* Modalitäten. Als Semantik dieser Konstrukte definieren wir:

$$[\![\langle\langle\alpha\rangle\rangle\Phi]\!]e := \pi^*_\alpha([\![\Phi]\!]e),$$

mit

$$\pi^*_\alpha(S) = \{s' \mid \exists s \in S : \exists t_1, t_2 \in S : s' \stackrel{\tau}{\rightarrow}{}^* t_1 \stackrel{\alpha}{\rightarrow} t_2 \stackrel{\tau}{\rightarrow}{}^* s\}$$

Neben diesen Erweiterungen wird auch eine Einschränkung vorgenommen. Wir erlauben keine atomaren Propositionen in Formeln, obwohl diese für den Prozeß des Modelcheckings kein Problem darstellen. Wir betrachten in TRUTH nicht explizit gegebene Transitionssysteme sondern solche, die implizit in Form einer CCS–Spezifikation gegeben sind. Für solche Transitionssysteme ist nicht klar, wie eine Interpretation der atomaren Propositionen definiert werden soll.

Dabei handelt es sich aber nicht um eine wirkliche Einschränkung, da wir bei Formeln für CCS–Spezifikationen in der Regel nur an der Kommunikationsstruktur des Systems interessiert sind. Dies liegt daran, daß in CCS weder Variablen mit Belegungen noch Prozesse im eigentlichen Sinne existieren, über die man Aussagen mit Hilfe von atomaren Propositionen machen könnte. Die Kommunikationsstruktur ist von vorrangiger Bedeutung. „Atomare Aussagen“ über die Kommunikationsstruktur sprechen über die Möglichkeit und Unmöglichkeit verschiedener Aktionen. Solche lassen sich leicht mit Hilfe der Modalitäten und tt oder ff ausdrücken. Die Formel $\langle\alpha\rangle$tt ist zum Beispiel erfüllt, wenn ein α–Schritt möglich ist, während die Formel $[\alpha]$ff erfüllt ist, wenn keine α–Schritte möglich sind.

4.4.2. Der Datentyp für Formeln

Der Datentyp für Formeln des μ–Kalküls ist eng an der Struktur der Formeln orientiert:

```
data GenMuFormula label
  = MuTrue
  | MuFalse
```

```
  | MuAtom Id
  | MuNeg (GenMuFormula label)
  | MuOr (GenMuFormula label) (GenMuFormula label)
  | MuAnd (GenMuFormula label) (GenMuFormula label)
  | MuDiam Bool Bool [label] (GenMuFormula label)
  | MuBox Bool Bool [label] (GenMuFormula label)
  | MuLFP Id (GenMuFormula label)
  | MuGFP Id (GenMuFormula label)
  | MuApp Id [GenMuFormula label]
  deriving Eq
```

Er ist polymorph im Datentyp zur Beschriftung der Modalitäten. Für Variablen wird wieder der Typ `Id` benutzt, wie er auch für alle anderen Bezeichner im System Verwendung findet.

Bei den beiden Modalitäten wird in den zwei `Bool`–Werten vermerkt, ob es sich um starke oder schwache Modalitäten handelt, sowie ob die angegebene Aktionenmenge oder deren Komplement bei der Interpretation betrachtet werden soll.

4.4.3. Modelchecking

Der in TRUTH implementierte Modelchecking–Algorithmus ist eine Variante des oben beschriebenen, verbesserten Algorithmus. Vor dem Aufruf des eigentlichen Modelchekkers werden die Makros in den Formeln ersetzt und die minimalen Fixpunkte durch maximale ausgedrückt. Es wird überprüft, ob alle in der Spezifikation verwendeten Bezeichner auch gebunden sind.

Repräsentierung von Formeln

Zur Beschleunigung des Modelcheckers operiert dieser nicht auf dem oben beschriebenen algebraischen Datentyp. Insbesondere die Substitution, wie sie bei der Abwicklung der Fixpunkte notwendig wird, ist eine aufwendige Operation. Stattdessen finden die folgenden Datentypen Verwendung:

```
data FastFormula
  = FTrue
  | FFalse
  | FNeg Int
  | FAnd Int Int
  | FOr Int Int
  | FBox Bool Bool [Action] Int
  | FDiam Bool Bool [Action] Int
```

```
  | FGFP Int

type FTable = Array Int FastFormula
type SubFMap = FiniteMap Int [Int]
```

Die `Int`–Werte, die in `FastFormula` auftreten, werden als Indizes in einem Array vom Typ `FTable` verwendet. Auf diese Weise ist es möglich, gleiche Unterformeln auszudrücken. Wichtiger noch ist, daß es bei den Fixpunkt–Formeln möglich ist, direkt die Formeln abzuspeichern, die sich durch Abwicklung der Fixpunkte ergeben, da nur diese für das Modelchecking von Interesse sind. Die aufwendige Substitution zur Laufzeit des Modelcheckers wird somit vermieden.

Ein solches Vorgehen ist natürlich nur dann möglich, wenn die Anzahl der zu betrachtenden Formeln endlich ist. Obwohl die Abwicklung von Fixpunkten zu einer Verlängerung von Formeln führt, ist dies für das betrachtete Tableau–Verfahren stets der Fall.

Bei der Umwandlung der Formel in dieser Darstellung wird darüber hinaus Information über die wechselseitige Abhängigkeit der Fixpunkte ermittelt. Insbesondere wird in einer Umgebung vom Typ `SubFMap` gespeichert, welche Hypothesen bei der Abwicklung der jeweiligen Fixpunkte aus der Hypothesenmenge gestrichen werden müssen. So wird auch diese Berechnung während der Laufzeit des Modelcheckers vermieden.

Ein weiterer, wichtiger Effekt ist, daß jede relevante Formel durch einen `Int`–Wert repräsentiert wird, was den Speicherplatz zur Speicherung der Hypothesen reduziert.

Speicherung der Hypothesen–Mengen

Entscheidend für die Effizienz des Modelcheckings ist die Repräsentierung der Hypothesen–Mengen. Dies ist aus mehreren Gründen der Fall. Bei jedem Fixpunkt–Schritt des Tableau–Algorithmus muß entschieden werden, ob $s : \Gamma \in H$ gilt. Bei einer naiven Implementierung ist dies mit einem Aufwand verbunden, der linear, zumindest aber logarithmisch in der Größe der betrachteten Hypothesenmenge ist. Darüber hinaus müssen beim Abrollen von Fixpunkten Hypothesen aus der Hypothesen–Menge entfernt werden. Auch dies hat bei naiver Implementierung einen Aufwand, der linear in der Größe der Menge H ist. Man erkennt bereits, wie wichtig es ist, ein destruktives Update der Menge H durchführen zu können, da sonst eine zeit– und vor allem platz–effiziente Implementierung der beschriebenen Funktionen nicht möglich wäre.

Aus diesem Grund wählen wir eine andere Möglichkeit zur Speicherung der Hypothesen–Mengen, wie sie auch in der Implementierung der Edinburgh Concurrency Workbench verwendet wird.

Wir speichern die Hypothesen nicht in der Form einer Menge oder Liste, sondern wir annotieren die Zustände des Transitionssystems mit den Hypothesen, die über diesen Zustand im Laufe des Modelcheckings getätigt wurden. Da das Transitionssystem als

veränderbares Array im Rahmen einer Zustandsmonade abgelegt wird, ist sowohl das Hinzufügen von Hypothesen als auch der Test, ob eine gewisse Hypothese schon getätigt wurde, in konstanter Zeit möglich.

Dieses Vorgehen erzeugt aber ein anderes Problem: Der ursprüngliche Tableau–Algorithmus ist nicht–deterministisch. Dieser Nicht–Determinismus muß in der Implementierung aufgelöst werden. In der vorliegenden Implementierung in TRUTH geschieht dies durch eine Tiefensuche. Auf jeden Fall gibt es Stellen im Algorithmus, an denen das Tableau verzweigt. Bei einer globalen Speicherung der Hypothesen–Menge, wie dies in unserer Implementierung der Fall ist, müssen daher alle Änderungen an der Hypothesen–Menge rückgängig gemacht werden, damit der Algorithmus korrekt arbeitet. Ändert man also die Hypothesen–Mengen, indem man Hypothesen entfernt, so müssen diese Hypothesen gespeichert werden, um sie später wieder hinzuzufügen, was mit einem erheblichen Aufwand verbunden ist. Hinzu kommt, daß bei jedem Fixpunkt–Schritt das gesamte Transitionssystem nach zu entfernenden Hypothesen durchsucht werden muß, was angesichts der Größe der Transitionssysteme mit erheblichem Aufwand verbunden ist.

Wir lösen dieses Problem, indem wir auf eine solche Löschung von Hypothesen gänzlich verzichten. Stattdessen versehen wir jede Hypothese mit einer Zeitangabe, die es erlaubt zu entscheiden, ob die Hypothese zu einem späteren Zeitpunkt als gültig oder entfernt angesehen werden muß. Dabei handelt es sich nicht wirklich um eine Zeitangabe, sondern um einen Zähler, der die Anzahl der Hypothesen über andere Fixpunktformeln aufsummiert, von denen eine Fixpunktformel abhängt. Sollte eine Formel von mehreren anderen abhängen, so werden die entsprechenden Werte in einer festen Reihenfolge in einer Liste gespeichert.

Da bei den Zuständen Hypothesen über verschiedene Formeln gespeichert werden müssen, wird nun jeder Zustand mit einem Wert vom Typ `Assumptions` beschriftet. Damit darüber hinaus noch eine bedarfsgesteuerte Generierung des Transitionssystems möglich wird, erweitert man diese Annotation um einen Wert vom Typ `Bool` und implementiert für den so erhaltenen Typ die Klasse `LTSLabeling`.

```
type Timestamp = [Int]
type Assumptions = FiniteMap Int [Timestamp]
type ACounter s = MutArray s Int Int

data TabLabel = TabLabel Bool Assumptions

instance LTSLabeling TabLabel where
  isExpanded (TabLabel b _) = b
  markExpanded (TabLabel _ a) = TabLabel True a
  defLabeling = TabLabel False emptyFM
```

Die Anzahl der Hypothesen, die über die verschiedenen Fixpunktformeln aktiv sind, werden zur Laufzeit in einem veränderbaren Array vom Typ `ACounter` gespeichert, damit eine schnelle Änderung möglich ist.

Ein Fixpunkt–Schritt des Algorithmus sieht damit wie folgt aus:

```
case ftable!i of
  FGFP i1
    -> extractTimestamp acounter i subfm `thenLTS` \ timestamp ->
       ifLTS ( isActiveAssumption st i timestamp )
         ( returnLTS (True, emptyAEnv) )
         ( addAssumption acounter st i timestamp `seqLTS`
           check st i1 acounter                  `thenLTS` \ (res,aenv) ->
           ...
         )
```

Zunächst wird der aktuelle Timestamp bezüglich der betrachteten Formel generiert. Mit Hilfe dieses Timestamps ist es möglich zu entscheiden, ob eine aktive Hypothese über den Zustand und die Formel existiert. In diesem Fall ist das Tableau erfolgreich. Ist keine aktive Hypothese vorhanden, wird eine neue Hypothese, die mit dem aktuellen Timestamp versehen ist, zu den bestehenden hinzugefügt, und der Algorithmus versucht, das Tableau für die abgerollte Formel zu beweisen.

`isActiveAssumption` geht dabei folgendermaßen vor. Es werden die Hypothesen für den zu betrachtenden Zustand aus dem Transitionssystem gelesen. Dann wird die Liste der Timestaps für die zu betrachtende Fixpunktformel aus der `FiniteMap` gelesen. Sollten keine Hypothesen vorhanden sein, so wird dabei die leere Liste zurückgeliefert. Sind Hypothesen vorhanden, so wird der aktuelle Timestamp mit dem Timestamp der zuletzt hinzugefügten Hypothese verglichen. Im Falle einer Übereinstimmung ist die Hypothese noch aktiv und das Tableau damit erfolgreich. Im anderen Fall sind Hypothesen hinzugekommen, von denen die aktuelle abhängt. Dabei hätte diese aus der Hypothesenmenge entfernt werden müssen und daher wird so vorgegangen, als ob keine Hypothese existierte.

```
isActiveAssumption::LTSState->Int->Timestamp->LTS_M s TabLabel Bool
isActiveAssumption st i ts
  = getAddLabelLTS st `thenLTS` \ (TabLabel _ assums) ->
    case lookupWithDefaultFM assums [] i  of
         [] -> returnLTS False
         (x:_) -> returnLTS (x==ts)
```

Beschleunigung des Algorithmus

Wie schon ausgeführt, ist es von großer Wichtigkeit, Informationen über bereits bewiesene Sequenzen zu speichern, um einen effizienteren Algorithmus zu erhalten.

Dies wird in der Implementierung in TRUTH auf eine recht einfache Weise realisiert. Wenn der Algorithmus ein erfolgreiches Tableau für einen Fixpunktknoten $H \vdash s \in \nu X.\Phi$ konstruiert und somit gezeigt hat, daß der entsprechende Knoten ein gültiger Knoten ist, so wird die Hypothese $s : \nu X.\Phi$ nicht wieder entfernt. Dadurch werden alle weiteren Berechnungen nicht mit Hypothesenmenge H, sondern mit $H \cup \{s : \nu X, \Phi\}$ durchgeführt.

Auf diese Weise ist es für weitere Tableaus nicht mehr nötig, einen Beweis für $s \in \nu X.\Phi$ zu führen, da sich dieser direkt aus der Hypothesenmenge ergibt.

Wenn nun der Beweis für einen anderen Fixpunktknoten $\sigma' = H' \vdash s' \in \nu Y.\Psi$ fehlschlägt, muß die entsprechende Hypothese natürlich aus der Hypothesenmenge entfernt werden. Dabei ergibt sich das Problem, daß der Beweis von $H \vdash s \in \nu X.\Phi$ auf dieser Annahme beruht haben kann. Dies ist der Fall, wenn Y in Φ frei auftritt. Dadurch, daß diese Hypothese nun nicht mehr aufrechterhalten werden kann, wird auch der Beweis für $H \vdash s \in \nu X.\Phi$ ungültig, also muß auch die Hypothese $s : \nu X.\Phi$ nachträglich wieder entfernt werden.

Zu diesem Zweck liefert die Funktion `check` nicht nur einen `Bool` zurück, welcher die Gültigkeit des gegebenen Tableaus anzeigt, sondern zusätzlich einen Wert vom Typ

```
type AEnv = FiniteMap Int [LTSState]
```

welcher Formeln auf die Liste der Zustände abbildet, für die Hypothesen nicht entfernt wurden.

Im Falle des Scheiterns des Beweises eines Fixpunktknotens werden mit Hilfe dieser Abbildung alle Hypothesen ermittelt, auf deren Beweis der aktuelle Knoten einen Einfluß gehabt haben kann. Diese Hypothesen werden dann aus der Hypothesenmenge entfernt.

Der vollständige Algorithmus im Falle eines Fixpunktschrittes sieht also nun folgendermaßen aus:

```
case ftable!i of
  FGFP i1 ->
    extractTimestamp acounter i subfm `thenLTS` \ timestamp ->
    ifLTS ( isActiveAssumption st i timestamp )
          ( returnLTS (True, emptyAEnv) )
          ( addAssumption acounter st i timestamp `seqLTS`
            check st i1 acounter `thenLTS` \ (res,aenv) ->
            if res
              then returnLTS (res,addToAEnv aenv i st)
              else
                removeAssumption acounter st i `seqLTS`
                removeDepAssumptions acounter aenv i    `thenLTS` \ aenv'->
                returnLTS (res,aenv')
          )
```

Nach dem rekursiven Aufruf von `check` wird in Abhängigkeit des Ergebnisses entweder die Hypothese beibehalten und zur Umgebung `aenv` hinzugefügt, oder die Hypothese wird entfernt und mit ihr alle, deren Beweis auf ihr beruht haben könnte.

Es werden auf diese Weise unter Umständen zu viele Hypothesen entfernt, doch um „schärfer“ bei der Entfernung vorgehen zu können, müßten zusätzliche Informationen gespeichert werden, was den Speicherplatzbedarf des Algorithmus vergrößern würde.

4.4.4. Andere Modelchecker

Es existieren viele verschiedene Algorithmen zum Modelchecking für den μ–Kalkül über endlichen Kripkestrukturen. Aus diesem Grund ist es sinnvoll, ein Werkzeug wie TRUTH variabel im Hinblick auf den verwendeten Modelchecking–Algorithmus zu gestalten.

Die Signatur der Funktion, die den Tableau–basierten Modelchecker aufruft, sieht folgendermaßen aus:

```
tableaucheck::(Int,Int,Int) -> ProcEnv -> Process -> MuFormula -> Bool
```

Es werden als Parameter die Größen der Tabellen für das Transitionssystem erwartet. Dazu werden eine Spezifikation und eine Formel übergeben.

So lassen sich beliebige Modelchecker programmieren und an der entsprechenden Stelle einbauen. Das wird zusätzlich erleichtert durch die wechselnden Annotationen, die in das Transitionssystem eingetragen werden können.

4.5. Laufzeit–Messungen

Wie auch bei der Generierung der Transitionssysteme wurde TRUTH mit anderen Werkzeugen im Hinblick auf das Laufzeitverhalten des Modelcheckers verglichen. Die Ergebnisse finden sich in der unten stehenden Tabelle. Die Formeln, die zum Test verwendet wurden, finden sich in Anhang B. Dort findet sich auch die Beschreibung der Transitionssysteme 2LN und 3LN, über denen die Formeln ausgewertet wurden.

Neben einem Vergleich der drei Werkzeuge kann dieser Test auch als ein Vergleich von drei Modelchecking–Algorithmen aufgefaßt werden. Die NCSU CWB verfügt über einen Modelchecker, der auf das Fragment $L\mu_1$ optimiert wurde. Darüber hinaus existiert angeblich auch ein Modelchecker für $L\mu_2$, doch dieser ist nicht in der Lage, mit der Formel Φ_6 umzugehen. In der Edinburgh Concurrency Workbench wird ein spielbasierter Modelchecking–Algorithmus verwendet.

	TRUTH		NCSU		CWB	
Φ_1, 2LN	1.37 s	1.47 MB	1.3 s	-	1.37 s	-
Φ_1, 3LN	73.16 s	34.78 MB	64.99 s	147.49 MB	75.54 s	153.30 MB
Φ_2, 2LN	0.07 s	0.48 MB	1.36 s	-	0.11 s	-
Φ_2, 3LN	0.11 s	0.48 MB	67.91 s	148.87 MB	0.13 s	-
Φ_3, 2LN	5.76 s	2.66 MB	1.56 s	-	1.88 s	-
Φ_3, 3LN	598.52 s	133.55 MB	73.66 s	151.12 MB	99.37 s	196.82 MB
Φ_4, 2LN	2.22 s	1.75 MB	1.70 s	-	1.62 s	-
Φ_4, 3LN	187.82 s	92.50 MB	78.23 s	148.43 MB	87.76 s	180.13 MB
Φ_5, 2LN	0.10 s	0.47 MB	1.910 s	-	0.11 s	-
Φ_5, 3LN	0.14 s	0.47 MB	85.72 s	155.62 MB	0.12 s	-
Φ_6, 2LN	2.54 s	1.82 MB	n.a.	n.a.	0.14 s	-
Φ_6, 3LN	out of mem.	-	n.a.	n.a.	0.16 s	-

Bei der Ermittlung des Speicherverbrauchs von NCSU und CWB ergeben sich die gleichen Einschränkungen, wie sie bereits in Kapitel 3 erwähnt wurden.

Wie man deutlich sieht, sind lokale Modelchecking–Algorithmen den globalen oft überlegen. Dies sieht man insbesondere im Fall der Formel Φ_5. In diesem Fall sind TRUTH und die Edinburgh Concurrency Workbench der NCSU Concurrency Workbench weit überlegen, da letztere zunächst das gesamte Transitionssystem bestimmt. Dadurch erklären sich die enormen Unterschiede in Laufzeit und Speicherverbrauch.

In den Fällen, in denen das gesamte Transitionssystem in das Modelchecking mit einbezogen wird, liegt TRUTH mit seiner Leistung hinter den anderen Bewerbern zurück. Dies liegt im Fall der NCSU Concurrency Workbench daran, daß dort ein Modelchecking–Algorithmus zum Einsatz kommt, der speziell auf $L\mu_1$ und $L\mu_2$ abgestimmt ist [BC96a, BC96b], während der Algorithmus in TRUTH den vollen μ–Kalkül verarbeiten kann.

Die Werte für CWB, insbesondere für Φ_6, zeigen, daß der spielbasierte Ansatz sowohl dem von NCSU als auch dem von TRUTH überlegen ist. Es wird daher bereits an der Implementierung eines spielbasierten Modelcheckers für TRUTH gearbeitet.

Wenn entsprechende Algorithmen auch in TRUTH implementiert sind, wird die Leistung von TRUTH mit denen der anderen Werkzeuge vergleichbar sein.

5. Simulation von CCS–Prozessen

Bei der Entwicklung einer Spezifikation ist es oftmals notwendig, sich einen Überblick über die Arbeitsweise des spezifizierten Systems zu verschaffen. Viele Verifikations–Werkzeuge bieten hierzu die Möglichkeit zur interaktiven Simulation von Spezifikationen.

Die Nützlichkeit dieser Funktionen ist aus verschiedenen Gründen eingeschränkt, wie weiter unten beschrieben wird. Trotzdem ist es wünscheswert, ein geeignetes Verfahren zur interaktiven Simulation von Spezifikationen zu haben. Dies hat noch einen weiteren Grund.

Modelchecking ist ein geeignetes Verfahren, sich von der Korrektheit von Spezifikationen zu überzeugen. Zusätzlich ist es auch eine geeignete Möglichkeit zur Entwicklung korrekter Systeme, insbesondere, wenn Gegenbeispiele zu nicht erfüllten Formeln generiert werden. Diese demonstrieren die Schwächen des Systems und erleichtern auf diese Weise die Weiterentwicklung der Spezifikation.

Zur Darstellung solcher Gegenbeispiele sind aufwendige Mittel notwendig, da es sich dabei oft nicht einfach um Zustände des Systems handelt, die gewisse Eigenschaften verletzen, sondern oftmals um unendliche Läufe.

Spielbasiertes Modelchecking ist ein geeignetes Verfahren zum Modelchecking und erlaubt darüber hinaus eine elegante Angabe in Form einer Gewinnstrategie eines Ehrenfeucht–Fraissé–Spiels auf dem Transitionssystem. Eine Simulations–Funktion könnte dabei zur Darstellung und Durchführung der Spiele dienen und somit einen wichtigen Beitrag bei der Entwicklung von Spezifikationen leisten.

Damit die interaktive Simulation zu einem besseren Verständnis des Systems führen kann, ist es wichtig, daß das System in einzelne Komponenten zerfällt. Diese können als mehr oder weniger autonome Einheiten aufgefaßt und als solche verstanden werden. Die natürlichste Abstraktion, die hierbei hilfreich ist, ist die Vorstellung, daß das System in eine Reihe von Prozessen zerfällt, die auf verschiedene Weisen miteinander kommunizieren.

CCS eignet sich in dieser Hinsicht nicht dazu, ein System zu zergliedern, da es in CCS keinen Prozeßbegriff im eigentlichen Sinne gibt. Ein CCS–Ausdruck stellt ein monolithisches Gebilde dar, in dem keine einzelnen Prozesse auszumachen sind – das

Gesamtverhalten steht im Vordergrund. Demzufolge sind auch die Simulationsmöglichkeiten vorhandener Werkzeuge sehr beschränkt. Die NCSU CWB erlaubt genau wie die Edinburgh CWB die Simulation von CCS–Spezifikationen, ohne daß eine Zergliederung der Spezifikation möglich wäre. Daß dies zum Verständnis komplizierterer Systeme beitragen kann, ist zumindest fraglich.

In TRUTH ist es gelungen, durch eine kleine Erweiterung der Syntax von CCS einen Prozeßbegriff einzuführen. Dieser erweist sich als sehr hilfreich für die Zergliederung eines Systems in ein Systen kommunizierender Prozesse. Dadurch wird eine interaktive Simulation sinnvoll durchführbar.

Dieses Kapitel beschreibt zunächst die Erweiterung von CCS, die vorgenommen wurde, um eine sinnvolle Simulation zu ermöglichen. Danach wird das Vorgehen bei der Implementierung der Simulation in TRUTH beschrieben. Diese wurde erschwert durch das Fehlen geeigneter Bibliotheken zur Programmierung graphischer Benutzerschnittstellen für den `GHC`. Die Maßnahmen, die ergriffen wurden, um dieses Problem zu lösen, werden ebenfalls beschrieben.

5.1. Prozesse in CCS

Auch wenn CCS–Ausdrücke als monolithische Gebilde aufgefaßt und in der Regel nur als Ganzes betrachtet werden, ist die Vorgehensweise, die man bei der Entwicklung einer Spezifikation verfolgt, oft anders. Dabei wird das System zunächst in eine Menge von Prozessen zerlegt. Für diese wird dann separat eine Spezifikation erstellt und diese dann zu dem Gesamtsystem zusammengefaßt. Dabei ergibt sich oftmals die folgende Struktur in der Spezifikation:

$$\begin{aligned} P_1 &= \ldots \\ P_2 &= \ldots \\ &\vdots \\ P_n &= \ldots \\ System &= (P_1|P_2|\ldots|P_n) \setminus Internals \end{aligned}$$

Diese Struktur tritt zum Beispiel auch bei allen CCS–Spezifikationen auf, die zum Testen von TRUTH verwendet wurden.

Zur Simulation eines solchen Systems bietet sich eine Zerlegung in die Prozesse P_1 bis P_n an. Es reicht jedoch nicht aus, jedes Auftreten des Parallel–Operators als einen Punkt zur Zergliederung des Systems aufzufassen, da es auf diese Weise nicht möglich wäre, den Grad der Zergliederung des Systems zu beeinflussen. Darüber hinaus muß auch der Restriktion eine gewisse Bedeutung zukommen, da sie Kommunikation zwischen den eingeschränkten Prozessen erzwingen kann. Zuletzt sollte es möglich sein, daß sich die

Struktur des Systems dynamisch durch Hinzukommen oder Wegfallen von Prozessen verändern kann.

Aus diesen Überlegungen leitet sich die Struktur von Prozessen ab, wie sie in der interaktiven Simulation verwendet wird:

$$\begin{aligned} SimProc ::= & \; CCSProc \\ & \mid SimPar[SimProc] \\ & \mid SimResSimProcRestriction \\ & \mid SimSpawnSimProc \end{aligned}$$

Dabei machen wir von der Assoziativität des Parallel–Operators Gebrauch, indem wir $SimPar$ nicht binär sondern mit beliebiger Arität verwenden.

Der Benutzer zeigt durch die Verwendung des Operators „`#`“ anstelle von „`|`“ an, daß die Ausdrücke, die mit diesem Operator verbunden sind, als eigenständige Prozesse bei der Simulation dargestellt werden sollen.

Zu einer Abspaltung eines neuen Prozesses kommt es dann, wenn bei einem $CCSProc$ ein „`#`“ als oberster Operator oder nur unter Restriktionen auftritt.

Der CCS–Ausdruck $a.A + b.(B \# C)$ bildet bei der Simulation eine Einheit, während dieser Ausdruck durch einen b–Schritt in zwei separat zu betrachtende Prozesse B und C zerfällt.

5.2. Die `Haskell`–Implementierung

Eine komplexe Zergliederung, wie sie oben beschrieben wurde, läßt sich nicht mehr auf sinnvolle Weise text–basiert darstellen. Deswegen war es notwendig, graphische Ausgaben zu erzeugen. Zwar existiert für `Haskell` eine Bibliothek zur Programmierung graphischer Benutzerschnittstellen, diese ist jedoch noch im Alpha–Stadium und so fehlerhaft, daß eine Programmierung mit Hilfe von `Haggis` [FJ96] zur Zeit nicht sinnvoll scheint.

Es war notwendig, die Darstellung der Simulation durch ein externes Programm vorzunehmen. Dieses sollte über die Darstellung hinaus möglichst wenig Funktionalität besitzen müssen.

Aus diesem Grund wurde eine text–basierte Schnittstelle geschaffen, über die TRUTH und das Darstellungsprogramm kommunizieren. Diese erlaubt es, die Darstellungskomponente mit geringem Aufwand zu implementieren.

Darüber hinaus erlaubt die Definition der Text–Schnittstelle eine einfache Reimplementierung der Darstellungs–Komponente.

5.2.1. Datenstrukturen

Zur Simulation eines Ausdrucks wird dieser vom Typ `CCSProcess` in den Typ `SimUnit` umgewandelt. Dieser spiegelt die oben beschriebene Struktur wieder und speichert darüber hinaus einige weitere Informationen.

```
data SimUnit
  = SimProc CCSProcess [ActAndEffect] Unique
  | SimPar [SimUnit] Unique
  | SimRes SimUnit Id Unique
type ActAndEffect = (SimAction,CCSProcess)
data UniqAction = MkUniqAction Action Unique
data SimAction
  = SingleAction UniqAction
  | CombinedAction UniqAction
                   UniqAction UniqAction
```

Die `Unique`–Werte werden eingeführt, um eine eindeutige Identifizierung der verschiedenen Komponenten zu ermöglichen. Wir erlauben nur die Restriktion durch einen Mengenbezeichner und nicht durch eine Menge, um sicherzustellen, daß eine übersichtliche Ausgabe erzeugt werden kann.

Bei den atomaren Einheiten – den CCS–Prozessen – werden zusätzlich jegliche Aktionsmöglichkeiten dieser Prozesse abgelegt. Diese werden dann über die Schnittstelle übertragen, so daß es möglich ist, die verschiedenen Aktionen mit ihren Effekten zu assoziieren. Dadurch kann zwischen verschiedenen Schritten des Systems, die durch die gleiche Aktion beschriftet sind, unterschieden werden.

Aus praktischen Gründen unterscheiden wir darüber hinaus Aktionen, die ein einzelner Prozeß allein durchführt, von solchen, die die Interaktion zweier Prozesse voraussetzt.

5.2.2. Definition der Schnittstelle

Die Kommunikation gliedert sich in zwei Phasen: In einem ersten Schritt wird die Anfangskonfiguration des simulierten Systems definiert. Alle weiteren Schritte beziehen sich auf diese Struktur und beschreiben nur noch die aufgetretenen Änderungen. Die EBNF–Grammatik für die Initialisierung ist in Abbildung 5.1 dargestellt.

Unique steht für einen eindeutigen Bezeichner. ProcTerm steht für den CCS–Ausdruck, der den entsprechenden Zustand bezeichnet. Sollte es sich dabei um einen Zustand handeln, der nur durch einen Bezeichner beschriftet ist, so wird die rechte Seite der definierenden Gleichung als ProcDetails mit angegeben. Bei Restriktionen wird der Name der restringierenden Menge als Id und die Definition dieser Menge als ResDetails angegeben. Bei den Aktionen wird schließlich der Name der Aktion sowie der aus dieser

$$
\begin{aligned}
\mathit{Init} ::= \ & \texttt{\#BEGIN_UNIT} \\
& \mathit{Unit} \\
& \texttt{\#END_UNIT} \\
& \texttt{\#BEGIN_CHOICES} \\
& \mathit{Choices} \\
& \texttt{\#END_CHOICES} \\
& \mathit{LeashOrUnleash} \\
\mathit{Unit} ::= \ & \texttt{\#PROC Unique ProcTerm ProcDetails} \\
& \texttt{\#BEGIN_ACTS } \mathit{Actions} \texttt{ \#END_ACTS} \\
| \ & \texttt{\#BEGIN_PAR Unique} \\
& \mathit{Units} \\
& \texttt{\#END_PAR} \\
| \ & \texttt{\#BEGIN_RES Unique Id ResDetails} \\
& \mathit{Unit} \\
& \texttt{\#END_RES} \\
\mathit{Units} ::= \ & \mathit{UnitUnits} \\
| \ & \mathit{Unit} \\
\mathit{Actions} ::= \ & \mathit{Action}\ \mathit{Actions} \\
| \ & \varepsilon \\
\mathit{Action} ::= \ & \texttt{\#ACTION Unique Name ProcTerm} \\
\mathit{Choices} ::= \ & \mathit{Choice}\ \mathit{Choices} \\
| \ & \varepsilon \\
\mathit{Choice} ::= \ & \texttt{\#SINGLE_CHOICE Unique} \\
| \ & \texttt{\#COMBINED_CHOICE Unique Unique Unique} \\
\mathit{LeashOrUnleash} ::= \ & \texttt{\#LEASH} \\
| \ & \texttt{\#UNLEASH}
\end{aligned}
$$

Abbildung 5.1: Grammatik der Initialisierung

Aktion resultierende CCS–Ausdruck als ProcTerm angegeben. #LEASH bzw. #UNLEASH geben an, ob die Darstellungskomponente direkt auf ein Update von TRUTH warten soll (#LEASH), oder ob der Benutzer den nächsten Schritt des Systems bestimmt (#UNLEASH). Die Darstellungskomponente meldet eine Antwort zurück, die gemäß der nächsten Grammatik aufgebaut ist.

$$
\begin{aligned}
\mathit{Answer} ::= {} & \texttt{\#CHOICE}\ \mathsf{Unique} \\
\mid {} & \texttt{\#RESTART} \\
\mid {} & \texttt{\#QUIT} \\
\mid {} & \texttt{\#BACK} \\
\mid {} & \texttt{\#RANDOM}\ \mathsf{Steps}
\end{aligned}
$$

Eine CHOICE–Nachricht signalisiert, daß der Benutzer sich für die mit dem Unique bezeichnete Alternative entschieden hat. RESTART bzw. QUIT veranlassen einen Neustart der Simulation bzw. deren Beendigung. BACK verursacht die Rücknahme des letzten Schrittes, während RANDOM die Ausführung von Steps zufälligen Schritten initiiert.
Für diese Funktion ist die LEASH– bzw. UNLEASH–Nachricht eingeführt worden, da es vorkommen kann, daß mehrere Schritte dargestellt werden müssen, ohne daß eine Benutzereingabe möglich ist.
Beim Übertragen einer Änderung wird zunächst der Name der Aktion übergeben, die die Änderung des Gesamtsystems kennzeichnet. Danach werden alle lokalen Änderungen der Teilprozesse angegeben, wobei der sich ändernde Prozeß durch seinen Unique identifiziert wird. Kommt es durch den Schritt zu einer Aufspaltung eines Teilprozesses, so wird dies durch das Schlüsselwort #SPAWN gekennzeichnet, sonst erscheint das Schlüsselwort #CHANGE. Nachdem alle Änderungen übergeben wurden, werden die neuen Aktionsmöglichkeiten des Systems angegeben und wiederum mit #LEASH oder #UNLEASH angezeigt, ob Benutzereingaben möglich sein sollen oder nicht. Sollte es durch ein Deadlock unmöglich werden, weitere Schritte auszuführen, obwohl dies durch einen #RANDOM–Befehl gefordert war, so benachrichtigt TRUTH die Darstellungskomponente durch Übergabe des Schlüsselwortes #DEADLOCK. Als EBNF–Grammatik sieht das folgendermaßen aus:

$$
\begin{aligned}
\mathit{Update} ::= {} & \mathsf{Name} \\
& \texttt{\#BEGIN_CHANGES}\ \mathit{Changes}\ \texttt{\#END_CHANGES} \\
& \texttt{\#BEGIN_CHOICES}\ \mathit{Choices}\ \texttt{\#END_CHANGES} \\
& \mathit{LeashOrUnleash} \\
\mid {} & \texttt{\#DEADLOCK} \\
\mathit{Changes} ::= {} & \mathit{Change}\ \mathit{Changes} \\
\mid {} & \varepsilon \\
\mathit{Change} ::= {} & \texttt{\#CHANGE}\ \mathsf{Unique}\ \mathsf{Name}\ \mathit{Unit} \\
\mid {} & \texttt{\#SPAWN}\ \mathsf{Unique}\ \mathsf{Action}\ \mathit{Unit}
\end{aligned}
$$

5.3. Die Darstellungskomponente

Ziel bei der Implementierung der Darstellungskomponente war ein System, das den Benutzer bei der interaktiven Simulation und dem Verständnis des simulierten Systems optimal unterstützt. Dazu gehört eine einfache Bedienbarkeit genauso wie eine leicht verständliche, übersichtliche Darstellung der Prozesse.

In der Darstellung finden alle Elemente des Typs `SimUnit` eine klare Entsprechung. Die visuellen Komponenten sind in Abbildung 5.2 dargestellt.

Alle Zusatzinformationen, die in dieser Darstellung keinen Platz finden, die der Darstellungskomponente jedoch bekannt sind, werden beim Eintreten des Mauszeigers in die jeweilige Komponente in einer Statuszeile dargestellt. So sind Informationen über restringierte Aktionen, die Definition von Bezeichnern sowie das Ergebnis einer Aktion eines Prozesses abfragbar.

Da bei jedem Prozeß alle Aktionen abgebildet werden, die dieser zum gegebenen Zeitpunk durchführen kann, ist eine Unterstützung des Benutzers bei der Auswahl der durchzuführenden Aktion wichtig. Dies geschieht durch eine farbliche Kodierung der Aktionen.

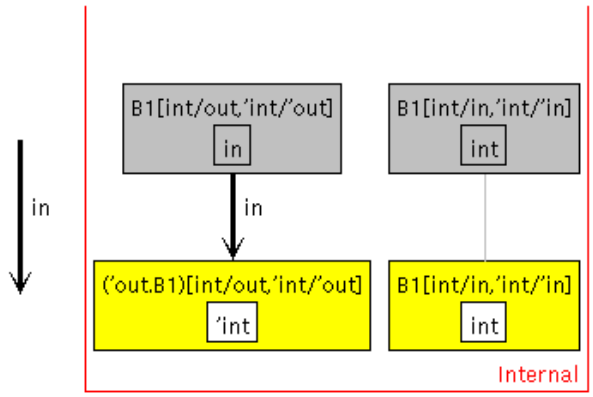
Aktionen, die zum gegebenen Zeitpunk unmöglich sind, erscheinen grau unterlegt. Selektierte Aktionen erscheinen grün. Sollte eine Aktion selektiert sein, so werden alle nicht–korrespondierenden Aktionen auch inaktiviert und grau unterlegt. Sobald eine Selektion vorgenommen wurde, die einen Schritt des Gesamtsystems zuläßt, wird ein Knopf aktiviert, mit dem ein Weiterschalten des Systems veranlaßt werden kann.

Von großer Wichtigkeit für die Bedienbarkeit der Simulation ist die Existenz eines Back–Knopfs, der es erlaubt, beliebig viele Schritte zurückzunehmen und so zu einem vorherigen Zustand des Systems zurückzukehren. Zur Vereinfachung gibt es weiterhin einen Restart–Knopf, der die Simulation wieder in den Anfangszustand versetzt. Schließlich ist es möglich, eine gewisse Anzahl von Schritten zufällig ausführen zu lassen. Dies geschieht durch Eingabe der gewünschten Anzahl von Schritten, gefolgt von der Betätigung des Random–Knopfes.

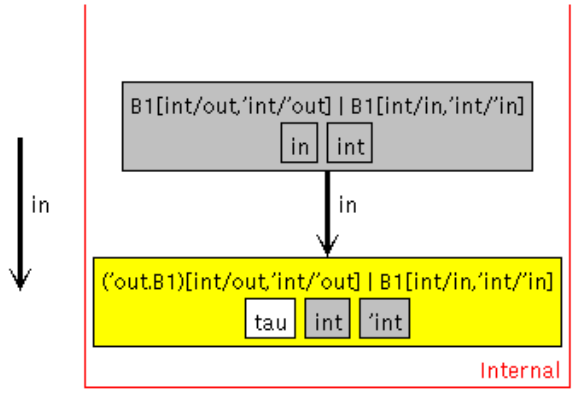
Zum Verständnis des Systems ist es wichtig, nicht nur den aktuellen Zustand des Systems abzubilden, sondern auch die Geschichte der Simulation darzustellen. In dieser Implementierung geschieht dies. Dabei werden sowohl die Schritte der einzelnen Komponenten als auch die Schritte des Gesamtsystems dargestellt. Abbildung 5.3 zeigt das Aussehen der Darstellungskomponente.

5.3.1. Implementierung der Darstellungskomponente

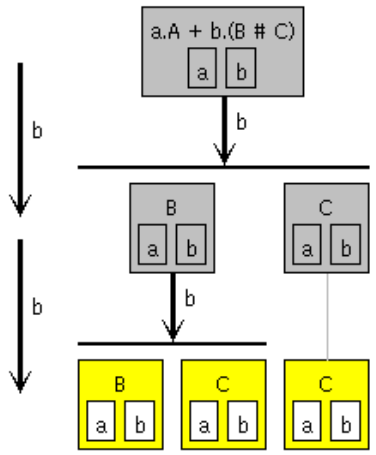
Wie bereits geschildert, war eine Implementierung der graphischen Oberfläche in `Haskell` nicht möglich, da keine entsprechenden Bibliotheken existieren. Also mußte nach einer Alternative gesucht werden, die schließlich in `Java` gefunden wurde.



Ein System mit zwei Prozessen. Der erste hat bereits einen in–Schritt durchgeführt. Nun können beide Prozesse zusammen einen Schritt durchführen. Die beiden Prozesse sind durch Internal eingeschränkt.



Das gleiche System ohne Zergliederung in Komponenten. Das System hat drei Möglichkeiten zu schalten, jedoch werden zwei davon durch die Restriktion verhindert.



Ein Beispiel für ein System, bei dem zur Laufzeit weitere Prozesse abgespalten werden.

Abbildung 5.2: Graphische Elemente der Simulation

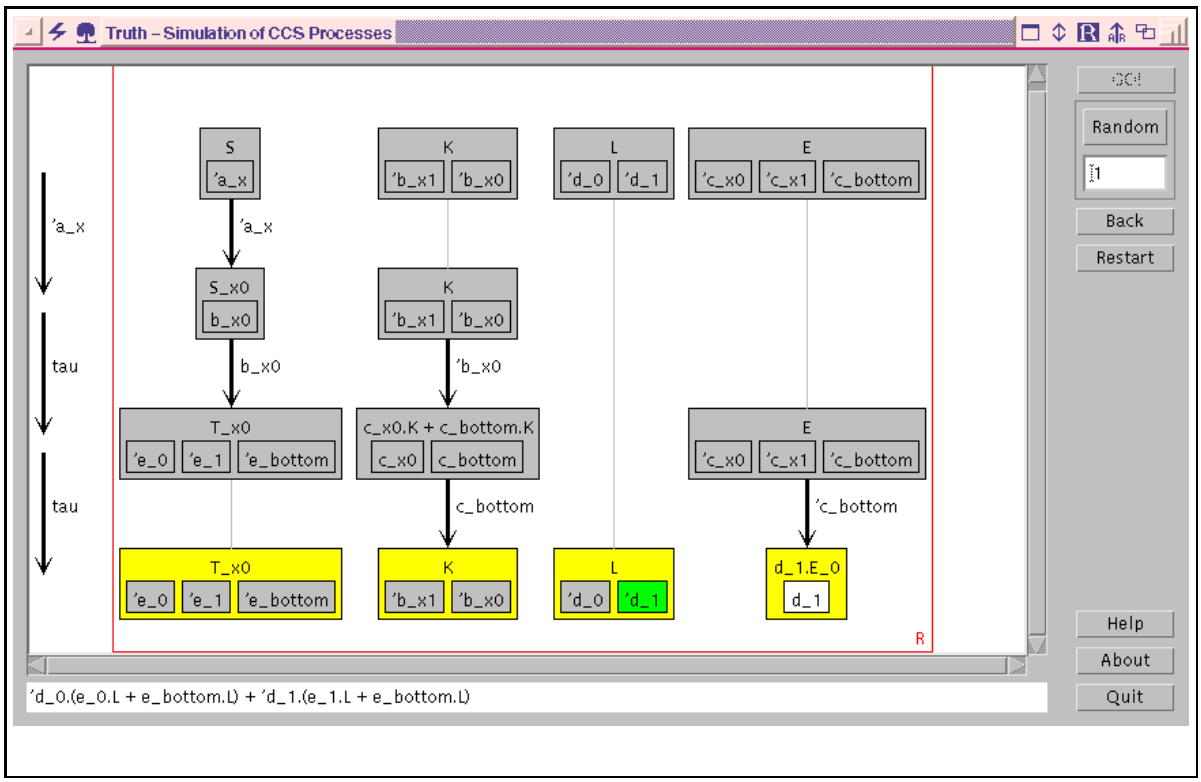


Abbildung 5.3: Die Darstellungskomponente

`Java` [GJS97] unterstützt die Implementierung graphischer Benutzerschnittstellen durch sein Advanced Windowing Toolkit. Dieses erlaubt eine einfache Programmierung von Standard–Komponenten für graphische Benutzerschnittstellen. Die Ansprüche, die bei der Implementierung der graphischen Schnittstelle gemacht wurden, liegen jedoch über dem, was durch das AWT unmittelbar unterstützt wird. Insbesondere die komplexe Anordnung interaktiver Elemente, wie wir sie zur Darstellung des Systems benötigen, ist mit Hilfe der Layout–Manager nicht möglich. Diese bilden im AWT das Standard–Mittel zur Anordnung graphischer Elemente auf dem Bildschirm. Dieses mußte ersetzt werden durch eine Vorgehensweise, die den besonderen Ansprüchen unserer Anwendung genügt. Alle Objekte, die solche dynamischen Komponenten realisieren, sind daher mit Methoden ausgestattet, die das Layout dieser Komponenten unterstützen.

Datenstrukturen

Da die `Haskell`–Datenstruktur, welche die simulierten Prozesse darstellt, rekursiv ist, lag es nahe, diese Datenstruktur in der `Java` Datenstruktur nachzubilden. Dies geschieht so, wie dies in einer objektorientierten Sprache üblich ist. Die abstrakte Klasse, die die gemeinsame Funktionalität repräsentiert, heißt `SimArenaElement` und hat im wesentlichen die folgenden Methoden:

```
public abstract int getAbsoluteWidth(FontMetrics fm);
public abstract int getAbsoluteDepth(FontMetrics fm);
```

```
public abstract int getAbsoluteHeight(FontMetrics fm);
public abstract int getHistoryDepth(FontMetrics fm);
```

Diese Funktionen dienen dazu, die Ausdehnung auf dem Bildschirm zu bestimmen.

```
public abstract void paint(Graphics g,FontMetrics fm,int x,int y,int topy);
```

Wir leiten `SimArenaElement` nicht von `Component` ab. Dennoch soll eine Graphikausgabe erfolgen. Die entsprechende Methode erhält jedoch zusätzliche Parameter, mit denen die Plazierung der Ausgabe gesteuert wird.

```
public abstract SList getActions();
public abstract SList getSimBoxes();
public abstract SList getSimResAreas();
```

Die vorliegende Implementierung erlaubt es, Kontextinformationen über dargestellte Objekte abzufragen, indem der Mauszeiger in den entsprechenden Bereich bewegt wird. Alle Bereiche, die sich auf diese Weise verhalten und zu einer Ausgabeaktion führen, können mit diesen Methoden abgefragt werden.

```
public abstract void update(Unique uniq, String label,
  SimBox new_simbox, int y, int counter);
public abstract SimArenaElement spawn(Unique uniq,
  String label, SimArenaElement new_element, int y, int counter);
```

Die Veränderung einer Komponente wird mit diesen Methoden veranlaßt. Sie korrespondieren in direkter Weise zu den `#CHANGE` bzw. `#SPAWN` Kommandos, wie sie oben beschrieben wurden.

```
public abstract void nextStep(int y, int counter);
public abstract SimArenaElement back(int counter);
```

Diese Methoden benachrichtigen eine Komponente von der Tatsache, daß ein Schritt bei der Simulation durchgeführt bzw. ein Rückschritt vom Benutzer angefordert wurde.

Das Einlesen von `SimArenaElement` abgeleiteten Klassen geschieht automatisch und transparent für den Benutzer dieser Klassen. Diese werden nicht exportiert, so daß der einzige Zugriff über die oben beschriebenen Methoden erfolgen muß. Die folgende Methode wandelt die oben beschriebene Textrepräsentierung in ein Objekt vom Typ `SimArenaElement` um.

```
public static SimArenaElement readSimArenaElement(StreamTokenizer t)
  throws IOException;
```

Darüber hinaus ist noch eine Reihe von Methoden implementiert, die die Verarbeitung einer Liste von Objekten dieses Typs unterstützen.

Es gibt vier Klassen, die von der Klasse `SimArenaElement` abgeleitet sind.

- `AEProcess` entspricht einem atomaren Teilprozeß. Der wesentliche Bestandteil eines Objekts von diesem Typ ist ein Objekt vom Typ `SimBoxHistory`. Diese Klasse wird später noch näher erläutert. Objekte vom Typ `AEProcess` bilden die Blätter der Baumstruktur, die das simulierte System repräsentiert.

- `AERes` entspricht der Restriktion einer Instanz der Klasse `SimArenaElement`, welche durch einen Bezeichner restringiert ist.

- `AEPar` repräsentiert die parallele Kombination `SimArenaElements`. Diese werden in Form einer verketteten Liste gespeichert. Die Implementierung der abstrakten Methoden benutzen die oben erwähnten Hilfsfunktionen zur Verarbeitung einer Liste von Instanzen von `SimArenaElement`.

- `AESpawn` entspricht der Abspaltung von Prozessen. Diese Klasse wurde zusätzlich eingeführt, da sie in der Darstellung völlig neue Anforderungen stellt. Zusätzlich zu einem Objekt vom Typ `SimArenaElement` ist auch eine weitere `SimBoxHistory` gespeichert, die die gemeinsame Vergangenheit aller Unterprozesse dieses Objektes darstellt und auch dementsprechend auf dem Bildschirm erscheinen muß.

Die zentrale Klasse bei der Darstellung des Systems ist die Klasse `SimBoxHistory`, in der ein atomarer Prozeß sowie dessen Geschichte während der Simulation abgelegt wird. Diese Klasse speichert eine Liste von Objekten vom Typ `SimBox`. Sie stellen die Vergangenheit des dargestellten Prozesses dar. Darüber hinaus wird der aktuelle Zustand des Prozesses gespeichert. Es werden wiederum alle für die Berechnung des Layouts benötigten Methoden bereitgestellt. Die Methoden `update`, `nextStep` und `back` ermöglichen die Manipulation eines Objektes. In dieser Klasse sind auch die Methoden zur graphischen Ausgabe implementiert.

Klassenbibliotheken

Bei der Implementierung werden an verschiedenen Stellen Listen und ähnliche Datenstrukturen verwendet. Wir benutzen dafür die Implementierungen, die in der Generic Library Collection von Objectspace zur Verfügung gestellt werden [Obj97]. Diese stellt effiziente Implementierungen der gebräuchlichsten Container–Klassen bereit. Dazu gehören einfach und doppelt verkettete Listen, Stacks, Queues und andere Datenstrukturen. In Anlehnung an die STL für C++ existieren für jede dieser Datenstrukturen Iterator–Klassen, über die die Manipulation der Datenstrukturen möglich ist.

Wir benutzen insbesondere die einfach verkettete Liste bei der Speicherung paralleler Prozesse, sowie eine Queue zur Speicherung der Geschichte eines simulierten Prozesses.

6. Ein Beispiel: Das Alternating Bit Protocol

Dieses letzte Kapitel soll die Arbeitsweise und Funktionalität von TRUTH an einem größeren Beispiel darstellen.

Netzwerk–Protokolle sind typische Beispiele für verteilte Systeme. Manche Protokolle regeln die Kommunikation zwischen einer großen Anzahl von Rechnern, die über ein oder mehrere Kanäle kommunizieren wollen. Andere dienen dazu, sichere Kommunikation über unsichere Kanäle zu ermöglichen. Ein unsicherer Kanal kann Nachrichten verlieren, duplizieren oder verstümmeln.

In unserem Beispiel betrachten wir Kanäle, die eine einmalige Übertragung jeder Nachricht garantieren, dabei aber die Nachricht verstümmeln können. Eine Kommunikation wird weiterhin erschwert, indem auch der Kanal, über den der Empfänger Bestätigungen an den Sender zurückschickt, diese Nachrichten verstümmeln kann.

Ein bekanntes Beispiel für die Realisierung eines Protokolls unter solchen Bedingungen ist das Alternating Bit Protocol [Mil89], kurz ABP. Es benutzt ein Bit, um die Übertragung so zu regeln, daß im Falle der Korruption einer Nachricht diese erneut versendet wird und damit eine fehlerlose Übertragung von Daten möglich wird.

Die zwei Kanäle modellieren wir in CCS in der folgenden Weise:

$$K \stackrel{\text{def}}{=} \overline{\texttt{sendreq}_0}.(\texttt{send}_0.K + \texttt{send}_\perp.K) + \overline{\texttt{sendreq}_1}.(\texttt{send}_1.K + \texttt{send}_\perp.K)$$
$$L \stackrel{\text{def}}{=} \overline{\texttt{ackreq}_0}.(\texttt{ack}_0.L + \texttt{ack}_\perp.L) + \overline{\texttt{ackreq}_1}.(\texttt{ack}_1.L + \texttt{ack}_\perp.L)$$

Nach Empfang einer $\overline{\texttt{sendreq}_i}$–Nachricht, kann K diese entweder weitersenden und eine $\texttt{send}_i$–Aktion ausführen, oder aber eine verstümmelte Nachricht $\texttt{send}_\perp$ versenden. Das Verhalten von L für die Bestätigungen ist analog modelliert.

An diesem Beispiel ist gut zu erkennen, wie die Abstraktion vom eigentlichen System stattfindet. Es wird weder der Wert der übermittelten Nachricht in die Modellierung mit einbezogen, noch die Art und Weise, wie Sender und Empfänger feststellen, daß eine Nachricht wirklich verstümmelt wurde. Dies hat für die Funktionsweise des Protokolls keine Auswirkungen und wird aus diesem Grund nicht betrachtet.

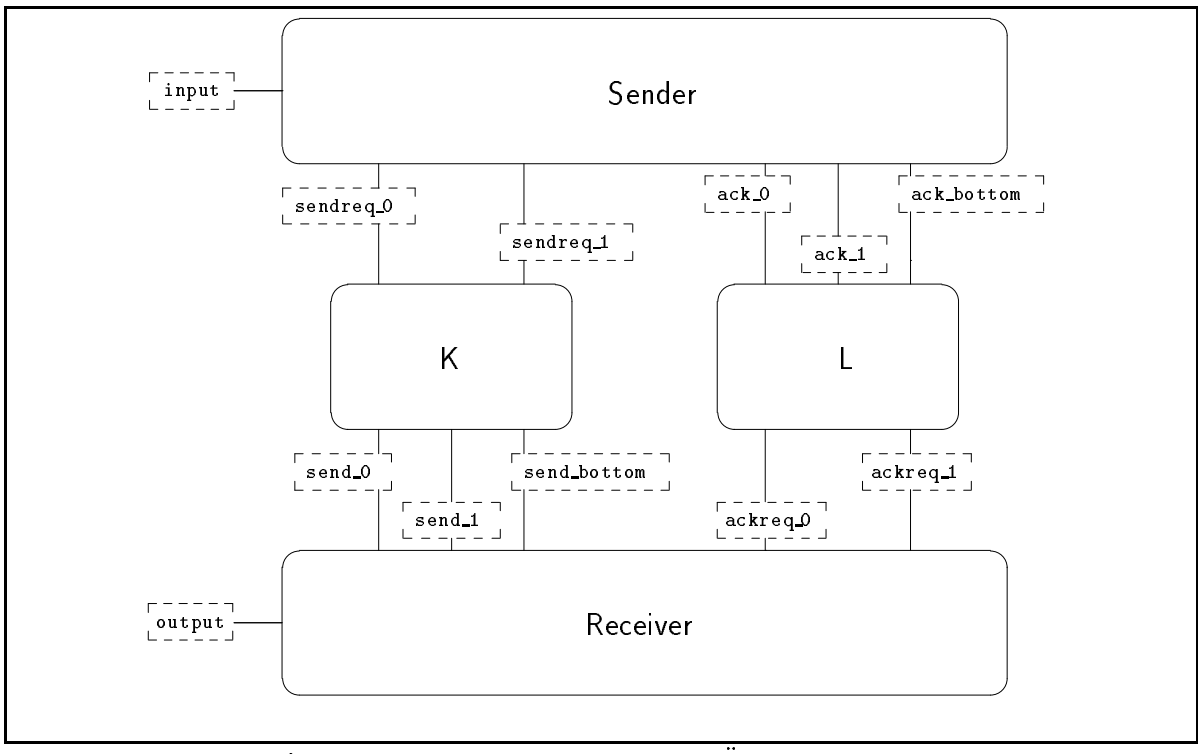


Abbildung 6.1: Eine schematische Übersicht des ABP

Sender und Empfänger nutzen nun dieses eine Bit, um die Datenübertragung auch im Fehlerfall sicherzustellen. Die Definition von Sender und Empfänger ist umfangreicher. Die gesamte Spezifikation in der von TRUTH verwendeten Syntax findet sich in Abbildung 6.2. Eine schematische Darstellung des Systems findet sich in Abbildung 6.1. Sender und Empfänger nutzen die Aktionen `input` und `output`, um Nachrichten von der Umgebung anzunehmen bzw. die Übermittlung einer Nachricht anzuzeigen.

Neben der Spezifikation des Systems müssen wir eine Reihe von Eigenschaften festlegen, die das System erfüllen soll. Diese müssen für TRUTH als Formeln des μ–Kalküls formuliert werde.

Eine wesentliche Forderung, die man in der Regel an ein verteiltes System stellt, ist, daß es nicht in einen Deadlock kommen kann. Ein Deadlock ist ein Zustand, von dem aus keine Schritte mehr möglich sind. Eine Formel, die beschreibt, daß kein erreichbarer Zustand existiert, der keine ausgehenden Transitionen besitzt, ist zum Beispiel:

$$NoDeadlock := AG(\langle - \rangle tt) \equiv \nu X.\langle - \rangle tt \wedge [-]X$$

Ein Livelock des Alternating Bit Protocols liegt dann vor, wenn es einen Zustand des Systems gibt, von dem aus eine unendliche Folge von τ–Schritten möglich ist. Eine Formel, die besagt, daß kein solcher Zustand existiert, ist beispielsweise:

$$NoLivelock := AG(\neg \nu X.\langle \tau \rangle X)$$

Das ABP ist in der Lage, eine Nachricht anzunehmen, wenn ein `input`–Schritt möglich ist, oder aber eine endliche Folge von τ–Schritten zu einem Zustand führt, von dem ein

```
def ABP = (Sender | K | L | Receiver) \ Internal
def Sender = S_0
def S_0 = 'input.S_0'
def S_1 = 'input.S_1'
def S_0' = sendreq_0.S_0''
def S_1' = sendreq_1.S_1''
def S_0'' = 'ack_0.S_1 + 'ack_1.S_0' + 'ack_bottom.S_0'
def S_1'' = 'ack_1.S_0 + 'ack_0.S_1' + 'ack_bottom.S_1'

def Receiver = R_0
def R_0 = 'send_0.output.ackreq_0.R_1
        + 'send_1.ackreq_1.R_0
        + 'send_bottom.ackreq_1.R_0
def R_1 = 'send_1.output.ackreq_1.R_0
        + 'send_0.ackreq_0.R_1
        + 'send_bottom.ackreq_0.R_1

def K = 'sendreq_0.(send_0.K + send_bottom.K)
      + 'sendreq_1.(send_1.K + send_bottom.K)

def L = 'ackreq_0.(ack_0.L + ack_bottom.L)
      + 'ackreq_1.(ack_1.L + ack_bottom.L)

def Internal = { sendreq_0, sendreq_1, send_0, send_1, send_bottom,
                 ackreq_0, ackreq_1, ack_0, ack_1, ack_bottom }
```

Abbildung 6.2: Die Spezifikation des ABP

solcher Schritt möglich ist.

$$CanInput := \mu X.\langle\overline{\texttt{input}}\rangle tt \vee \langle\tau\rangle X$$

Analog läßt sich eine Formel formulieren, die aussagt, daß das System eine Nachricht ausgeben kann.

$$CanOutput := \mu X.\langle\texttt{output}\rangle tt \vee \langle\tau\rangle X$$

Das System ist arbeitsbereit, wenn stets eine der beiden Formeln zutrifft.

$$Operating := AG(CanInput \vee CanOutput)$$

Eine wichtige Eigenschaft des Protokolls ist, daß keine Nachricht doppelt ausgegeben wird, oder daß eine Nachricht verlorengeht.

$$NoRepeat := AG([\overline{\texttt{input}}]\neg CanInput \wedge [\texttt{output}]\neg CanOutput)$$

Die entsprechenden Formeln in der Syntax von TRUTH haben folgende Form:

```
prop AG(P) = max X.P && [-]X
prop NoDeadlock = AG(<->tt)
prop NoLivelock = AG(max X.<tau>X)
prop CanInput = min Y.<'input>tt || <tau>Y
prop CanOutput = min Y.<output>tt || <tau>Y
prop Operating = AG (CanInput() || CanOutput())
prop NoRepeat = AG (['input](~CanInput()) || [output](~CanOutput()))
```

Mit diesen Definitionen in einer Datei „`abp`“ betrachten wir nun eine Beispielsitzung mit TRUTH.

```
The incredible Truth - System
(Version 0.6 --- December 1997) - ENJOY!

truth> file abp
```

TRUTH meldet sich mit einer Statusmeldung und erwartet eine Eingabe. Durch die Eingabe von „`file abp`“ wird die Datei mit den Definitionen in das System geladen.

```
truth> size ABP
The process ABP has 23 states and 28 transitions.
```

Mit dem Befehl `size` läßt sich ein erster Eindruck von der Größe des Systems gewinnen. Das hier betrachtete System ist mit 23 Zuständen eher klein.

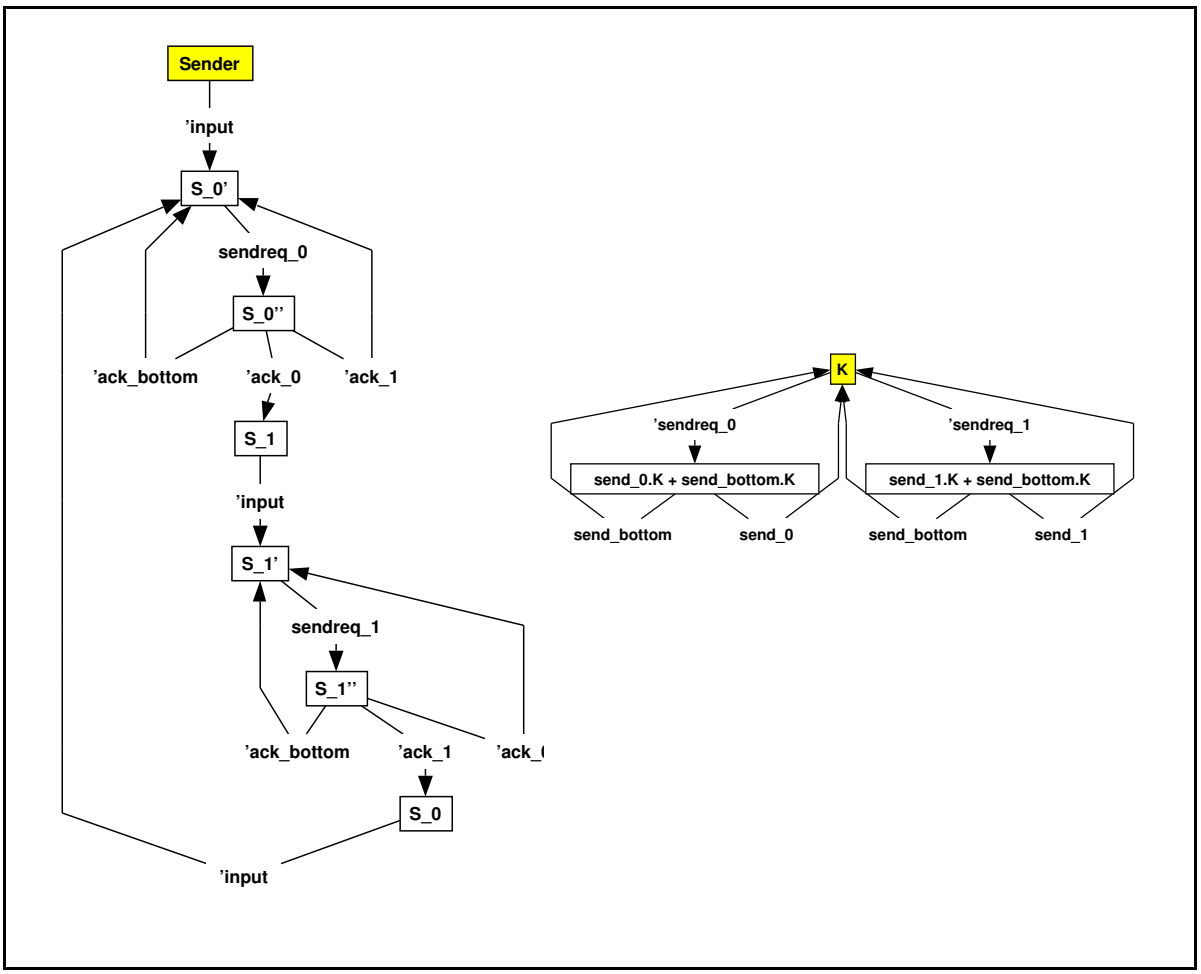


Abbildung 6.3: Ausgabe von Transitionssystemen mit *daVinci*

```
truth> plot Sender
truth> plot K
truth> plot ABP
```

Der Befehl `plot` stellt die Transitionssysteme, die sich als Semantik der verschiedenen Ausdrücke ergeben, graphisch dar. Die Ausgaben werden mit Hilfe von *daVinci* generiert. Eine Darstellung der Ausgaben findet sich in den Abbildungen 6.3 und 6.4.

Der Tableau–Modelchecker wird mit dem Befehl `tab` gestartet. Eine Überprüfung der oben beschriebenen Formeln ergibt die nächste Ausgabe.

```
truth> tab ABP NoDeadlock
TRUE, the process satisfies the formula
truth> tab ABP NoLivelock
FALSE, the process does not satisfy the formula
truth> tab ABP Operating
TRUE, the process satisfies the formula
truth> tab ABP NoRepeat
TRUE, the process satisfies the formula
```

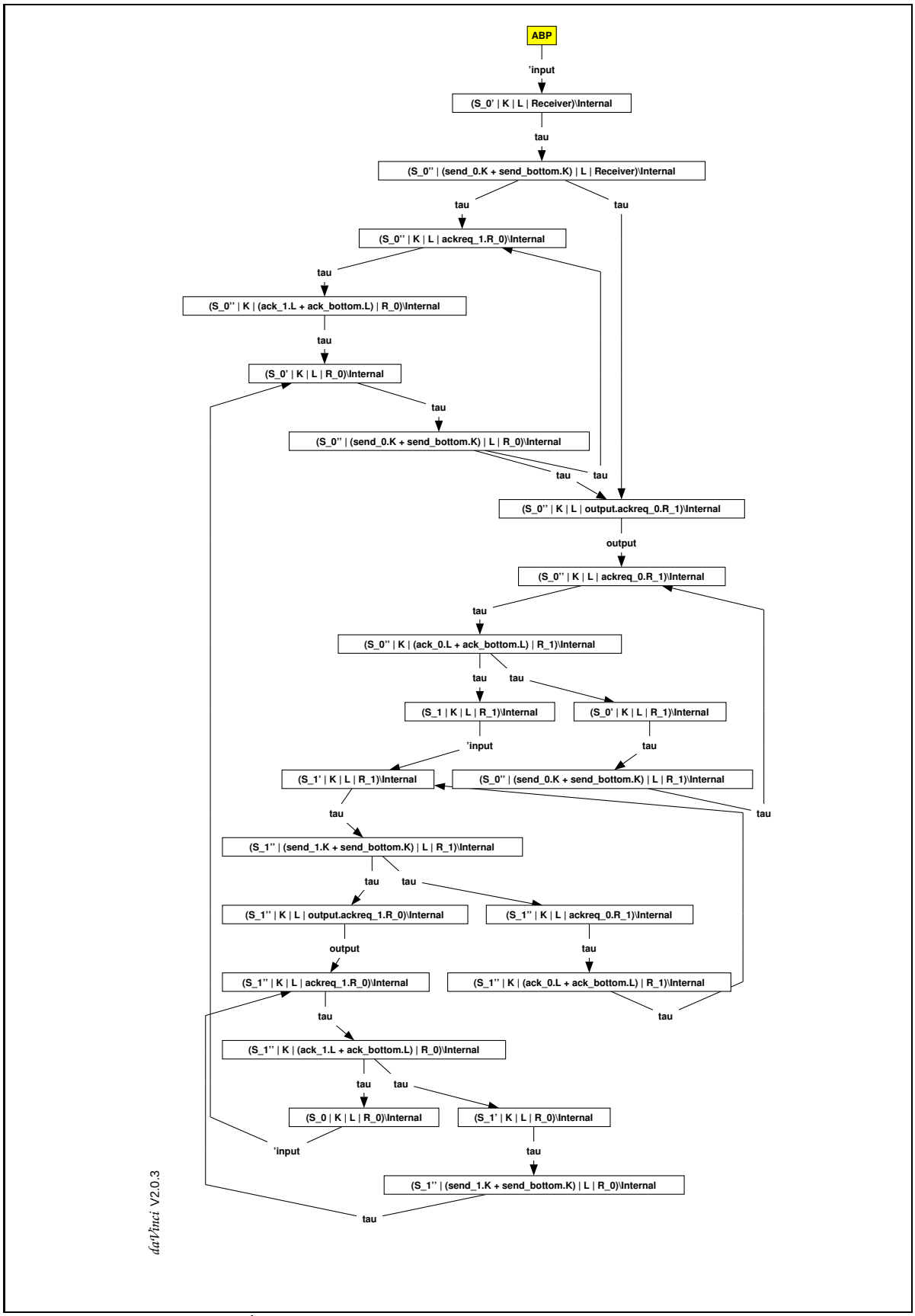


Abbildung 6.4: Das Transitionssystem des ABP

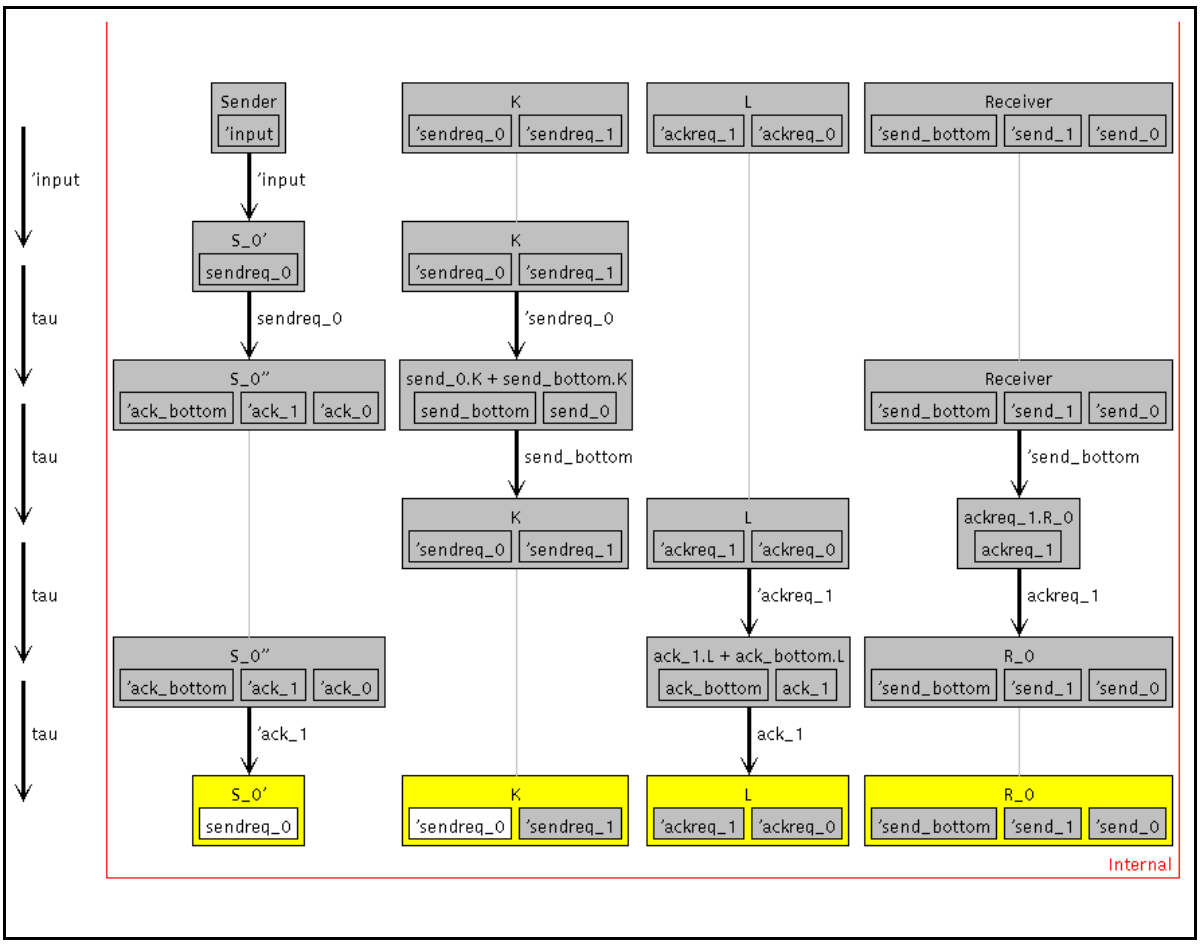


Abbildung 6.5: Simulation des ABP

Der Modelchecker ermittelt, daß die Formel *NoLivelock* nicht erfüllt ist. Eine Überprüfung des Systems mit Hilfe der Simulation bestätigt dies. Dazu muß zunächst eine Unterteilung in Teilprozesse vorgenommen werden.

```
truth> def SimABP = (Sender # K # L # Receiver) \ Internal
truth> sim SimABP
```

Wenn ein Kanal ständig falsch überträgt, so kommt es in der Tat zu einem unendlichen τ–Zykel. Abbildung 6.5 zeigt diesen Zykel. In der untersten Zeile liegt der gleiche Zustand vor wie in Zeile 2, diese Schleife läßt sich unendlich oft wiederholen. Eine mögliche Verbesserung des Systems, in der so etwas nicht mehr vorkommt, könnte zum Beispiel nur eine feste Anzahl fehlerhafter Übertragungen in Folge zulassen, um dann in einen Fehlerzustand zu wechseln.

Zusammenfassung und Ausblick

Diese Arbeit beschreibt die Architektur und die Implementierung von TRUTH, einem System zur automatischen Verifikation verteilter Systeme. Bei der Entwicklung von TRUTH wurde besonderer Wert auf seine leichte Modifizierbarkeit und Erweiterbarkeit gelegt, so daß es als Grundlage weitergehender Implementierungen dienen kann. Insbesondere die Verwendung anderer semantischer Bereiche zur Modellierung des Verhaltens verteilter Systeme scheint dabei neue Möglichkeiten bei der Verifikation zu eröffnen.

In der jetzigen Fassung ist TRUTH in der Lage, CCS–Spezifikationen zu verarbeiten und diesen eine Transitionssystem–Semantik zuzuordnen. Ein Abstraktionsmodul erlaubt den Austausch von CCS durch einen anderen Spezifikations–Formalismus mit geringem Aufwand. Die Implementierung eines Werkzeugs, das die notwendigen Funktionen und Datentypen für verschiedene Formalismen automatisch erzeugt, ist wünschenswert und bereits geplant.

Durch die Speicherung der Transitionssysteme mit Hilfe von veränderlichen Arrays zeigt TRUTH ein sehr gutes Laufzeitverhalten, das sich mit dem bereits vorhandener Werkzeugen durchaus messen kann. Da Transitionsysteme als abstrakte Datentypen mit einer kleinen Schnittstelle dargestellt werden, ist darüber hinaus ein Austausch dieser Komponente durch eine BDD–basierte Speicherung möglich. Solche Implementierungen speichern Transitionssysteme wesentlich kompakter ab und sind mit Erfolg bei der Verifikation sehr großer Systeme verwendet worden.

Letztendlich ist es jedoch wünschenswert, Transitionssysteme durch andere semantische Strukturen zu ersetzen, die die Nebenläufigkeit von Prozessen besser modellieren. Nur so kann das Problem der Explosion des Zustandsraums gelöst werden. Die jetzige Implementierung bildet einen geeigneten Ausgangspunkt für die Verwendung anderer semantischer Bereiche.

Das wichtigste Mittel zur Verifikation von verteilten Systemen mit TRUTH ist der eingebaute Modelchecker für den vollen μ–Kalkül. Der μ–Kalkül ermöglicht die Formulierung vieler Korrektheitseigenschaften, die durch den Modelchecker automatisch überprüft werden. Es wurde ein Tableau–basierte Modelchecking–Algorithmus implementiert, der für viele Formeln gute Laufzeiten erzielt. Spiel–basiertes Modelchecking hat sich jedoch als effizienterer Ansatz für Modelchecking für den μ–Kalkül herausgestellt, so daß eine Verwendung entsprechender Algorithmen eine deutliche Verbesserung

des Laufzeitverhaltens von TRUTH zur Folge hätte. Durch den modularen Aufbau von TRUTH ist eine solche Änderung ohne großen Aufwand vorzunehmen.

Eine neuartige Eigenschaft von TRUTH ist seine Simulationsfunktion. Diese ermöglicht eine graphische, interaktive, prozeß–orientierte Simulation von CCS–Prozessen. Darüber hinaus kann sie als Hilfmittel dienen, Gegenbeispiele eines spiel–basierten Modelcheckers darzustellen.

Die Wahl von `Haskell` als Programmiersprache bietet große Vorteile im Hinblick auf die Wartbarkeit und Änderbarkeit von TRUTH. Durch die Verwendung fortgeschrittener, funktionaler Programmierkonzepte wie Zustands–Monaden war es möglich, eine hohe Effizienz zu erreichen. Durch die Verwendung von Multi–Parameter–Klassen, wie sie in der Version 3.0 des `GHC` zur Verfügung stehen, wird es möglich sein, das Design von TRUTH deutlich zu vereinfachen. Dadurch wird eine noch leichtere Modifizierbarkeit des Systems erreicht werden.

A. Halbordnungen und Verbände

In Kapitel 4 werden einige elementare Sätze aus der Halbordnungstheorie verwendet. An dieser Stelle soll eine Notation vereinbart und diese Sätze kurz vorgestellt werden. Es wird eine Vertrautheit des Lesers mit dieser Materie vorausgesetzt, daher wird auf Beispiele und weitergehende Erläuterungen verzichtet.

Definition A.1 (Halbordnung) Es sei S eine nicht–leere Menge und $\leq \subseteq S \times S$. Gilt

$$\begin{aligned} \forall s \in S : s \leq s & \quad \text{Reflexivität} \\ \forall s_1, s_2, s_3 \in S : s_1 \leq s_2 \wedge s_2 \leq s_3 \Rightarrow s_1 \leq s_3 & \quad \text{Transitivität} \\ \forall s_1, s_2 \in S : s_1 \leq s_2 \wedge s_2 \leq s_1 \Rightarrow s_1 = s_2 & \quad \text{Antisymmetrie} \end{aligned}$$

so nennt man $(S, \leq)$ *Halbordnung.*

Es sei $(S, \leq)$ eine Halbordnung, $D \subseteq S, D \neq \emptyset$. Gilt

$$\forall s_1, s_2 \in D : \exists s_3 \in S : s_1 \leq s_3 \wedge s_2 \leq s_3$$

so heißt D *gerichtet.*

Es sei $\langle S, \leq \rangle$ eine Halbordnung und $D \subseteq S, a \in S$:

- a heißt *obere Schranke* von S, falls $\forall s \in D : s \leq a$.
- a heißt *kleinstes Element von* D, falls $a \in D$ und $\forall s \in D : a \leq s$.
- Besitzt die Menge $\{s \in S \mid x \text{ ist obere Schranke von D}\}$ ein kleinstes Element a, so heißt a *kleinste obere Schranke von* D oder auch *Supremum* von D. Als Schreibweise hierfür vereinbaren wir $a = \bigsqcup S$. $s_1 \sqcup s_2$ bezeichnet das Supremum von $D = \{s_1, s_2\}$, falls es existiert.

Kehrt man $\leq$ um und ersetzt „obere“ durch „untere“, „kleinste“ durch „größte“ sowie „Supremum“ durch „Infimum“ so erhält man die dualen Definitionen für *untere Schranke*, *größtes Element*, bzw. *größte untere Schranke.*

Definition A.2 Eine Halbordnung $(S, \leq)$ heißt *vollständig*, falls

1. es ein kleinstes Element $\bot \in S$ gibt und
2. $\bigsqcup D \in S$ für jede gerichtete Teilmenge $D \subseteq S$ existiert.

Eine vollständige Halbordnung nennen wir auch *CPO* für „complete partial ordering“.

Definition A.3 Eine Halbordnung $\langle S, \leq \rangle$ heißt *Verband*, falls

$$\forall s_1, s_2 \in S : \exists s_1 \sqcup s_2, s_1 \sqcap s_2 \in S.$$

Ein Verband heißt *vollständig*, falls alle Suprema und Infima beliebiger Teilmengen existieren.

Definition A.4 Es seien $\langle S, \leq \rangle, \langle S', \leq' \rangle$ CPOs und $f : S \to S'$.

- f heißt *monoton*, falls gilt $\forall s_1, s_2 \in S : f(s_1) \leq' f(s_2)$.
- f heißt *stetig*, falls für jede gerichtete Teilmenge $D \subseteq S$ auch deren Bild $f(D)$ gerichtet und $f(\bigsqcup D) = \bigsqcup' f(D)$ ist.

Satz A.5 (Fixpunktsatz von Tarski) Es sei $\langle S, \leq \rangle$ ein CPO und $f : S \to S$ eine stetige Funktion von S auf sich. Dann existiert

$$s_0 = \bigsqcup \{ f^{(i)}(\bot) \mid i \in \mathbb{N} \}$$

und es gilt

1. $f(s_0) = s_0$ und
2. $\forall s \in S : f(s) = s \Rightarrow s_0 \leq s$.

Beweis: Da f stetig ist, ist f notwendig auch monoton, und damit gilt:

$$\forall i \in \mathbb{N} : f^{(i)}(\bot) \leq f^{(i+1)}(\bot)$$

Damit ist $\bot, f(\bot), f^{(2)}(\bot), \ldots$ eine aufsteigende Kette. Sie ist offensichtlich gerichtet und ihr Supremum existiert.

1. Da f stetig ist, gilt:

$$\begin{aligned} f(s_0) &= f(\bigsqcup \{ f^{(i)}(\bot) \mid i \in \mathbb{N} \}) \\ &= \bigsqcup \{ f^{(i+1)}(\bot) \mid i \in \mathbb{N} \} \\ &= \bigsqcup \{ f^{(i)}(\bot) \mid i \in \mathbb{N} \} \\ &= s_0 \end{aligned}$$

2. Es sei s_1 ein beliebiger Fixpunkt von f. Dann gilt $\forall i \in \mathbb{N} : f^{(i)}(\bot) \leq f^{(i)}(s_1)$, da f monoton und $\bot \leq s_1$ ist. Damit folgt

$$s_0 = \bigsqcup\{f^{(i)}(\bot) \mid i \in \mathbb{N}\} \leq \bigsqcup\{f^{(i)}(s_1) \mid i \in \mathbb{N}\} = \bigsqcup\{s_1\} = s_1$$

□

Über vollständigen Verbänden läßt sich der Satz analog für maximale Fixpunkte von stetigen Funktionen formulieren. Die nach diesem Satz eindeutig bestimmten kleinsten und größten Fixpunkte einer solchen Funktion f bezeichnen wir mit μf bzw. νf.

Die Feststellung, daß über endlichen Verbänden alle monotonen Funktionen gleichzeitig stetig sind, ermöglicht die effektive Berechenbarkeit der kleinsten und größten Fixpunkte. Man erhält diese durch Fixpunkt–Iteration, ausgehend vom kleinsten bzw. größten Element des Verbandes.

B. Tests

TRUTH wurde anhand verschiedener Spezifikationen und Formeln gestestet und mit anderen Werkzeugen zur Verifikation verteilter Systeme verglichen. In diesem Anhang findet sich eine knappe Beschreibung der verwendeten Spezifikationen und Formeln.

2LN Die Verkettung von zwei Prozessen, die über unsichere Kanäle mit Hilfe des Alternating Bit Protocols kommunizieren.

3LN Das gleiche System mit 3 Prozessen.

Knuth Die Implementierung einer Lösung des Wechselseitigen Ausschluß–Problems nach Knuth.

802 Ein Teil der Implementierung des Data Link Control Layers gemäß IEEE 802.2.

Mailer Eine Spezifikation des Email–Systems, wie es an der Edinburgh University eine zeitlang im Einsatz war.

ATM Eine Spezifikation der Connect–Phase zweier ATM–Switches.

Für die Größe der Transitionssysteme, die sich aus den obigen Spezifikationen ergeben, erhält man die folgenden Werte:

	Zustände	Transitionen
2LN	1589	6819
3LN	44431	280456
Knuth	253	508
802	331	975
Mailer	1616	5928
ATM	1715	8305

Als logische Formeln wurden verschiedene Formeln des μ–Kalküls verwendet und über 2LN und 3LN interpretiert.

$$\Phi_1 = \mu X.[-]\mathrm{ff} \vee \langle - \rangle X$$
$$\Phi_2 = \mu X.\langle \overline{\texttt{receive}} \rangle \mathrm{tt} \vee \langle \tau \rangle X$$
$$\Phi_2' = \mu X.\langle \texttt{send} \rangle \mathrm{tt} \vee \langle \tau \rangle X$$
$$\Phi_3 = AG(\Phi_2 \vee \Phi_2')$$

$$\Phi_4 = AG([\texttt{send}]\neg\Phi_2 \vee [\overline{\texttt{receive}}]\neg\Phi_3)$$

$$\Phi_5 = AG([\texttt{send}](\nu X.\langle\overline{\texttt{receive}}\rangle\mathrm{tt} \vee ([\texttt{send}]\mathrm{ff} \wedge [-]X)) \wedge [\overline{\texttt{receive}}](\nu X.\langle\texttt{send}\rangle\mathrm{tt} \vee ([\overline{\texttt{receive}}]\mathrm{ff} \wedge [-]X))$$

$$\Phi_6 = \nu Y.\mu X.[-]((\langle\texttt{send}\rangle\mathrm{tt} \wedge Y) \vee X)$$

Es folgt eine informelle Angabe der Bedeutung der obigen Formeln.

- Φ_1 Es existiert ein erreichbarer Zustand, von dem aus keine Transitionen möglich sind.
- Φ_2 Es existiert ein τ–Pfad, auf dem irgendwann ein $\overline{\texttt{receive}}$–Schritt möglich ist.
- Φ_3 Das System arbeitet, das heißt zu jedem Zeitpunkt ist nach einer Reihe von τ–Schritten entweder ein `send`– oder ein $\overline{\texttt{receive}}$–Schritt möglich.
- Φ_4 Nach einem `send`–Schritt existiert kein τ–Pfad, der wieder zu einem `send`–Schritt führt. Darüber hinaus gilt die gleiche Aussage für einen $\overline{\texttt{receive}}$–Schritt.
- Φ_5 Eine schärfere Formulierung der obigen Aussage.
- Φ_6 Das System ist in der Lage, unendlich oft einen `send`–Schritt durchzuführen.

C. Quelldateien

In diesem Anhang wird die Organisation der Quelldateien auf der beiligenden Diskette und die Aufgaben der Module beschrieben.

- Verzeichnis \:
  - `Makefile`: Steuerdatei zur automatischen Übersetzung von TRUTH.
  - `truth`: Shell–Skript zum Aufruf von `truth.exe` mit den Standard–Parametern.
  - `truth.linux.exe`: Ausführbare Datei für Linux.
  - `truth.solaris.exe`: Ausführbare Datei für Solaris.
- Verzeichnis `basics`:
  - `Id.lhs`: Definition des Typs `Id`, der zur Repräsentierung von Bezeichnern in TRUTH verwendet wird. Es werden verschiedene Typklassen, wie `Hashable` und `Outputable` für `Id` implementiert.
- Verzeichnis `ccs`:
  - `CCSAction.lhs`: Definition des Datentyps zur Repräsentierung der Aktionen von CCS–Prozessen, Implementierung verschiedener Klassen für `CCSAction`.
  - `CCSEnv.lhs`: Definition der Umgebung zur Speicherung von CCS–Spezifikationen.
  - `CCSEqParser.y`: `Happy`–Spezifikation der Grammatik zum Parsern von CCS–Gleichungen.
  - `CCSLexer.lhs`: Definition des Scanners für CCS–Ausdrücke.
  - `CCSParser.lhs`: Funktionen, die den Zugriff auf die verschiedenen Parser regeln und dabei die Fehlerbehandlung realisieren.
  - `CCSProcess.lhs`: Der Datentyp zur Speicherung von CCS–Ausdrücken, sowie eine Reihe von Funktionen zu deren Verarbeitung inklusive der Implementierung der Klassen `Hashable` und `Outputable`.

  - `CCSProcParser.y`: `Happy`–Spezifikation der Grammatik zum Parsen von CCS–Ausdrücken.
  - `CCSRelabeling.lhs`: Definition des abstrakten Datentyps zur Speicherung von Umbenennungen und Funktionen zu deren Anwendung bei der Berechung der Einzelschritt–Ableitungen.
  - `CCSRestriction.lhs`: Definition des abstrakten Datentyps zur Speicherung von Restriktionen sowie Funktionen zu deren Anwendung bei der Berechnung der Einzelschritt–Ableitungen.
  - `CCSSOS.lhs`: Funktionen zur Berechung der Einzelschritt–Ableitungen von CCS–Ausdrücken mit Hilfe einer Hash–Tabelle.

- Verzeichnis `examples`: Beispiel–Spezifikationen, mit denen TRUTH getestet wurde.

- Verzeichnis `gui`:
  - `AlertDialog.java`: Hilfsklasse zur Darstellung eines Fensters mit einem Button, durch den das Fenster wieder geschlossen wird.
  - `Arrow.java`: Klasse zur Repräsentierung der Pfeile in der graphischen Darstellung.
  - `ConfirmDialog.java`: Hilfklasse, die eine String in einem Fenster darstellt und eine Auswahl mit zwei Buttons ermöglicht.
  - `GUI.java`: Hauptdatei, die das Layout definiert und die Kommunikation mit TRUTH realisiert.
  - `ImageCanvas.java`: Klasse, die im Konstruktor eine URL einer Graphik erwartet und eine Canvas erzeugt, die diese Graphik enthält.
  - `SimAction.java`: Klasse zur Repräsentierung von Aktionen.
  - `SimArena.java`: Klasse zur Darstellung der Simulation innerhalb einer Canvas, die auch die Mauseingaben verarbeitet.
  - `SimAreneElement.java`: Abstrakte Klasse zur Darstellung der verschiedenen Möglichkeiten eines Prozesses während der Simulation.
  - `SimBox.java`: Klasse zur Darstellung eines Prozesses mit seinen Aktionsmöglichkeiten.
  - `SimBoxAction.java`: Klasse zur Darstellung einer Aktion innerhalb einer `SimBox`.
  - `SimBoxHistory.java`: Klasse zur Speicherung des Verlaufs der Simulation für einen Prozeß.
  - `SimChoice.java`: Klasse zur Verarbeitung der Auswahlmöglichkeiten des Benutzers während der Simulation.

  - `SimResArea.java`: Klasse, die die Darstellung der Details einer Restriktion bei überfahren durch den Mauszeiger realisiert.
  - `Timeline.java`: Klasse zur Darstellung der globalen Aktionen des simulierten Systems.
  - `Unique.java`: `Java`–Repräsentierung von `Unique`–Werten.

- Verzeichnis `logic`:
  - `FLClosure.lhs`: Berechung der Fisher–Ladner Hülle einer Formeln aus $L\mu$, gleichzeitig Umwandlung in den Typ `FastFormula`, der auch in diesem Modul definiert wird.
  - `MuEnv.lhs`: Umgebung zur Speicherung von Formeldefinitionen in TRUTH, Ersetzung der Formelmakros.
  - `MuFormula.lhs`: Definition des Datentyps `GenMuFormula` zu Repräsentierung von Formeln von $L\mu$, einige Funktionen wie Substitution und Berechnung einiger Normalformen.
  - `MuLexer.lhs`: Scanner zur lexikalischen Verarbeitung von Formeln aus $L\mu$.
  - `MuParser.y`: `Happy`–Spezifikation einer Grammatik zum Parsern von Formeln aus $L\mu$.
  - `MuTableau.lhs`: Implementierung des Tableau–basierten Modelcheckers basierend auf der Transitionssystem–Semantik.

- Verzeichnis `lts`:
  - `Analysis.lhs`: Implementierung verschiedener Analysen, die sich auf die Transitionssystem–Semantik stützen, z. B. die Ausgabe von Transitionssystemen mit *daVinci* und die Suche nach Deadlocks.
  - `LTSClosure.lhs`: Bestimmung der τ–Hülle von Zuständen im Transitionssystem für die Interpretierung der schwachen Modalitäten $\langle\langle - \rangle\rangle$ und $[[-]]$.
  - `LTSLabeling.lhs`: Definition der Typklasse `LTSLabeling`, die benutzt wird, bedarfsgesteuerte Generierung von Transitionssystemen zu ermöglichen.
  - `LTSMonad.lhs`: Definition der Monade, die für Algorithmen benutzt wird, die sich auf Transitionssysteme stützen.
  - `LTSState.lhs`: Definition des abstrakten Datentyps `LTSState`, der zur Identifizierung von Zuständen in den Transitionssystemen benutzt wird.
  - `MutLTS.lhs`: Definition des abstrakten Datentyps `MutLTS`, der zur Darstellung von Transitionssystemen im Rahmen einer Zustandsmonade verwendet wird.
  - `PAToLTS.lhs`: Funktionen, die die Generierung von Transitionssystemen auf Grundlage der Einzelschritt–Ableitungen realisieren.

- Verzeichnis `main`:
  - `Main.lhs`: Definition der `main`–Funktion, Ein– und Ausgabe–Funktionen, Treiber–Funktionen für die verschiedenen Befehle.
  - `MainEnv.lhs`: Umgebung für die Treiber–Funktionen.
  - `Options.lhs`: Verarbeitung der Optionen, die bei Programmstart angegeben werden können.
- Verzeichnis `pa`:
  - `Derivatives.lhs`: Funktionen zur Berechnung verschiedener Ableitungen von Prozessen basierend auf der Einzelschritt-Ableitung.
  - `ProcAlg.lhs`: Definition der Typen und Funktionen zur Implementierung verschiedener Spzifikations–Formalismen.
  - `Simulation.lhs`: Steuerung der interaktiven Simulation von Prozessen.
- Verzeichnis `utils`:
  - `Error.lhs`: Eine einfache Monade zur Fehlerbehandlung.
  - `Exec.lhs`: Starten von anderen Prozessen über die Posix–Bibliotheken, Kommunikation über Pipes, Signal–Handler.
  - `Hash.lhs`: Definition der Typklasse `Hashable`, Implementierung von Hash–Tabellen mit Zustands–Monaden.
  - `Outputable.lhs`: Definition der `Outputable`–Typklasse, Definition einiger Standard–Instanzen.
  - `SST.lhs`: Die Zustands–Monade, die in TRUTH verwendet wird, sowie verschiedene Kombinatoren für diese Monade.
  - `UniqSupply.lhs`: Definition des Typs `UniqSupply`, der die Vergabe von `Unique`–Werten koordiniert sowie eine Monade zur komfortablen Benutzung dieses Typs.
  - `Unique.lhs`: Definition des Typs `Unique` und der Typklasse `Uniquable.lhs`, einige Hilfsfunktionen für `Unique`

Literaturverzeichnis

[Ada93] Stephen Adams. Efficient sets: a balancing act. *Journal of functional programming*, 3(4):553–562, October 1993.

[BC96a] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal μ–calculus. *Lecture Notes in Computer Science*, 1055:107–??, 1996.

[BC96b] Girish Bhat and Rance Cleaveland. Efficient model checking via the equational μ-calculus. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[Bra96] J. C. Bradfield. The modal mu-calculus alternation hierarchy is strict. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 233–246, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[Bry86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[CMS95] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. *Lecture Notes in Computer Science*, 1019:153–??, 1995.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CS92] R. Cleaveland and B. Steffen. A linear–time model–checking algorithm for the alternation–free modal mu–calculus. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91)*, volume 575 of *LNCS*, pages 48–58, Berlin, Germany, July 1992. Springer.

[CS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. *Lecture Notes in Computer Science*, 1102:394–??, 1996.

[CW96] Edmund M. Clarke and Jeanette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[DB95] A. Dsouza and B. Bloom. Generating BDD models for process algebra terms. *Lecture Notes in Computer Science*, 939:16–??, 1995.

[EC81] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *Proceedings of the Seventh International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181, Berlin, 1981. Springer-Verlag.

[EC86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mucalculus. In *Proccedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, Washington, D.C., 1986. IEEE Computer Society Press.

[Eme96] E. A. Emerson. Automated temporal reasoning about reactive systems. *Lecture Notes in Computer Science*, 1043:41–??, 1996.

[Eme97] E. A. Emerson. *Model checking and the mu-calculus*, volume 31 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, chapter 6. American Mathematical Society, 1997.

[FJ96] S. Finne and S. P. Jones. Composing the user interface with Haggis. *Lecture Notes in Computer Science*, 1129:1–??, 1996.

[FW94] M. Fröhlich and M. Werner. The graph visualization system daVinci - A user interface for applications. Technical Report 5/94, Department of Computer Science; University of Bremen, September 1994.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, ? 1997.

[GPVW95] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. Internal Report CSR-69-80, University of Edinburgh, Department of Computer Science, September 1980.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, December 1996.

[Hoa83] C. A. R. Hoare. Notes on CSP. Technical Report 83-4, Dept. Computer Science, University of Wollongong, March 1983.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Hug95] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[JHH+93] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology Technical Conference*, Keele, 1993.

[JL91] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In R. J. M. Hughes, editor, *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, NY, 1991. Springer-Verlag. Lecture Notes in Computer Science 523.

[JNR97] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green card: a foreign–language interface for haskell. Technical report, Glasgow University, February 1997.

[KL95] D. King and J. Launchbury. Structuring depth first search algorithms in haskell. In *Principles of Programming Languages*, San Francisco, 1995.

[Koz82] Dexter Kozen. Results on the propositional μ-calculus. In Mogens Nielsen and Erik Meineche Schmidt, editors, *Automata, Languages and Programming, 9th Colloquium*, volume 140 of *Lecture Notes in Computer Science*, pages 348–359, Aarhus, Denmark, 12–16 July 1982. Springer-Verlag.

[LJ94] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

[Mar97] Simon Marlow. Happy user guide. Technical report, Glasgow University, December 1997.

[Mes90] José Meseguer. Rewriting as a unified model of concurrency. In *Proceedings, Concur'90 Conference*, Lecture Notes in Computer Science, Volume 458, pages 384–400, Amsterdam, August 1990. Springer. Also, Report SRI-CSL-90-02R, Computer Science Lab, SRI International.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.

[MPW91] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR'91*, LNCS 527, pages 45–60, Amsterdam, August 1991. Springer-Verlag.

[Obj97] Objectspace Inc. JGL online documentation. WWW, `http://www.objectspace.com/jgl/documentation.html`, 1997.

[PH+96] John Peterson, Kevin Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, May 1996.

[Plo81] Gordon D Plotkin. A Structural Approach to Operational Semantics. Tech. Rep. FN-19, DAIMI, Univ. of Aarhus, Denmark, September 1981.

[PW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1992*, pages 71–84, New York, NY, USA, 1993. ACM Press.

[SJ95] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 355–366, San Francisco, California, January 22–25, 1995. ACM Press.

[Sto95] Anthony L. Stornetta. Implementation of an efficient parallel bdd package. Master's thesis, University of California, Santa Barbara, 1995.

[WN93] Glynn Winskel and Mogens Nielsen. Models for concurrency. Technical report, Basic Research in Computer Science (BRICS), Aarhus, 1993. IB-B941066.

[Zuk97] John Zukowski. *Java AWT Reference*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1997.