Technische Universität Dresden
Fakultät Informatik

Bakkalaureatsarbeit

zum Thema

Überwachung von Java-Programmen mittels Java PathFinder

vorgelegt von
Matthias Fruth
geboren am 30. November 1979 in Dresden

eingereicht am 12. September 2002

Betreuender Hochschullehrer: Prof. Dr. rer. nat. habil. Horst Reichel

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Moderne Informationssysteme werden immer komplexer. Es gibt bestimmte Klassen von Fehlern, die mit den klassischen Validierungsverfahren Test und Simulation praktisch nicht gefunden werden können. Dagegen können formale Verifikationsverfahren alle möglichen Verhaltensweisen eines Systems überprüfen und sie liefern beweisbar richtige Resultate.

Im Vergleich zur Überprüfung von Hardware und Entwürfen gilt die formale Verifikation von Software nach wie vor als schwierig. Hauptproblem ist die effiziente Modellierung dynamischer Datenstrukturen, die potentiell unendlich groß sind.

Deshalb besteht der Wunsch nach einer weitgehend automatisierten und leistungsfähigen Verifikationsumgebung zur direkten Überprüfung von Quellkode für den regelmäßigen Einsatz durch Softwareingenieure. Diese würde zu einer höheren Geschwindigkeit und Qualität in der Softwareentwicklung beitragen. Das gilt um so mehr im Bereich von Rapid Prototyping und Extreme Programming, wo der Kode oftmals zugleich Entwurf und Dokumentation ist. Standardisierte Programmiersprachen ermöglichten dann Vergleiche von Verifikationswerkzeugen.

1.2 Stand der Forschung

Im Mittelpunkt dieser Arbeit steht das Projekt Java PathFinder[1] der Automated Software Engineering Group des NASA Ames Research Center. Nachdem die Version JPF1 [18] eine Übersetzung eingeschränkter Java-Programmen in die Modellierungssprache des Model Checkers Spin ermöglichte, ist JPF2 [4, 32] ein vollwertiger Model Checker. Inzwischen wird bereits am Nachfolger Java PathExplorer [19] gearbeitet.

Damit vergleichbare Ansätze finden sich in den Projekten Bandera[2] von der Kansas State University und Slam[3] von Microsoft Research.

[1] `http://ase.arc.nasa.gov/jpf/`
[2] `http://www.cis.ksu.edu/bandera/`
[3] `http://www.research.microsoft.com/slam/`

Bandera [14, 11] übersetzt Java-Quellkode in die Modellierungssprachen der Model Checker PVS, Spin und SMV. Einige Teile dieses Projektes werden auch von JPF genutzt.

Slam [2, 1] ist ein Model Checker für Sicherheitseigenschaften von C-Programmen.

1.3 Diese Arbeit

In dieser Arbeit soll das System JPF2, kürzer JPF, theoretisch und praktisch untersucht werden. Ziele dieser Arbeit sind:

- Darstellung der theoretischen Grundlagen und wesentlichen Algorithmen von JPF
- Demonstration der Leistungsfähigkeit von JPF auf der Basis repräsentativer Beispiele
- Abschätzung der prinzipiellen und praktischen Grenzen von JPF
- Vorschläge für weiterführende Arbeiten

Im nächsten Kapitel werden die wesentlichen Grundlagen nebenläufiger Systeme sowie der nebenläufigen Programmierung in Java zusammenfassend dargestellt. Kapitel 3 erläutert die theoretischen Grundlagen von Model Checking, um daraus schließlich einen Verifikationsalgorithmus zu entwickeln. Im 4. Kapitel werden die von Java PathFinder verwendeten Effizienzverbesserungen zum Model Checking diskutiert. Kapitel 5 beschäftigt sich mit dem Systemaufbau von JPF und liefert eine kritische Bewertung der erhaltenen Ergebnisse. Zum Schluß faßt Kapitel 6 die wichtigsten Resultate zusammen und zeigt Möglichkeiten für weiterführende Arbeiten auf.

Kapitel 2

Nebenläufigkeit in JAVA

Im ersten Abschnitt sollen wichtige Begriffe sowie interessante Probleme nebenläufiger Systeme thematisiert werden. Der zweite Abschnitt beschäftigt sich mit Möglichkeiten der nebenläufigen Programmierung in der Programmiersprache JAVA.

Grundkenntnisse über die wesentlichen Konzepte von JAVA werden vorausgesetzt. Für eine Einführung in JAVA sei auf das reichhaltige Angebot an Fachliteratur verwiesen, zum Beispiel [26]. Die offizielle Spezifikation der Programmiersprache ist in [15] zu finden. In dieser Arbeit wird von JAVA in der Version 1.3.1_04 ausgegangen.

2.1 Systematik nebenläufiger Systeme

In Ausführung befindliche Programme werden im Betriebssystem als *Prozesse* dargestellt. Darunter versteht man unter anderem den Programmzähler, zugewiesene Speicherbereiche sowie Prozeßverwaltungsinformationen. Ein Prozeß kann die Zustände *rechnend*, *rechenbereit* und *wartend* annehmen. Ein System mehrerer Prozesse, daß nicht sequentiell ist, heißt *nebenläufiges System*.

Mit verschiedenen Algorithmen bestimmt das Betriebssystem die Zuteilung der Betriebsmittel (Rechenzeit, Speicher) zu den Prozessen. Diese Vorgänge werden unter dem Begriff *Scheduling* zusammengefaßt. Um mehrere Programme gleichzeitig ablaufen zu lassen, führt das Betriebssystem schnelle Wechsel zwischen den entsprechenden Prozessen durch. Bei einem Wechsel wird die Anweisung, an der das Programm unterbrochen wurde, für eine spätere Fortsetzung gespeichert. Diese Stelle ist im allgemeinen nicht vorhersagbar, da Schedulingverfahren nichtdeterministisch sind. Für das Scheduling atomare Operationen eines Prozesses werden *Aktionen* genannt.

Da von modernen Programmen häufig mehrere Aufgaben gleichzeitig erledigt werden sollen, ist es sinnvoll, diese – soweit das möglich ist – parallel zueinander ablaufen zu lassen. Während die Adreßräume verschiedener Prozesse voneinander getrennt sind, können verschiedene zu einem Prozeß gehörige Kontrollflüsse dessen Adreßraum gemeinsam nutzen (sie besitzen aber in JAVA getrennte Namensräume). Nebenläufige Systeme dieser sogenannten *Threads* werden mit dem Wort *Multithreading* beschrieben. Ein System, das nicht ter-

minierend ist, sondern neue Kontrollflüsse erzeugen kann, wird *reaktives System* genannt.

Beim parallelen Zugriff mehrerer Prozesse auf eine gemeinsame Variable, von denen wenigstens einer schreibend ist, können abhängig von der Zugriffsreihenfolge unterschiedliche lokale Werte der Variablen auftreten. Man spricht hier von *zeitkritischen Abläufen* [30].

Definition 2.1 (zeitkritischer Ablauf)
Seien P_1 und P_2 beliebige Prozesse. Seien $\mathcal{R}$ und $\mathcal{W}$ Funktionen von der Menge aller Prozesse auf die Menge aller Datenelemente[1]*, so daß $\mathcal{R}(P)$ und $\mathcal{W}(P)$ die Mengen aller von P gelesenen bzw. geschriebenen Datenelemente angeben. Falls*

$$\mathcal{W}(P_1) \cap (\mathcal{R}(P_2) \cup \mathcal{W}(P_2)) \neq \emptyset,$$

wird die Ausführung von P_1 und P_2 ***zeitkritischer Ablauf*** *genannt.*

Für diese *kritischen Bereiche* muß daher eine Synchronisation erfolgen, so daß nur lesende Zugriffe gleichzeitig erfolgen können. Dieses Prinzip wird als *wechselseitiger Ausschluß* bezeichnet. Dafür übliche Methoden sind im Betriebssystem implementierte *Monitore* oder *Sperrvariablen*.

Bei der Synchronisation von Prozessen kann es vorkommen, daß diese für immer blockiert werden. Dieser Zustand heißt *Verklemmung* [30].

Definition 2.2 (Verklemmung)
Eine Menge von Prozessen befindet sich in einem ***Verklemmungszustand****, falls jeder Prozeß dieser Menge auf ein Ereignis wartet, das nur ein anderer Prozeß dieser Menge auslösen kann.*

Es ist klar, daß diese Probleme von Prozessen analog auch bei Threads existieren. Methoden zur Synchronisation von Threads müssen von der jeweiligen Programmiersprache bereitgestellt werden.

2.2 Multithreading in JAVA

2.2.1 Die Klasse `Thread`

Zur nebenläufigen Programmierung besitzt JAVA die vordefinierte Klasse `Thread`. Die möglichen Zustände eines Threads und zur Zustandsüberführung verfügbare Methoden sind in Abbildung 2.1 zu sehen. Die wichtigsten Methoden der Klasse `Thread` werden im folgenden kurz vorgestellt [26]:

- Der Operator `new` erzeugt ein Exemplar der Klasse `Thread` (also einen neuen Thread). Die Methode `run()` enthält die vom Thread auszuführenden Anweisungen.
- `start()` überführt den Thread in den Zustand rechenwillig.
- Durch Aufruf von `yield()` gibt der Thread die CPU freiwillig ab.

[1] Dieser Begriff faßt Objekte und primitive Datenelemente zusammen.

- `sleep()` setzt den Thread für einen bestimmten Zeitraum aus.
- `join()` blockiert den Thread, wenn er auf die Beendigung eines anderen Threads warten soll.
- `wait()` blockiert den Thread.
- `notify()` benachrichtigt einen blockierten Thread.
- `notifyAll()` benachrichtigt alle blockierten Threads.
- `interrupt()` unterbricht durch `sleep`, `join` oder `wait` blockierte Threads, so daß diese wieder rechenwillig werden.

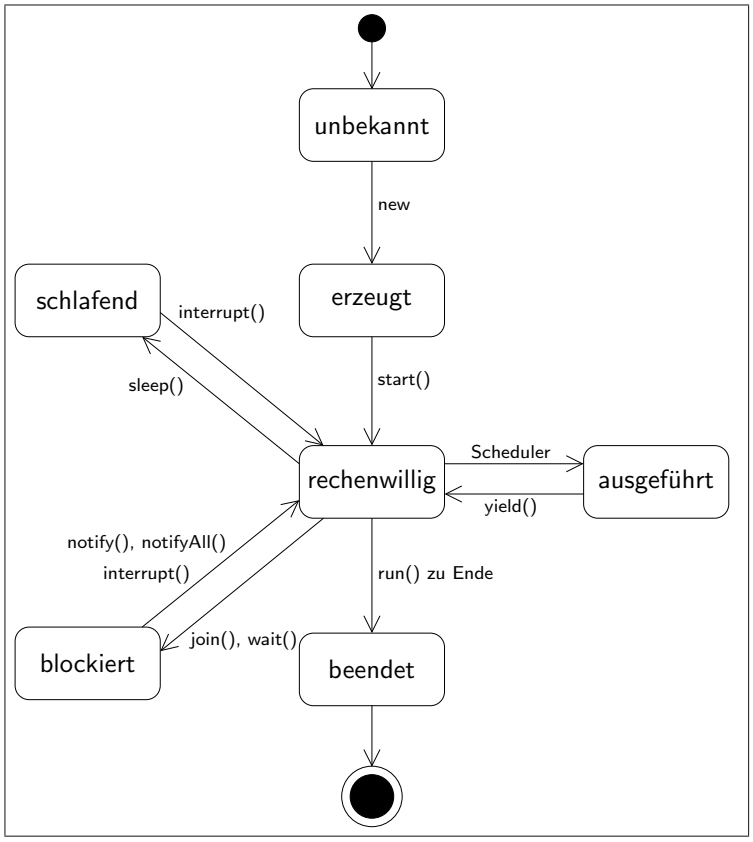

Abbildung 2.1: Zustände eines Threads in JAVA [26]

2.2.2 Synchronisation von Threads

JAVA bietet mehrere Möglichkeiten zur Synchronisation nebenläufiger Threads. Zum einen gibt es Monitore, welche durch den Methodenmodifier `synchronized` deklariert werden. Synchronisation durch *Sperrobjekte* erlaubt die Methode `synchronized()`, die jedes beliebige Objekt als Sperrobjekt verwenden kann. Es ist außerdem möglich, durch die Methoden `join`, `wait`, `notify` und `notifyAll` Threads mit Ereignissen zu synchronisieren.

Kapitel 3

Model Checking

In diesem Kapitel werden die theoretischen Grundlagen und algorithmischen Verfahren, auf denen Model Checking basiert, formal dargestellt. Um eine Einordnung von Model Checking in das Gebiet der Verifikation zu ermöglichen, gibt 3.1 eine überblicksmäßige Klassifikation von Verifikationsverfahren an. Anschließend wird in 3.2 die Modellierung reaktiver Systeme erörtert. In 3.3 werden interessante Eigenschaften nebenläufiger Systeme betrachtet und mit temporalen Logiken formalisiert. Zum Abschluß zeigt 3.4 die bei Model Checking auftretenden Probleme sowie mögliche Lösungsansätze und es wird ein Model-Checking-Algorithmus konstruiert.

3.1 Einordnung

Verifikation umfaßt im wesentlichen zwei Bereiche. Test und Simulation gehören zu den *Validierungsverfahren*. Diese Verfahren sind gut skalierbar und finden bereits mit geringem Aufwand viele Fehler in Entwurf und Implementierung. Sie sind in Ausbildung und Praxis weit verbreitet und werden gut durch Softwarewerkzeuge unterstützt. Allerdings können bestimmte Klassen von Systemeigenschaften kaum durch Validierung überprüft werden. Im Gegensatz dazu stehen die *formalen Verifikationsverfahren*. Diese sind – basierend auf mathematischer Theorie – in der Lage, sämtliche Systemeigenschaften beweisbar zu überprüfen und arbeiten automatisch. Nachteilig sind ihre hohen Anforderungen an Laufzeit und Speicherplatz[1], auch ihre Handhabung ist vergleichsweise schwierig. Im Entwurf komplexer sequentieller Systeme (insbesondere von Hardware und Protokollen) sind sie bereits erfolgreich und ihre Verbreitung steigt [10]. Es besteht aber nach wie vor großer Bedarf an ausgebildeten Experten sowie unterstützenden Softwarewerkzeugen. Die Leistungsfähigkeit formaler Verifikationsverfahren kann durch den Einsatz formaler Entwurfsmethoden noch erhöht werden. Man faßt formale Verifikations- und Entwurfsmethoden unter dem Begriff *formale Methoden* [10] zusammen.

Formale Verifikationsverfahren bestehen im wesentlichen aus drei Teilen: einer Modellierungssprache zur Beschreibung des Systems, einer Spezifikati-

[1] Dieses sogenannte *Zustandsexplosionsproblem* tritt insbesondere bei nebenläufigen und arithmetisch komplexen Systemen auf, mehr dazu in 3.4.1.

onssprache zur Formulierung von erwünschten (und unerwünschten) Systemeigenschaften und dem eigentlichen Verifikationsalgorithmus. Für die Spezifikation können sowohl funktionelle Eigenschaften und Timingeigenschaften als auch Leistungscharakteristik oder interne Struktur des Systems von Bedeutung sein. Hier sollen ausschließlich Verhaltenseigenschaften untersucht werden, es gibt aber auch Ansätze für Nichtverhaltenseigenschaften wie Leistungsvermögen, Echtzeitanforderungen, Sicherheit und Architektur des Systems [10]. Man unterscheidet formale Verifikationsverfahren in *Theorembeweisen* und *Model Checking*. Während erstere beweisbasiert und daher schwer automatisier- und handhabbar sind, handelt es sich bei Model Checking um ein modellbasiertes, weitgehend automatisches und verglichen mit Theorembeweisen relativ gut handhabbares Verfahren. Model Checking ist besonders erfolgreich bei der Verifikation nebenläufiger, reaktiver Systeme. Es kann direkt nach der Testphase, auch im Entwurf, eingesetzt werden, da partielle Spezifikationen möglich sind. Wenn die Verifikation einer Spezifikation fehlgeschlägt, kann Model Checking dazu plausible Gegenbeispiele konstruieren.

In dieser Arbeit soll *Temporal Logic Model Checking* [10, 9, 21] im Mittelpunkt stehen. Dabei erfolgt die Systemmodellierung durch spezielle Graphenstrukturen und die Spezifikation der Eigenschaften mittels temporaler Logiken. Andere Ansätze verfolgt *Behavior Conformance Checking* [10, 9]. Dabei werden sowohl Modell als auch Eigenschaften durch spezielle Automaten dargestellt und deren Verhalten bzw. akzeptierte Sprachen untersucht. Folgend ist mit Model Checking immer Temporal Logic Model Checking gemeint.

3.2 Modellierung reaktiver Systeme

Während man sich bei terminierenden Systemen hauptsächlich für die Eingabe-Ausgabe-Relation interessiert, die zum Beispiel im HOARE-Kalkül (siehe Kapitel 4 in [21]) beschrieben werden kann, sind bei reaktiven Systemen Aussagen über Systemzustände und Berechnungspfade interessant. Daher bietet sich hier eine Zustandsmodellierung an, wobei Zustände des Modells Werte von Datenelementen und Transitionen Auswirkungen von Aktionen im System repräsentieren.

Um Model Checking auf reaktive Systeme anwenden zu können, beschreibt man diese durch markierte Transitionsgraphen, sogenannte KRIPKE-Strukturen [8, 9, 21].

Definition 3.1 (KRIPKE**-Struktur)**
Eine KRIPKE-***Struktur*** $\mathcal{M} = (S, S_0, \rightarrow, L)$ *über einer Menge* $\mathcal{AP}$ *atomarer Aussagen ist wie folgt definiert:*

1. S *ist die endliche* ***Zustandsmenge****.*
2. $S_0 \subseteq S$ *ist die* ***Startzustandsmenge****.*
3. $\rightarrow \subseteq S \times S$ *ist die* ***Transitionsrelation****, wobei für jedes* $s \in S$ *ein* $s' \in S$ *mit* $s \rightarrow s'$ *existiert.*

4. $L : S \to \mathfrak{P}(\mathcal{AP})$ *ist die* ***Beschriftungsfunktion****, die jedem Zustand die Menge der in ihm gültigen atomaren Formeln zuordnet.*

Auf die Angabe der Startzustandsmenge kann manchmal verzichtet werden. KRIPKE-Strukturen lassen sich gut durch gerichtete Graphen darstellen.

Beispiel [21] Eine KRIPKE-Struktur sei durch die Zustandsmenge $S = \{s_0, s_1, s_2\}$, die Transitionsrelation $\to = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$ und die Beschriftungsfunktion $L : s_0 \mapsto \{p, q\}, s_1 \mapsto \{q, r\}, s_2 \mapsto \{r\}$ gegeben. Der dazugehörige gerichtete Graph mit der Knotenmenge S, der Kantenmenge $\to$ und den Knotenmarkierungen gemäß L ist in Abbildung 3.1 dargestellt.

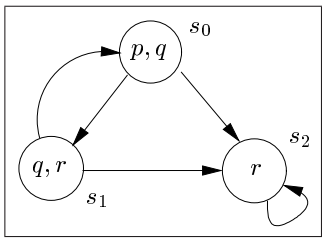

Abbildung 3.1: Graph einer KRIPKE-Struktur [21]

3.3 Spezifikation von Systemeigenschaften

Eigenschaften reaktiver Systeme werden bezüglich ihrer Zustände und Transitionen formuliert. So ist beispielsweise interessant, ob ein Zustand s von einem anderen Zustand s' aus überhaupt, mehrmals oder unendlich oft erreicht bzw. durchlaufen werden kann und auf welchen Pfaden dies geschieht. Bei der Untersuchung dieser Pfade ist insbesondere die Reihenfolge der durchlaufenen Zustände und Transitionen von Interesse.

3.3.1 Temporale Logiken

Um Eigenschaften einzelner Berechnungen zu beschreiben, spricht man über deren Pfade, man formuliert *Pfadformeln*. Aussagen über die Existenz bestimmter Pfade sowie über alle von einem Zustand ausgehenden Pfade kann man mittels *Zustandsformeln* treffen. Um solche Eigenschaften zu formulieren nutzt man *temporale Logiken*.

Die Computation Tree Logic CTL*

Die *Computation Tree Logic CTL** [8, 9, 21] ist eine nichtlineare[2] temporale Logik mit diskreter Zeit (wobei Zeit nicht explizit benutzt wird). CTL*-Formeln beschreiben Eigenschaften von Berechnungsbäumen, sogenannten *Computation Trees*. Diese ermöglichen eine gute Veranschaulichung aller von einem Startzustand ausgehenden Berechnungen. Für den Zustand s_0 der KRIPKE-Struktur aus Abbildung 3.1 wird dies in Abbildung 3.2 gezeigt.

[2] Man spricht auch von verzweigender Zeit (*branching time*).

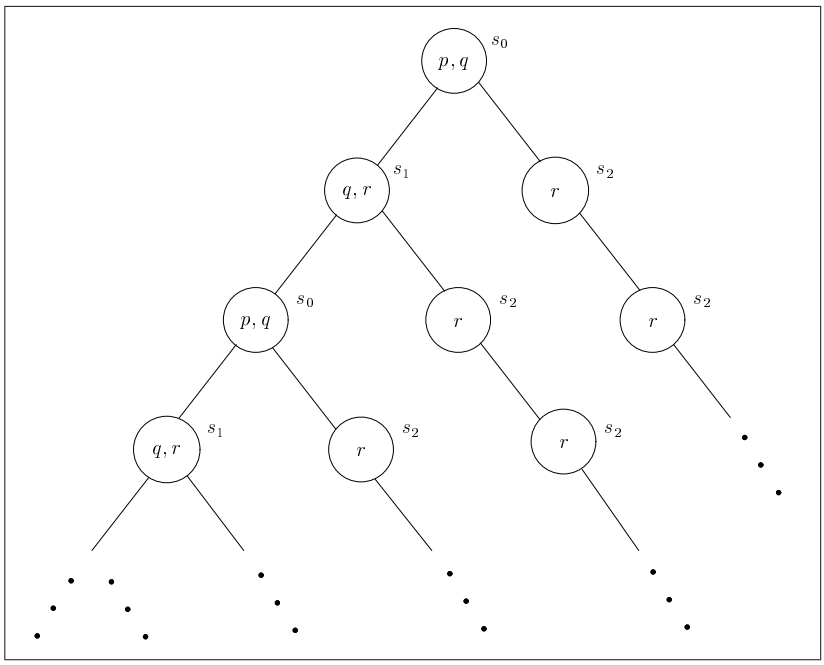

Abbildung 3.2: Berechnungsbaum in einer KRIPKE-Struktur [21]

In CTL* gibt es neben den bekannten aussagenlogischen Operatoren zwei *Pfadquantoren* und vier *temporale Operatoren*. Die Pfadquantoren **E** („es gibt einen Pfad“) und **A** („für alle Pfade“) beschreiben, auf welche Pfade sich die folgenden Teilformeln beziehen. Die temporalen Operatoren **X** („next state“), **F** („future state“), **G** („globally“) und **U** („until“) drücken lineare Pfadeigenschaften aus.

Definition 3.2 (Syntax von CTL*)
In CTL gibt es zwei Arten von Formeln:* ***Zustandsformeln*** *und* ***Pfadformeln****. Sei $\mathcal{AP}$ eine nichtleere Menge atomarer Aussagen.*

1. *Die Syntax der Zustandsformeln ist wie folgt definiert:*
   - *Wenn $\varphi \in \mathcal{AP}$, dann ist φ eine Zustandsformel.*
   - *Wenn φ und ψ Zustandsformeln sind, dann auch $\neg\varphi$ und $\varphi \vee \psi$.*
   - *Wenn φ eine Pfadformel ist, dann ist $\mathbf{E}\,\varphi$ eine Zustandsformel.*
2. *Die Syntax der Pfadformeln ist wie folgt definiert:*
   - *Wenn φ eine Zustandsformel ist, dann ist φ auch eine Pfadformel.*
   - *Wenn φ und ψ Pfadformeln sind, dann auch $\neg\varphi$, $\varphi \vee \psi$, $\mathbf{X}\,\varphi$ und $\varphi\,\mathbf{U}\,\psi$.*
3. *Zusätzlich werden die nachstehenden abkürzenden Notationen vereinbart:*
   - $\top \stackrel{def}{=} \varphi \vee \neg\varphi$ *für ein* $\varphi \in \mathcal{AP}$
   - $\varphi \wedge \psi \stackrel{def}{=} \neg(\neg\varphi \vee \neg\psi)$

- $\mathbf{A}\,\varphi \stackrel{def}{=} \neg\mathbf{E}\neg\varphi$
- $\mathbf{F}\,\varphi \stackrel{def}{=} \top\,\mathbf{U}\,\varphi$
- $\mathbf{G}\,\varphi \stackrel{def}{=} \neg\mathbf{F}\neg\varphi$

4. *CTL* ist genau die Menge der Zustandsformeln über $\mathcal{AP}$.*

Definition 3.3 (Pfad, Suffix)

1. *Eine unendliche Folge $\pi = s_0, s_1, \ldots$ von Zuständen aus S wird genau dann* ***Pfad*** *in der* KRIPKE*-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$ genannt, wenn $s_i \rightarrow s_{i+1}$ für alle $i \geq 0$ gilt.*

2. *Der Pfad $\pi^i \stackrel{def}{=} s_i, s_{i+1}, \ldots$ wird* ***Suffix*** *von π genannt.*

Im folgenden soll eine Semantik für CTL* angegeben werden. Vom induktiven Aufbau der Zustands- und Pfadformeln ausgehend, wird – beginnend mit den atomaren Aussagen – die Gültigkeit einer Formel in einem Zustand einer KRIPKE-Struktur festgelegt.

Definition 3.4 (Semantik von CTL*)
Sei $\mathcal{M} = (S, S_0, \rightarrow, L)$ eine KRIPKE*-Struktur. Die Gültigkeit einer Formel φ in einem Zustand $s \in S$ bzw. auf einem Pfad π in $\mathcal{M}$ wird notiert als*

$$\mathcal{M}, s \vDash \varphi \quad \textit{bzw.} \quad \mathcal{M}, \pi \vDash \varphi$$

Die ***Erfüllbarkeitsrelation*** $\vDash$ *ist induktiv über allen Formeln von CTL* wie folgt definiert*[3]*:*

1. $\mathcal{M}, s \vDash p \stackrel{def}{\Longleftrightarrow} p \in L(s)$ *für* $p \in \mathcal{AP}$
2. $\mathcal{M}, s \vDash \neg\varphi \stackrel{def}{\Longleftrightarrow} \mathcal{M}, s \nvDash \varphi$
3. $\mathcal{M}, s \vDash \varphi \vee \psi \stackrel{def}{\Longleftrightarrow} \mathcal{M}, s \vDash \varphi$ *oder* $\mathcal{M}, s \vDash \psi$
4. $\mathcal{M}, s \vDash \mathbf{E}\,\varphi \stackrel{def}{\Longleftrightarrow}$ *Es gibt einen Pfad* $\pi = s, s', \ldots$, *so daß* $\mathcal{M}, \pi \vDash \varphi$.
5. $\mathcal{M}, \pi \vDash \varphi \stackrel{def}{\Longleftrightarrow} \pi = s, s', \ldots$ *und* $\mathcal{M}, s \vDash \varphi$
6. $\mathcal{M}, \pi \vDash \neg\varphi \stackrel{def}{\Longleftrightarrow} \mathcal{M}, \pi \nvDash \varphi$
7. $\mathcal{M}, \pi \vDash \varphi \vee \psi \stackrel{def}{\Longleftrightarrow} \mathcal{M}, \pi \vDash \varphi$ *oder* $\mathcal{M}, \pi \vDash \psi$
8. $\mathcal{M}, \pi \vDash \mathbf{X}\,\varphi \stackrel{def}{\Longleftrightarrow} \mathcal{M}, \pi^1 \vDash \varphi$
9. $\mathcal{M}, \pi \vDash \varphi\,\mathbf{U}\,\psi \stackrel{def}{\Longleftrightarrow}$ *Es gibt ein* $k \geq 0$, *so daß* $\mathcal{M}, \pi^k \vDash \psi$ *und* $\mathcal{M}, \pi^i \vDash \varphi$ *für alle* $0 \leq i < k$.

Eine Formel φ ist genau dann wahr in einer KRIPKE*-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$, notiert $\mathcal{M} \vDash \varphi$, wenn $\mathcal{M}, s \vDash \varphi$ für alle $s \in S_0$.*

[3] φ und ψ sollen – jeweils passend – für Zustands- oder Pfadformeln stehen.

Manchmal werden wir $\mathcal{M}$ weglassen, wenn klar ist, daß wir uns auf die Gültigkeit einer Formel in dieser KRIPKE-Struktur beziehen.

Definition 3.5 (Semantische Äquivalenz)
*Zwei Formeln φ und ψ werden genau dann **semantisch äquivalent** genannt, notiert $\varphi \equiv \psi$, wenn für jede KRIPKE-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$ und jeden Zustand $s \in S$ gilt:*

$$\mathcal{M}, s \vDash \varphi \quad \Longleftrightarrow \quad \mathcal{M}, s \vDash \psi$$

Die temporalen Logiken CTL und LTL

Zwei für Model Checking interessante Teilmengen von CTL* sind CTL und LTL. Die *Computation Tree Logic CTL* [8, 9, 21] ist eine nichtlineare Logik, in deren Formeln die temporalen Operatoren **X**, **F**, **G** und **U** nur direkt nach einem der Pfadquantoren **E** und **A** vorkommen.

Definition 3.6 (Syntax von CTL)
Eine CTL-Formel ist genau dann eine CTL-Formel, wenn die in ihr enthaltenen Pfadformeln wie folgt aufgebaut sind:*

- *Wenn φ und ψ Zustandsformeln sind, dann sind $\mathbf{X}\,\varphi$ und $\varphi\,\mathbf{U}\,\psi$ Pfadformeln.*
- *Wenn φ eine Pfadformel ist, dann auch $\neg\varphi$.*

Die *Linear Temporal Logic LTL* [8, 9, 21] ist eine lineare Logik, das bedeutet, daß ihre Pfadformeln nur Eigenschaften einzelner Pfade beschreiben können.

Definition 3.7 (Syntax von LTL)
Sei $\mathcal{AP}$ eine Menge atomarer Aussagen. LTL ist die Menge der folgend beschriebenen Zustandsformeln.

1. *Wenn $\varphi \in \mathcal{AP}$, dann ist φ eine Pfadformel.*
2. *Wenn φ und ψ Pfadformeln sind, dann auch $\neg\varphi$, $\varphi \vee \psi$, $\mathbf{X}\,\varphi$ und $\varphi\,\mathbf{U}\,\psi$.*
3. *Wenn φ eine Pfadformel ist, dann ist $\mathbf{A}\,\varphi$ eine Zustandsformel.*

Die größte Teilformel φ einer LTL-Formel $\mathbf{A}\,\varphi$ wird *eingeschränkte Pfadformel* [9] genannt, da sie keine Zustandsformeln als Teilformeln enthält.

Definition 3.8 (eingeschränkte Pfadformel)
*Sei $\mathbf{A}\,\varphi$ eine LTL-Formel. Dann wird φ **eingeschränkte Pfadformel** genannt.*

CTL*, CTL und LTL sind unterschiedlich ausdrucksstark. Dabei werden LTL und CTL von CTL* majorisiert. So gibt es zur CTL-Formel $\mathbf{AG}(\mathbf{EF}\,p)$ keine äquivalente LTL-Formel, die LTL-Formel $\mathbf{A}(\mathbf{FG}\,p)$ kann nicht in CTL ausgedrückt werden. Zur CTL*-Formel $\mathbf{AG}(\mathbf{EF}\,p) \vee \mathbf{A}(\mathbf{FG}\,p)$ existieren weder in CTL noch in LTL äquivalente Formeln.

3.3.2 Sicherheits- und Lebendigkeitseigenschaften

Eigenschaften reaktiver Systeme werden in zwei Klassen zusammengefaßt: *Sicherheitseigenschaften* und *Lebendigkeitseigenschaften*. Sicherheitseigenschaften sagen aus, daß „schlechte Ereignisse niemals eintreten“, Lebendigkeitseigenschaften daß „ein gutes Ereignis irgendwann eintritt“ [29]. STIRLING [29] definiert dazu die folgenden allgemeinen Spezifikationsmuster für CTL. In LTL sind nur starke Sicherheits- und Lebendigkeitseigenschaften formulierbar.

Definition 3.9

1. *Sei φ ein unerwünschtes Ereignis. Dann unterscheidet man die Sicherheitseigenschaften*

   - $\mathbf{Safety}(\varphi) \stackrel{def}{=} \mathbf{AG}\,\neg\varphi$ *(**Starke Sicherheit**)*
   - $\mathbf{WSafety}(\varphi) \stackrel{def}{=} \mathbf{EG}\,\neg\varphi$ *(**Schwache Sicherheit**)*

2. *Sei φ ein gewünschtes Ereignis. Dann unterscheidet man die Lebendigkeitseigenschaften*

   - $\mathbf{Liveness}(\varphi) \stackrel{def}{=} \mathbf{AF}\,\varphi$ *(**Starke Lebendigkeit**)*
   - $\mathbf{WLiveness}(\varphi) \stackrel{def}{=} \mathbf{EF}\,\varphi$ *(**Schwache Lebendigkeit**)*

Bemerkung 3.10
Zu jeder Sicherheitseigenschaft $\mathbf{A}\,\varphi$ gibt es eine duale Lebendigkeitseigenschaft $\mathbf{E}\,\psi$, so daß $\varphi \equiv \neg\psi$.

Folgend werden einige typische Spezifikationsmuster für CTL und LTL angegeben.

Beispiel

1. $\mathbf{AG}\,\neg$Fehler: In keinem Zustand tritt der Fehler auf.

2. $\mathbf{AG}$(Anfrage $\rightarrow \mathbf{AF}$ Antwort): In jedem Zustand folgt auf eine Anfrage irgendwann eine Antwort.

3. $\mathbf{AG}(\mathbf{EF}$ reset): In jedem Zustand ist es möglich, den Zustand reset zu erreichen.

4. $\mathbf{AG}(\mathbf{AF}$ aktiviert): Ein Prozeß wird auf jedem Berechnungspfad unendlich häufig aktiviert.

5. $\mathbf{EF}$(gestartet $\wedge\ \neg$bereit): Es ist möglich, in einen Zustand zu gelangen, indem gestartet gilt, aber bereit nicht gilt.

6. $\mathbf{AF}(\mathbf{AG}$ Verklemmung): Ein Prozeß endet auf jedem Berechnungspfad irgendwann im Zustand Verklemmung.

Alle diese Formeln gehören zu CTL, 1., 4. und 6. auch zu LTL. Man sieht, daß 1. bis 4. Sicherheitseigenschaften sowie 5. und 6. Lebendigkeitseigenschaften sind. Bei 3. handelt es sich um eine sogenannte *Fairneßeigenschaft*[4].

3.4 LTL Model Checking

3.4.1 Das Model-Checking-Problem

Ziel von Verifikation ist es, die Gültigkeit einer Spezifikation in einem Modell zu bestimmen. Bei Model Checking überprüft man dazu temporallogische Eigenschaften von KRIPKE-Strukturen. Das *Model-Checking-Problem* kann wie folgt formalisiert werden [9]:

Definition 3.11
Das ***Model-Checking-Problem*** *ist wie folgt definiert:*
gegeben: *eine Kripke-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$, die ein reaktives System repräsentiert und eine Systemspezifikation als Formel φ einer temporalen Logik*
Problem: *Gilt $\mathcal{M} \vDash \varphi$?*

Die ersten Algorithmen zur Lösung des Model-Checking-Problems erforderten eine explizite Darstellung der zugrundeliegenden KRIPKE-Struktur. Bei einer Spezifikation mit n BOOLEschen Variablen ergibt dies bis zu 2^n Zustände. Das Hinzufügen einer neuen BOOLEschen Variablen zum Modell führt dann zu einer Verdoppelung des Zustandsraumes.

Dieser als *Zustandsexplosionsproblem* bekannten Tatsache kann mit verschiedenen Methoden entgegengewirkt werden (die exponentielle Komplexität für den worst case bleibt aber bestehen). Im folgenden sollen einige Möglichkeiten der Effizienzverbesserung vorgestellt werden. Die meisten Ansätze sind gut kombinierbar, da ihre Wirkungen orthogonal zueinander verlaufen.

Symbolisches Model Checking Binäre Entscheidungsbäume [6] in verschiedenen Varianten sind sehr effiziente Datenstrukturen. Durch die deutlich verringerte Zustandsgröße gegenüber expliziter Repräsentation wird eine erhebliche Platz- und Zeitersparnis erreicht. Die für Model Checking notwendigen Operationen sind sehr effizient implementierbar. Ausführliche Darstellungen finden sich unter anderem in [25, 9, 21].

Abstraktion Abstraktionsmethoden, die eine Verwendung kleinerer, abstrahierter Modelle ermöglichen, werden in 4.2 gezeigt.

Halbordnungsreduktion Halbordnungsreduktion ist die wichtigste Reduktionsmethode für asynchrone Systeme und wird in 4.1 dargestellt.

Symmetriereduktion Für Symmetrien in gleichförmig aufgebauten Systemen gibt es ein spezielles Reduktionsverfahren, siehe Kapitel 14 in [9].

[4] Da Fairneßeigenschaften in JPF keine Rolle spielen, wird hier auf weitere Ausführungen dazu verzichtet. Man findet aber ausführliche Darstellungen in der Literatur [9, 21].

Induktion Parametrisierte Strukturen eines Systems können durch Induktion ausgenutzt werden. Dies ist auch nützlich bei der Verifikation ganzer Klassen beliebig großer endlicher Systeme, siehe Kapitel 15 in [9].

Komposition Durch Ausnutzung von Invarianten kann man die Verifikation eines Systems manchmal auf die kleinerer Teilsysteme reduzieren, siehe Kapitel 12 in [9].

Heute wird fast nur noch symbolisches Model Checking angewendet. Explizite Zustandskodierung ist nur in seltenen Fällen notwendig. Wir werden in Kapitel 5 untersuchen, wie sich die explizite Zustandsdarstellung bei JPF auswirkt.

Die wichtigste Reduktionsmethode bei der Modellierung reaktiver Systeme ist Abstraktion. Halbordnungsreduktion ist eine der wichtigen Reduktionsmethoden für nebenläufige Systeme und ebenfalls sehr wirksam. Über Induktion und Komposition ist bisher eher wenig bekannt [10]. Beim Model Checking von JAVA-Programmen sind auch aus der Softwaretechnologie bekannte Techniken wie Slicing und Laufzeitmethoden anwendbar. Im nächsten Kapitel werden diese Methoden vertiefend diskutiert.

3.4.2 Ein Model-Checking-Algorithmus

Das System JAVA PATHFINDER ermöglicht Model Checking für Spezifikationen, die in LTL formuliert sind. Daher gilt der folgende Satz.

Satz 3.12 (Entscheidbarkeit)
Das Model-Checking-Problem für LTL ist entscheidbar.

Ziel der nachfolgenden theoretischen Betrachtungen ist eine konstruktive Lösung des Model-Checking-Problems für LTL. Dazu wird der Algorithmus von LICHTENSTEIN und PNUELI [24], der mit expliziter Zustandsnumerierung arbeitet, skizziert. Die weitere Darstellung folgt im wesentlichen CLARKE [9]. Ein Algorithmus für symbolisches Model Checking mit LTL, der auf anderen Ideen basiert, ist ebenfalls in [9] beschrieben.

Seien $\mathcal{M} = (S, S_0, \rightarrow, L)$ eine KRIPKE-Struktur und $\mathbf{A}\,\varphi$ eine Spezifikationsformel in LTL. Um $\mathcal{M}, s \vDash \mathbf{A}\,\varphi$ zu entscheiden, genügt es, dies für $\mathcal{M}, s \vDash \mathbf{E}\,\neg\varphi$ zu tun (es gilt $\mathcal{M}, s \vDash \mathbf{A}\,\varphi \Leftrightarrow \mathcal{M}, s \nvDash \mathbf{E}\,\neg\varphi$).

Die *Hülle* von φ – φ^h – soll alle Formeln enthalten, die zur Überprüfung der Gültigkeit von φ benötigt werden.

Definition 3.13 (Hülle)

1. *Die **Hülle** φ^h einer LTL-Formel φ ist die kleinste Menge, die φ enthält, so daß für alle $\varphi_1, \varphi_2 \in \varphi^h$ gilt:*

   - $\neg\varphi_1 \in \varphi^h \Leftrightarrow \varphi_1 \in \varphi^h$
   - $\varphi_1 \vee \varphi_2 \in \varphi^h \Rightarrow \varphi_1, \varphi_2 \in \varphi^h$
   - $\mathbf{X}\,\varphi_1 \in \varphi^h \Rightarrow \varphi_1 \in \varphi^h$
   - $\neg\mathbf{X}\,\varphi_1 \in \varphi^h \Rightarrow \mathbf{X}\,\neg\varphi_1 \in \varphi^h$
   - $\varphi_1\,\mathbf{U}\,\varphi_2 \in \varphi^h \Rightarrow \varphi_1, \varphi_2, \mathbf{X}(\varphi_1\,\mathbf{U}\,\varphi_2) \in \varphi^h$

2. *Für jede Formel φ_1 gilt: $\neg\neg\varphi_1$ wird mit φ_1 identifiziert.*

Ein *Atom* (s, K) soll jedem Zustand $s \in S$ eine maximale konsistente Menge von Formeln zuordnen, die auch mit der Beschriftung $L(s)$ konsistent ist.

Definition 3.14 (Atom)
Seien $\mathcal{M}$ eine KRIPKE*-Struktur, φ eine LTL-Formel und φ^h deren Hülle.*

1. *Ein **Atom** ist ein Paar $A = (s_A, K_A)$ mit $s_A \in S$ und $K_A \subseteq \varphi^h \cup \mathcal{AP}$, so daß gilt:*
   - $\forall \varphi_1 \in \mathcal{AP} : \varphi_1 \in K_A \Leftrightarrow \varphi_1 \in L(s_A)$.
   - $\forall \varphi_1 \in \varphi^h : \varphi_1 \in K_A \Leftrightarrow \neg\varphi_1 \notin K_A$.
   - $\forall \varphi_1 \vee \varphi_2 : \varphi_1 \vee \varphi_2 \in K_A \Leftrightarrow \varphi_1 \in K_A$ *oder* $\varphi_2 \in K_A$.
   - $\forall \neg\mathbf{X}\,\varphi_1 \in \varphi^h : \neg\mathbf{X}\,\varphi_1 \in K_A \Leftrightarrow \mathbf{X}\,\neg\varphi_1 \in K_A$.
   - $\forall \varphi_1\,\mathbf{U}\,\varphi_2 : \varphi_1\,\mathbf{U}\,\varphi_2 \in K_A \Leftrightarrow \varphi_2 \in K_A \vee \varphi_1, \mathbf{X}(\varphi_1\,\mathbf{U}\,\varphi_2) \in K_A$.
2. *Die Menge der Atome bezüglich $\mathcal{M}$ und φ wird mit $\mathcal{A}(\mathcal{M}, \varphi)$ bezeichnet.*

Das Kernstück des Algorithmus ist die Konstruktion eines *Tableaus*. Darunter versteht man einen Graphen über den Zuständen von $\mathcal{M}$ und den Atomen bezüglich φ und $\mathcal{M}$, dessen Kantenmenge durch eine Einschränkung der Kantenmenge von $\mathcal{M}$ entsteht. Wir werden sehen, daß eine Formel genau dann wahr in $\mathcal{M}$ ist, wenn sie im Tableau wahr ist.

Definition 3.15 (Tableau)
Seien $\mathcal{M} = (S, S_0, \rightarrow, L)$ eine KRIPKE*-Struktur, φ eine LTL-Formel und $\mathcal{A} = \mathcal{A}(\mathcal{M}, \varphi)$. Ein Graph $G(\mathcal{M}, \varphi) = (\mathcal{A}, R)$ wird genau dann **Tableau** genannt, wenn gilt:*

$$(A, B) \in R \quad \overset{def}{\Longleftrightarrow} \quad (s_A \rightarrow s_B) \wedge \forall\,\mathbf{X}\,\varphi_1 \in \varphi^h : (\mathbf{X}\,\varphi_1 \in K_A \Leftrightarrow \varphi_1 \in K_B)$$

Definition 3.16 (Lebendigkeitsfolge)
*Ein unendlicher Pfad π im Tableau $G(\mathcal{M}, \varphi)$ wird genau dann **Lebendigkeitsfolge**[5] genannt, wenn es für jedes Atom A von π und jede Formel $\varphi_1\,\mathbf{U}\,\varphi_2 \in K_A$ ein Atom B von π mit $\varphi_2 \in K_B$ gibt, welches von A aus über π erreichbar[6] ist.*

Lemma 3.17
$\mathcal{M}, s \vDash \mathbf{E}\,\varphi$ ist genau dann wahr, wenn es eine Lebendigkeitsfolge, beginnend mit (s, K), gibt, so daß $\varphi \in K$.

Beweis „$\Leftarrow$“: Sei $(s_0, K_0), (s_1, K_1), \ldots$ eine Lebendigkeitsfolge, beginnend mit $(s, K) = (s_0, K_0)$, mit $\varphi \in K$. Nach Definition ist $\pi = s_0, s_1, \ldots$ ein Pfad in $\mathcal{M}$. Es soll gezeigt werden, daß $\pi \vDash \varphi$ gilt. Dafür wird zunächst eine stärkere Behauptung bewiesen: für jede Formel $\varphi_1 \in \varphi^h$ und jedes $i \geq 0$ gilt: $\pi^i \vDash \varphi_1 \Leftrightarrow \varphi_1 \in K_i$. Der Beweis wird durch strukturelle Induktion über den Aufbau der LTL-Formel φ_1 geführt.

[5] Lebendigkeitsfolge wurde für den englischen Begriff *eventuality sequence* gewählt.
[6] Das schließt den Fall $A = B$ ein.

1. Wenn $\varphi_1 \in \mathcal{AP}$, dann ist nach Def. 3.14 $\varphi_1 \in K_i \Leftrightarrow \varphi_1 \in L(s_i)$, also $\varphi_1 \in K_i \Leftrightarrow \pi^i \vDash \varphi_1$.

2. Wenn $\varphi_1 = \neg\varphi_2$, dann ist $\pi^i \vDash \varphi_1 \Leftrightarrow \pi^i \nvDash \varphi_2$. Nach Induktionsvoraussetzung ist $\pi^i \vDash \varphi_1 \Leftrightarrow \pi^i \nvDash \varphi_2$ genau dann wahr, wenn $\varphi_2 \notin K_i$. Nach Def. 3.14 ist genau dann $\varphi_1 \in K_i$.

3. Wenn $\varphi_1 = \varphi_2 \vee \varphi_3$, dann ist $\pi^i \vDash \varphi_1 \Leftrightarrow \pi^i \vDash \varphi_2 \vee \pi^i \vDash \varphi_3$. Nach Induktionsvoraussetzung ist das genau dann wahr, wenn $\varphi_2 \in K_i$ oder $\varphi_3 \in K_i$. Nach Def. 3.14 ist genau dann $\varphi_1 \in K_i$.

4. Wenn $\varphi_1 = \mathbf{X}\,\varphi_2$, dann ist $\pi^i \vDash \varphi_1 \Leftrightarrow \pi^{i+1} \vDash \varphi_2$. Nach Induktionsvoraussetzung ist das genau dann wahr, wenn $\varphi_2 \in K_{i+1}$. Da nach Def. 3.15 $((s_i, K_i), (s_{i+1}, K_{i+1})) \in R$ gilt, ist das genau dann wahr, wenn $\varphi_1 \in K_i$.

5. Sei $\varphi_1 = \varphi_2\,\mathbf{U}\,\varphi_3$.

   „$\Rightarrow$“: Wenn $\varphi_1 \in K_i$, dann existiert nach Def. 3.16 ein $j \geq i$ mit $\varphi_3 \in K_j$. Wähle das kleinste solche j. Nach Def. 3.14 ist dann $\varphi_3 \in K_i$ oder $\varphi_2, \mathbf{X}\,\varphi_1 \in K_{i+1}$. Induktiv folgt $\varphi_2, \mathbf{X}\,\varphi_1 \in K_k$ für alle $i \leq k < j$. Nach Induktionsvoraussetzung ist dann $\pi^j \vDash \varphi_2$ und $\pi^k \vDash \varphi_3$ für alle $i \leq k < j$. Dann ist $\pi^i \vDash \varphi_1$.

   „$\Leftarrow$“: Wenn $\pi^i \vDash \varphi_1$, dann gibt es ein $j \geq i$ mit $\pi^j \vDash \varphi_3$ und $\pi^k \vDash \varphi_2$ für alle $i \leq k < j$. Wähle das kleinste solche j. Nach Induktionsvoraussetzung ist dann $\varphi_3 \in K_j$ und $\varphi_2 \in K_k$ für alle $i \leq k < j$. Angenommen $\varphi_1 \notin K_i$. Da $\varphi_2 \in K_i$ ist, ist dann nach Def. 3.14 $\mathbf{X}\,\varphi_1 \notin K_i$ und damit $\mathbf{X}\,\neg\varphi_1 \in K_i$. Nach Def. 3.15 ist dann $\neg\varphi_1 \in K_{i+1}$ und schließlich nach Def. 3.14 $\varphi_1 \notin K_{i+1}$. Mit dem gleichen Argument folgt induktiv, daß $\varphi_1 \notin K_j$. Das ist ein Widerspruch zu $\varphi_1 \in K_j$. Also ist $\varphi_1 \in K_i$.

„$\Rightarrow$“: $s \vDash \mathbf{E}\,\varphi$ sei wahr. Dann gibt es einen Pfad $\pi = s_0, s_1, \ldots$ in $\mathcal{M}$, beginnend mit $s = s_0$, so daß $\pi \vDash \varphi$. Sei $K_i = \{\varphi \mid \varphi \in \varphi^h \cup \mathcal{AP} \wedge \pi^i \vDash \varphi\}$ für alle $i \geq 0$. Dann gilt:

1. Wir zeigen, daß (s_i, K_i) ein Atom ist. Es ist klar, daß $s_i \in S$ und $K_i \subseteq \varphi^h \cup \mathcal{AP}$. Nun muß noch gezeigt werden, daß (s_i, K_i) Def. 3.14 genügt. Dafür wird strukturelle Induktion über den Aufbau von LTL-Formeln φ_1 verwendet.

   (a) Sei $\varphi_1 \in \mathcal{AP}$. $\varphi_1 \in K_i$ gilt genau dann, wenn $\pi^i \vDash \varphi_1$ bzw. $s_i \vDash \varphi_1$ wahr ist. Das ist genau dann, wenn $\varphi_1 \in L(s_i)$.

   (b) Sei $\varphi_1 \in \varphi^h$. $\pi^i \vDash \varphi_1$ ist genau dann wahr, wenn $\pi^i \nvDash \neg\varphi_1$ wahr ist. Also ist genau dann $\varphi_1 \in K_i$, wenn $\neg\varphi_1 \notin K_i$.

   (c) Sei $\varphi_1 \vee \varphi_2 \in \varphi^h$. $\pi^i \vDash \varphi_1 \vee \varphi_2$ ist genau dann wahr, wenn $\pi^i \vDash \varphi_1$ oder $\pi^i \vDash \varphi_2$ wahr ist. Also ist genau dann $\varphi_1 \vee \varphi_2 \in K_i$, wenn $\varphi_1 \in K_i$ oder $\varphi_2 \in K_i$.

   (d) Sei $\neg\mathbf{X}\,\varphi_1 \in \varphi^h$. $\pi^i \vDash \neg\mathbf{X}\,\varphi_1$ ist genau dann wahr, wenn $\pi^{i+1} \nvDash \varphi_1$ wahr ist. Das ist genau dann, wenn $\pi^{i+1} \vDash \neg\varphi_1$ bzw. $\pi \vDash \mathbf{X}\,\neg\varphi_1$. Also ist genau dann $\neg\mathbf{X}\,\varphi_1 \in K_i$, wenn $\mathbf{X}\,\neg\varphi_1 \in K_i$.

(e) Sei $\varphi_1 \mathbf{U} \varphi_2 \in \varphi^h$. $\pi^i \vDash \varphi_1 \mathbf{U} \varphi_2$ ist genau dann wahr, wenn es ein $j \geq i$ gibt, so daß $\pi^j \vDash \varphi_2$ und $\pi^k \vDash \varphi_1$ für alle $i \leq k < j$ wahr ist. Das ist genau dann, wenn aus $\pi^i \vDash \varphi_1 \mathbf{U} \varphi_2$ immer $\pi^i \vDash \varphi_1$ oder $\pi^i \vDash \mathbf{X}(\varphi_1 \mathbf{U} \varphi_2)$ folgt. Also ist genau dann $\varphi_1 \mathbf{U} \varphi_2 \in K_i$, wenn $\varphi_1 \in K_i$ oder $\mathbf{X}(\varphi_1 \mathbf{U} \varphi_2) \in K_i$.

2. Wir zeigen, daß es für alle $i \geq 0$ Transitionen $((s_i, K_i), (s_{i+1}, K_{i+1})) \in R$ gibt. Nach Def. 3.15 ist genau dann $(A, B) \in R$, wenn $s_A \rightarrow s_B$ und für jede Formel $\mathbf{X} \varphi_1 \in \varphi^h$ gilt, daß $\mathbf{X} \varphi_1 \in K_A \Leftrightarrow \varphi_1 \in K_B$. Das ist genau dann, wenn $\pi^i \vDash \mathbf{X} \varphi_1 \Leftrightarrow \pi^{i+1} \vDash \varphi_1$. Aus der Definition von K_i folgt, daß $\mathbf{X} \varphi_1 \in K_i \Leftrightarrow \varphi_1 \in K_{i+1}$.

3. Wir zeigen, daß der Pfad $(s_0, K_0), (s_1, K_1), \ldots$ eine Lebendigkeitsfolge in $G(\mathcal{M}, \varphi)$ ist. Wenn $\varphi_1 \mathbf{U} \varphi_2 \in K_i$, dann gilt $\pi^i \vDash \varphi_1 \mathbf{U} \varphi_2$. Dann gibt es ein $j \geq i$, so daß $\pi^j \vDash \varphi_2$. Dann ist $\varphi_2 \in K_j$ und K_j ist von K_i aus erreichbar.

□

Definition 3.18 (stark zusammenhängende Komponente)

1. *Ein maximaler Teilgraph C eines Graphen G, in dem jeder Knoten aus C von jedem anderen Knoten aus C durch einen Pfad in C erreichbar ist, wird **stark zusammenhängende Komponente** von G genannt.*

2. *Eine stark zusammenhängende Komponente wird **nichttrivial** genannt, wenn sie mehr als einen Knoten oder einen Zyklus der Länge 1 besitzt.*

3. *Eine nichttriviale stark zusammenhängende Komponente C eines Tableaus wird **selbsterfüllend** genannt, wenn sie für jedes in ihr enthaltene Atom A und jede Formel $\varphi_1 \mathbf{U} \varphi_2 \in K_A$ ein Atom B enthält, so daß $\varphi_2 \in K_B$.*

Lemma 3.19
Es gibt genau dann eine Lebendigkeitsfolge in $G(\mathcal{M}, \varphi)$, beginnend mit (s, K), wenn es einen Pfad von (s, K) zu einer selbsterfüllenden stark zusammenhängenden Komponente gibt.

Beweis „⇒“: Es gebe eine Lebendigkeitsfolge, beginnend mit (s, K). Betrachte die Menge C' aller Atome, die in ihr unendlich oft vorkommen. Es ist klar, daß C' Teil einer stark zusammenhängenden Komponente C von $G(\mathcal{M}, \varphi)$ ist. Sei $\varphi = \varphi_1 \mathbf{U} \varphi_2 \in \varphi^h$ und $(s, K) \in C$ ein Atom mit $\varphi \in K$. Da C stark zusammenhängend ist, gibt es einen (endlichen) Pfad von (s, K) nach C' in C, also auch in $G(\mathcal{M}, \varphi)$. Wenn φ_2 auf diesem Pfad vorkommt, muß es ein Atom in C geben, das φ_2 enthält. Ansonsten muß φ in jedem Atom auf diesem Pfad enthalten sein, insbesondere auch in einem Atom von C'. Da C' durch eine Lebendigkeitsfolge charakterisiert ist, muß φ_2 in einem Atom von C', also auch von C, vorkommen. Damit ist C selbsterfüllend.

„⇐“: Es gebe einen Pfad in $G(\mathcal{M}, \varphi)$ von einem Atom (s, K) zu einer selbsterfüllenden stark zusammenhängenden Komponente C. Dann läßt sich eine Folge von Atomen in C so konstruieren, daß auf jedes Vorkommen der

Formel $\varphi_1 \mathbf{U} \varphi_2 \in \varphi^h$ eines der Formel φ_2 folgt. Betrachte nun Vorkommen von Formeln $\varphi = \varphi_1 \mathbf{U} \varphi_2 \in \varphi^h$ auf dem Pfad von (s, K) nach C. Auf jedes Vorkommen von φ muß nach Def. 3.14, 3.15 entweder ein Vorkommen von φ_2 folgen, oder φ muß in jedem Atom dieses Pfades enthalten sein. Da C selbsterfüllend und stark zusammenhängend ist, gibt es auch in letzterem Fall ein Vorkommen von φ_2 und dieses ist erreichbar. □

Satz 3.20 (Korollar)
$\mathcal{M}, s \vDash \mathbf{E} \varphi$ ist genau dann wahr, wenn es ein Atom $A = (s, K)$ in $G(\mathcal{M}, \varphi)$ gibt, so daß $\varphi \in K$ und ein Pfad in $G(\mathcal{M}, \varphi)$ von A zu einer selbsterfüllenden stark zusammenhängenden Komponente existiert.

Satz 3.20 ermöglicht es nun, die die grundsätzliche Struktur eines Algorithmus zur Lösung des Model-Checking-Problems für LTL zu skizzieren. Diese ist in Abbildung 3.3 dargestellt.

Eingabe: eine KRIPKE-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$, eine LTL-Formel $\mathbf{A} \varphi$
Problem: Gilt $\mathcal{M} \vDash \mathbf{A} \varphi$?

1. Berechne die Hülle $(\neg\varphi)^h$.
2. Berechne die Menge aller Atome $\mathcal{A} := \mathcal{A}(\mathcal{M}, \neg\varphi)$.
3. Konstruiere das Tableau $G(\mathcal{M}, \neg\varphi) = (\mathcal{A}, R)$.
4. Partitioniere $G(\mathcal{M}, \neg\varphi)$ in stark zusammenhängende Komponenten.
5. Betrachte alle Atome $A = (s, K)$ mit $s \in S_0$ und $\neg\varphi \in K$: Falls es einen Pfad von A zu einer selbsterfüllenden stark zusammenhängenden Komponente gibt, dann terminiere mit „nein“.
6. Terminiere mit „ja“.

Abbildung 3.3: Skizze eines Model-Checking-Algorithmus für LTL

Satz 3.21 (Totale Korrektheit)
1. *Wenn der Algorithmus terminiert, ist die Ausgabe genau dann „ja“, wenn $\mathcal{M} \vDash \mathbf{A} \varphi$.*
2. *Der Algorithmus terminiert für alle Eingaben $\mathcal{M}$, $\mathbf{A} \varphi$.*

Beweis

1. Angenommen, der Algorithmus terminiert mit „nein“. Dann gibt es in $G(\mathcal{M}, \neg\varphi)$ ein Atom $A = (s, K)$ mit $s \in S_0$, so daß $\neg\varphi \in K$ und ein Pfad von A zu einer selbsterfüllenden stark zusammenhängenden Komponente existiert. Nach Satz 3.20 gilt dann $\mathcal{M}, s \vDash \mathbf{E} \neg\varphi$. Das gilt genau dann, wenn $\mathcal{M}, s \nvDash \mathbf{A} \varphi$, also $\mathcal{M} \nvDash \mathbf{A} \varphi$. Angenommen, der Algorithmus terminiert mit „ja“. Mit dem gleichen Argument gilt dann $\mathcal{M} \vDash \mathbf{A} \varphi$.

2. Wir betrachten die einzelnen Schritte des Algorithmus. Da $\neg\varphi$ endlich ist, ist $(\neg\varphi)^h$ berechenbar. Es ist klar, daß dann auch die Menge $\mathcal{A}(\mathcal{M}, \neg\varphi)$ aller Atome berechenbar ist (im naiven Ansatz werden alle Elemente von $S \times ((\neg\varphi)^h \cup \mathcal{AP})$ auf die Bedingungen von Def. 3.14 untersucht). Deshalb terminiert auch die Tableaukonstruktion. Wir nehmen an, daß der Partitionierungsalgorithmus terminiert (das gilt zum Beispiel für den Algorithmus von TARJAN). Das Erreichbarkeitsproblem für den Graphen $G(\mathcal{M}, \neg\varphi)$ ist ebenfalls algorithmisch entscheidbar. □

Satz 3.22 (Korollar)
Der Algorithmus entscheidet das Model-Checking-Problem für LTL.

Bemerkung 3.23
Der Algorithmus hat eine Zeitkomplexität von $\mathcal{O}((|S| + |\rightarrow|) \cdot 2^{\mathcal{O}(|\varphi|)})$.

Begründung Bezeichne $|\varphi|$ die Länge der Formel φ. Dann hat $\neg\varphi$ höchstens $\mathcal{O}(|\varphi|)$ Teilformeln. Demnach gibt es höchstens $2^{\mathcal{O}(|\varphi|)}$ Mengen $K \subseteq \varphi^h$. Wenn man die in den Atomen enthaltenen atomaren Aussagen außer Acht läßt, gibt es somit höchstens $|S| \cdot 2^{|\varphi|}$ Atome. Die Konstruktion des Tableaus erfordert je Durchlauf des Graphen eine Zeit von $\mathcal{O}(|S| + |\rightarrow|)$. Für jedes Atom ist ein Durchlauf notwendig. Die Partitionierung in stark zusammenhängende Komponenten benötigt mit dem Algorithmus von TARJAN eine Zeit von $\mathcal{O}(|S| + |\rightarrow|)$. Die Erreichbarkeitsanalyse des Graphen $G(\mathcal{M}, \varphi)$ erfordert bezüglich $|\rightarrow|$ lineare Zeit.

Die folgenden Resultate liefern theoretische Grenzen für Model-Checking-Algorithmen.

Bemerkung 3.24
Das Model-Checking-Problem für CTL und LTL ist PSPACE-vollständig, das gilt sogar dann, wenn man nur eingeschränkte Pfadformeln betrachtet.*

Beispiel Gegeben sei das System „Mikrowellenofen“, das in Abbildung 3.4 dargestellt ist. Die Systemspezifikation $\mathbf{A}(\neg \mathit{Heat}\, \mathbf{U}\, \mathit{Close})$ besagt, daß der Ofen nie bei offener Tür warm wird. Wir überprüfen die Spezifikation mit dem Model-Checking-Algorithmus.

Dazu verwenden wir die duale Formel $\mathbf{E}\neg(\neg \mathit{Heat}\, \mathbf{U}\, \mathit{Close})$, also $\varphi = \neg \mathit{Heat}\, \mathbf{U}\, \mathit{Close}$. Um Schreibaufwand zu sparen, betrachten wir außerdem nur solche atomaren Aussagen, die Einfluß auf den Wahrheitswert der Formel haben, also $\mathcal{AP} = \{\mathit{Heat}, \mathit{Close}\}$.

1. $(\neg\varphi)^h = \{\neg\varphi, \varphi, \mathbf{X}\,\varphi, \neg\mathbf{X}\,\varphi, \mathbf{X}\,\neg\varphi, \mathit{Heat}, \neg\,\mathit{Heat}, \neg\,\mathit{Close}\}$ ist die Hülle von φ.

2. In den Zuständen 1 und 2 gelten die atomaren Aussagen $\neg\mathit{Heat}$ und $\neg\mathit{Close}$, sie müssen also in jedem Atom bezüglich 1 und 2 enthalten sein. Davon ausgehend ist frei wählbar, ob $\mathbf{X}\,\varphi$ in einem Atom enthalten ist. Daher ergeben sich die Atome $(1, K_1), (1, K_1'), (2, K_2)$ und $(2, K_2')$ mit

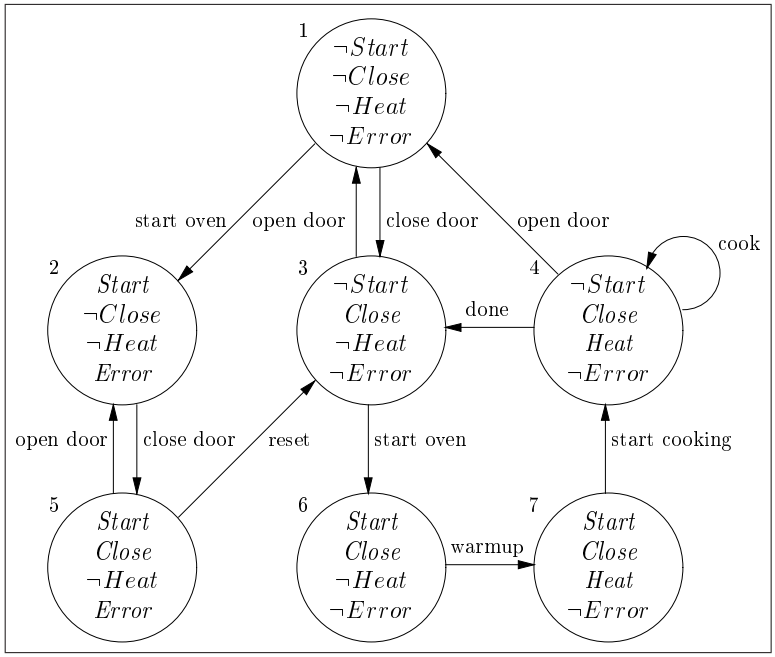

Abbildung 3.4: KRIPKE-Struktur für das Beispiel „Mikrowellenofen“ [9]

$K_1 = \{\neg Close, \neg Heat, \mathbf{X}\,\varphi, \varphi\}$ sowie $K_1' = \{\neg Close, \neg Heat, \neg\mathbf{X}\,\varphi, \mathbf{X}\,\neg\varphi, \neg\varphi\}$.

In den Zuständen 3, 5 und 6 gelten $\neg Heat$ und $Close$. Analog ergeben sich die Atome $(3, K_2), (3, K_2'), (5, K_2), (5, K_2'), (6, K_2)$ und $(6, K_2')$ mit $K_2 = \{Close, \neg Heat, \mathbf{X}\,\varphi, \varphi\}$ sowie $K_2' = \{Close, \neg Heat, \neg\mathbf{X}\,\varphi, \mathbf{X}\,\neg\varphi, \varphi\}$.

Für die Zustände 4 und 7 ergeben sich die Atome $(4, K_3), (4, K_3'), (7, K_3)$ und $(7, K_3')$ mit $K_3 = \{Close, Heat, \mathbf{X}\,\varphi, \varphi\}$ und $K_3' = \{Close, Heat, \neg\mathbf{X}\,\varphi, \mathbf{X}\,\neg\varphi, \varphi\}$.

3. Über der Menge aller Atome wird der Tableaugraph konstruiert. Dabei ist genau dann $((s, K), (s', K')) \in R$, wenn $s \to s'$ und für alle Formeln $\mathbf{X}\,\varphi_1 \in K$ auch $\varphi_1 \in K'$ ist. Die einzigen Formeln dieser Art sind $\mathbf{X}\,\varphi$ und $\mathbf{X}\,\neg\varphi$. Daher besitzt R genau die folgenden, durch Adjazenzlisten dargestellten Transitionen:

   - $(1, K_1) : [(2, K_1), (3, K_2), (3, K_2')]$
   - $(1, K_1') : [(2, K_1')]$
   - $(2, K_1) : [(5, K_2), (5, K_2')]$
   - $(3, K_2) : [(1, K_1), (6, K_2), (6, K_2')]$
   - $(3, K_2') : [(1, K_1')]$
   - $(4, K_3) : [(1, K_1), (3, K_2), (3, K_2'), (4, K_3), (4, K_3')]$
   - $(4, K_3') : [(1, K_1')]$
   - $(5, K_2) : [(2, K_1), (3, K_2), (3, K_2')]$
   - $(5, K_2') : [(2, K_1')]$

- $(6, K_2) : [(7, K_3), (7, K_3')]$
- $(7, K_3) : [(4, K_3), (4, K_3')]$

4. Daraus ergeben sich die stark zusammenhängenden Komponenten $C_1 = \{(1, K_1), (3, K_2), (6, K_2), (7, K_3), (4, K_3), (2, K_1), (5, K_2)\}$, $C_2 = \{(1, K_1')\}$, $C_3 = \{(2, K_1')\}$, $C_4 = \{(3, K_2')\}$, $C_5 = \{(4, K_3')\}$, $C_6 = \{(5, K_2')\}$, $C_7 = \{(6, K_2')\}$ und $C_8 = \{(7, K_3')\}$ mit den Transitionen $(C_1, C_4), (C_1, C_5)$, $(C_1, C_6), (C_1, C_7), (C_1, C_8), (C_2, C_3), (C_4, C_2), (C_5, C_2)$ sowie (C_6, C_3).

5. Es zeigt sich, daß nur C_1 und C_5 nichttrivial sind. Beide enthalten aber kein Atom A mit $\neg\varphi \in A$. Deshalb müssen auch sie nicht betrachtet werden.

6. Der Algorithmus terminiert mit „ja“, also gilt die Spezifikation $\mathbf{A}\,(\neg \mathit{Heat}\ \mathbf{U}\ \mathit{Close})$.

Kapitel 4

Effizienzverbesserungen

Insbesondere bei der Verwendung expliziter Zustandsnumerierung ist Model Checking sehr speicher- und rechenzeitintensiv. Dieses Kapitel soll alle wesentlichen in Java PathFinder vorkommenden Methoden zur Reduktion des Zustandsraumes erörtern.

Zu Beginn werden statische Methoden wie Slicing von Programmen, Halbordnungsreduktion und ein daraus speziell für JPF entwickeltes Fixpunktiterationsverfahren beschrieben. Im dritten Teil werden Abstraktionsmöglichkeiten, speziell Abstraktion durch Prädikate, diskutiert. Zum Schluß geht es um Algorithmen, die während der Laufzeit eines Programmes aktiv sind und ihren Einsatz unabhängig von oder in Kombination mit Model Checking.

4.1 Statische Reduktion

Statische Reduktionsverfahren basieren auf der Analyse von Quellkode. Dabei kommen Methoden zur Abhängigkeitsanalyse und symbolischen Auswertung von Programmanweisungen zum Einsatz.

4.1.1 Slicing

In den meisten Fällen beschreibt eine Spezifikationsformel nur einen kleinen Teil des gesamten Systems. Insbesondere wenn das System als Quellkode eines Programmes vorliegt, ist der relevante Programmabschnitt besonders einfach zu ermitteln. Die dazu verwendeten Methoden nennt man *Slicingmethoden* [31].

Definition 4.1 (Slicingkriterium)
Ein Paar (n, x), bestehend aus einer Zeilennummer n und einem Variablenbezeichner x, wird ***Slicingkriterium*** *genannt.*

Definition 4.2 (Programmslice)
Seien $C = (n, x)$ ein Slicingkriterium und P ein Programm. Dann wird die Menge aller Anweisungen von P, die Einfluß auf den Wert von x zum Abarbeitungszeitpunkt der Zeile n haben, ***Programmslice*** *von P bezüglich C genannt.*

Natürlich sind auch Verallgemeinerungen dieser Programmslices, zum Beispiel für den Fall, daß eine Spezifikation Aussagen über mehrere Slicingkriterien trifft, möglich. Ein Programmslice muß kein eigenständiges Programm sein. Man unterscheidet statische und dynamische Slices.

Definition 4.3
Ein Programmslice wird genau dann ***statisch*** *genannt, wenn er sich für alle Eingaben des Programmes bezüglich der Spezifikation wie das Programm verhält. Andernfalls wird er* ***dynamisch*** *genannt.*

Zum Abschluß wird ein Beispiel für statische und dynamische Slices angegeben.

Beispiel [31]
In Abbildung 4.1 ist ein Programm und sein statischer Slice bezüglich des Kriteriums $(10, \texttt{prod})$ zu sehen.

```
(1)   read(n);                 (1)   read(n);
(2)   i := 1;                  (2)   i := 1;
(3)   sum := 0;                (3)
(4)   prod := 1;               (4)   prod := 1;
(5)   while (i <= n) do        (5)   while (i <= n) do
      begin                          begin
(6)     sum := sum + i;        (6)
(7)     prod := prod * i;      (7)     prod := prod * i;
(8)     i := i + 1;            (8)     i := i + 1;
      end;                           end;
(9)   write(sum);              (9)
(10)  write(prod);             (10)  write(prod);
```

Abbildung 4.1: Beispiel für einen statischen Programmslice [31]

Abbildung 4.2 zeigt ein Programm und seinen dynamischen Slice bezüglich des Kriteriums $(8, \texttt{x})$ und der Eingabe $\texttt{n} = 2$.

```
(1)   read(n);                 (1)   read(n);
(2)   i := 1;                  (2)   i := 1;
(3)   while (i <= n) do        (3)   while (i <= n) do
      begin                          begin
(4)     if (i mod 2 = 0) then
(5)       x := 17;             (5)       x := 17;
        else                           else
(6)       x := 18;             (6)       ;
(7)     i := i + 1;            (8)     i := i + 1;
      end;                           end;
(8)   write(x);                (10)  write(x);
```

Abbildung 4.2: Beispiel für einen dynamischen Programmslice [31]

4.1.2 Halbordnungsreduktion

Bei der Modellierung nebenläufiger Systeme gibt es zu n parallel ausgeführten Aktionen $n!$ verschiedene Berechnungspfade und 2^n Zustände (siehe Abbildung

4.3). Häufig aber ist die Reihenfolge von Aktionen ohne Einfluß auf die Gültigkeit einer Spezifikation. In diesem Fall müßte nur ein Pfad stellvertretend für alle anderen betrachtet werden, der aus $n + 1$ Zuständen besteht.

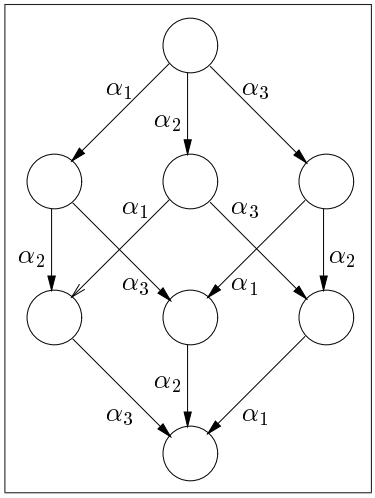

Abbildung 4.3: Modellierung parallel ausgeführter Aktionen [9]

Es wird gezeigt, daß Berechnungspfade, die sich bezüglich der betrachteten Spezifikation gleich verhalten, eine Äquivalenzrelation beschreiben. Um korrekte Resultate zu erhalten ist es ausreichend, den Model-Checking-Algorithmus auf das reduzierte Modell anzuwenden. Dieses Verfahren wird aus historischen Gründen *Halbordnungsreduktion* genannt. Der Darstellung von Clarke [9] folgend, werden nun die theoretischen Grundlagen dafür gelegt.

Definition 4.4 (Zustandstransitionssystem)
Sei $\mathcal{AP}$ eine Menge atomarer Aussagen. Ein ***Zustandstransitionssystem*** *$\widetilde{\mathcal{M}} = (S, S_0, \mathcal{T}, L)$ über $\mathcal{AP}$ ist wie folgt definiert:*

1. *S, S_0 und L entsprechen der Definition von* Kripke*-Systemen.*
2. *$\mathcal{T} \subseteq \mathfrak{P}(S \times S)$ ist die* ***Menge von Transitionsrelationen****.*

Bei der Beschreibung eines nebenläufigen Systems können die Begriffe Kripke-Struktur und Zustandstransitionssystem parallel benutzt werden. Wenn man Aussagen über verschiedene Berechnungspfade in einer Kripke-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$ treffen will, konstruiert man das dazugehörige Zustandstransitionssystem $\widetilde{\mathcal{M}} = (S, S_0, \mathcal{T}, L)$, so daß für alle $\alpha \in \mathcal{T}$ aus $\alpha(s, s')$ auch $s \rightarrow s'$ folgt und zu jedem Pfad von $\mathcal{M}$ ein äquivalenter Pfad in $\widetilde{\mathcal{M}}$ existiert. Um beispielsweise die Semantik eines Zustandstransitionssystems $\widetilde{\mathcal{M}} = (S, S_0, \mathcal{T}, L)$ zu beschreiben, erzeugt man die dazugehörige Kripke-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$, so daß $s \rightarrow s'$ genau dann gilt, wenn es ein $\alpha \in \mathcal{T}$ mit $\alpha(s, s')$ gibt.

Da wir uns auf die Untersuchung von Transitionsfolgen, also Mengen von Transitionen, konzentrieren wollen, soll der Begriff Transition auch für Elemente von $\mathcal{T}$ benutzt werden. Außerdem sollen zusammengehörige Kripke-Strukturen und Zustandstransitionssysteme durch den Begriff *Modelle* verallgemeinert werden.

Definition 4.5 (Transition)
Sei $\widetilde{\mathcal{M}} = (S, S_0, \mathcal{T}, L)$ ein Zustandstransitionssystem.

1. *Eine Transition $\alpha \in \mathcal{T}$ heißt genau dann **aktiviert** in einem Zustand $s \in S$, wenn es einen Zustand s' mit $\alpha(s, s')$ gibt. Andernfalls heißt sie **deaktiviert** in s.*

2. *Eine Transition α wird genau dann **deterministisch** genannt, wenn es für jeden Zustand $s \in S$ höchstens einen Zustand $s' \in S$ mit $\alpha(s, s')$ gibt. Man notiert dies als $s' = \alpha(s)$.*

Im folgenden werden nur deterministische Transitionen betrachtet.

Definition 4.6 (Pfad, Länge)
*Eine nicht notwendigerweise unendliche Folge $\pi = s_0, s_1, \ldots$ bzw. $\pi = s_0, \ldots, s_n$ von Zuständen aus S wird genau dann **Pfad** im Zustandstransitionssystem $\widetilde{\mathcal{M}} = (S, S_0, \mathcal{T}, L)$ genannt, wenn $\alpha_i(s_i, s_{i+1})$ für alle $i \geq 0$ bzw. für alle $0 \leq i < n$ gilt. Ein endlicher Pfad $s_0, \ldots, s_n$ hat die **Länge** n.*

Der in Abbildung 4.4 angegebene Algorithmus kann die Reduktion während der Generierung des Zustandsgraphen vornehmen, so daß niemals das komplette unreduzierte Modell gespeichert werden muß. Die verwendete Tiefensuche betrachtet jeweils nur solche Transitionen, die sich in einer repräsentativen Menge *ample*(s) von aktivierten Transitionen befinden. Damit die Reduktionen effektiv sind, muß die Menge *ample*(s) erheblich kleiner als *enabled*(s) und trotzdem relativ einfach zu berechnen sein. Sie muß aber groß genug sein, um den Wahrheitswert der Spezifikation zu erhalten. Eine exakte Definition für *ample* erfolgt später in Definition 4.13.

Eingabe: eine KRIPKE-Struktur $\mathcal{M} = (S, S_0, \rightarrow, L)$, eine Funktion $ample : S \rightarrow \mathfrak{P}(\mathcal{T})$

- Führe *expand_state*(s) für alle Zustände $s \in S_0$ aus.
- *expand_state*(s) ist wie folgt definiert:
  1. $work\_set(s) := ample(s)$;
  2. Wähle eine Transition $\alpha \in work\_set(s)$.
  3. $work\_set(s) := work\_set(s) \setminus \{\alpha\}$
  4. $s' := \alpha(s)$
  5. Wenn s' noch nicht als besucht markiert ist, dann markiere s' als besucht und führe *expand_state*(s') aus.
  6. Füge eine Kante $s \xrightarrow{\alpha} s'$ zum reduzierten Modell hinzu.
  7. Wenn $work\_set(s) = \emptyset$, dann terminiere. Andernfalls gehe zu Schritt 2.

Abbildung 4.4: Algorithmus zur Halbordnungsreduktion

Wir wollen Transitionen, die sich nicht gegenseitig deaktivieren können, und deren Reihenfolge beliebig ist, als voneinander *unabhängig* beschreiben.

Definition 4.7 (Abhängigkeit)
Sei $\widetilde{\mathcal{M}} = (S, S_0, \mathcal{T}, L)$ ein Zustandstransitionssystem.

1. *Die **Unabhängigkeitsrelation** $I \subseteq \mathcal{T} \times \mathcal{T}$ ist die größte symmetrische, irreflexive Relation, so daß für alle $s \in S$ und alle $(\alpha, \beta) \in I$ gilt:*
   - $\alpha, \beta \in enabled(s) \Rightarrow \alpha \in enabled(\beta(s))$
   - $\alpha, \beta \in enabled(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$
2. *$D = (\mathcal{T} \times \mathcal{T}) \setminus I$ wird **Abhängigkeitsrelation** genannt.*

Trotzdem funktioniert der naive Ansatz, alle voneinander unabhängigen Transitionen miteinander zu verschmelzen, nicht. Es besteht weiterhin die Möglichkeit, daß Zwischenzustände wie im hier $\alpha(s)$ und $\beta(s)$ für den Wahrheitswert der Spezifikation relevant sind, auch indirekt über ihre Folgezustände. Wir wollen diesen Aspekt formalisieren.

Definition 4.8 (Sichtbarkeit)
*Sei $\widetilde{\mathcal{M}} = (S, S_0, \mathcal{T}, L)$ ein Zustandstransitionssystem über $\mathcal{AP}$. Eine Transition $\alpha \in \mathcal{T}$ wird genau dann **unsichtbar** bezüglich einer Menge $\mathcal{AP}' \subseteq \mathcal{AP}$ genannt, wenn für alle Zustände $s, s' \in S$ mit $s' = \alpha(s)$ gilt:*

$$L(s) \cap \mathcal{AP}' = L(s') \cap \mathcal{AP}'$$

*Andernfalls wird sie **sichtbar** bezüglich $\mathcal{AP}'$ genannt.*

In nebenläufigen Systemen gibt es keine Korrelation zwischen der Anzahl von Transitionen, die zwischen zwei Aktionen stattfinden und der Zeit, die währenddessen vergeht. Deshalb ist hier ein spezielles Konzept der Äquivalenz zwischen Pfaden notwendig.

Definition 4.9 (Block, Blockäquivalenz)

1. *Eine Folge $s_0, \ldots, s_n$ mit $L(s_0) = \cdots = L(s_n)$ und n maximal wird **Block** genannt.*
2. *Zwei unendliche Pfade $\sigma = s_0, s_1, \ldots$ und $\rho = r_0, r_1, \ldots$ in einem Modell $\mathcal{M}$ werden genau dann **blockweise äquivalent** genannt, notiert $\sigma \sim_{bl} \rho$, wenn es zwei Folgen natürlicher Zahlen $0 = i_0 < i_1 < \ldots$ und $0 = j_0 < j_1 < \ldots$ gibt, so daß für alle $k \geq 0$ gilt:*

   $$\begin{aligned} L(s_{i_k}) &= L(s_{i_k+1}) = \cdots = L(s_{i_{k+1}-1}) \\ &= L(r_{j_k}) = L(r_{j_k+1}) = \cdots = L(r_{j_{k+1}-1}) \end{aligned}$$

   Für endliche Pfade der Länge n ist die Definition analog mit der Einschränkung $k < n$.
3. *Eine LTL-Formel $\mathbf{A}\,\varphi$ wird genau dann **invariant unter Blockäquivalenz** genannt, wenn für alle Modelle $\mathcal{M}$ und alle Pfade π, π' in $\mathcal{M}$ mit $\pi \sim_{bl} \pi'$ gilt:*

   $$\mathcal{M}, \pi \vDash \varphi \quad \Longleftrightarrow \quad \mathcal{M}, \pi' \vDash \varphi$$

Satz 4.10
Eine LTL-Formel ist genau dann invariant unter Blockäquivalenz, wenn sie in LTL_{-X}, also ohne den Operator $\mathbf{X}$, ausgedrückt werden kann.

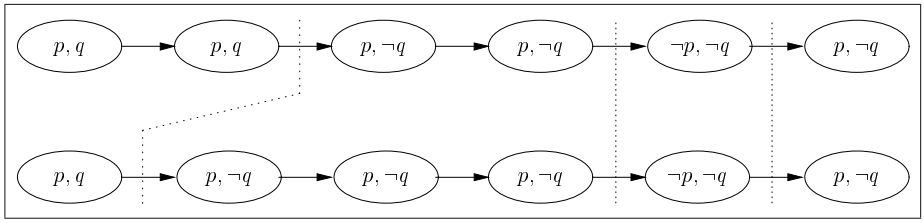

Abbildung 4.5: blockweise äquivalente Pfade [9]

Beweis Seien $\sigma = s_0, s_1, \ldots$ und $\rho = r_0, r_1, \ldots$ blockweise äquivalente Pfade. Wir zeigen durch strukturelle Induktion, daß $\sigma \vDash \varphi \Leftrightarrow \rho \vDash \varphi$ für alle Pfadformeln φ gilt.

1. Wenn $\varphi \in \mathcal{AP}$, dann gilt $\sigma \vDash \varphi \Leftrightarrow s_0 \vDash \varphi \Leftrightarrow \varphi \in L(s_0)$ (analog für ρ). Nach Def. 4.9 gilt $\varphi \in L(s_0)$ genau dann, wenn $\varphi \in L(r_0)$, also genau dann, wenn $\rho \vDash \varphi$.

2. Wenn $\varphi = \varphi_1 \vee \varphi_2$, dann gilt $\sigma \vDash \varphi \Leftrightarrow \sigma \vDash \varphi_1 \vee \sigma \vDash \varphi_2$. Nach Def. 4.9 gilt eine analoge Aussage für ρ. Nach Induktionsvoraussetzung gilt $\sigma \vDash \varphi_1 \Leftrightarrow \rho \vDash \varphi_1$ (analog für φ_2), also auch $\sigma \vDash \varphi \Leftrightarrow \rho \vDash \varphi$.

3. Wenn $\varphi = \varphi_1 \,\mathbf{U}\, \varphi_2$, dann gibt es ein i mit $\sigma^i \vDash \varphi_2$. Nach Induktionsvoraussetung ist das genau dann, wenn es ein j mit $\rho^j \vDash \varphi_2$ gibt. Außerdem gilt dann $\sigma^k \vDash \varphi_2$ für alle $0 \leq k < i$. Also gilt φ_1 in allen Blöcken von σ bis zu ausschließlich dem Block, in dem φ_2 gilt. Nach Induktionsvoraussetzung gelten analoge Aussagen für die Blöcke von ρ.

4. Wir zeigen durch strukturelle Induktion über Zustandsformeln φ_1, daß $\varphi = \mathbf{X}\,\varphi_1$ nicht invariant unter Blockäquivalenz ist. Dazu geben wir jeweils Pfade $\sigma = s_0, s_1, \ldots$ und $\rho = r_0, r_1, \ldots$ mit $\sigma \vDash \varphi \wedge \rho \vDash \varphi$ an.

   (a) Wenn $\varphi_1 \in \mathcal{AP}$, dann wähle die Pfade σ und ρ so, daß $\varphi_1 \in L(s_1) \wedge \varphi_1 \notin L(r_1)$, wie in Abbildung 4.5 für $\varphi_1 = q$.

   (b) Wenn $\varphi_1 = \neg\varphi_2$, dann gilt $\mathbf{X}\,\neg\varphi_1 \equiv \mathbf{X}\,\varphi_2$. Nach Induktionsvoraussetzung gibt es Pfade σ' und ρ' mit $\sigma' \vDash \mathbf{X}\,\varphi_2 \wedge \rho' \nvDash \mathbf{X}\,\varphi_2$. Setze $\sigma := \rho'$ und $\rho := \sigma'$.

   (c) Wenn $\varphi_1 = \varphi_2 \vee \varphi_3$, dann gilt $\mathbf{X}\,\varphi_1 \equiv \mathbf{X}\,\varphi_2 \vee \mathbf{X}\,\varphi_3$. Nach Induktionsvoraussetzung gibt es Pfade σ' und ρ' mit $\sigma' \vDash \mathbf{X}\,\varphi_2 \wedge \rho' \nvDash \mathbf{X}\,\varphi_2$. Wenn $\varphi_3 \not\equiv \neg\varphi_2$, wähle ρ' so, daß auch $\rho' \nvDash \mathbf{X}\,\varphi_3$. Andernfalls wähle einen Pfad ρ' der Länge 1, damit $\rho' \nvDash \mathbf{X}\,\varphi_3$. Setze $\sigma := \sigma'$ und $\rho := \rho'$.

   (d) Wenn $\varphi_1 = \mathbf{X}\,\varphi_2$, dann gibt es nach Induktionsvoraussetzung Pfade σ' und ρ' mit $\sigma' \vDash \varphi_1 \wedge \rho' \nvDash \varphi_1$. Wähle σ und ρ so, daß $\sigma^1 = \sigma'$ und $\rho^1 = \rho'$.

   (e) Wenn $\varphi_1 = \varphi_2 \,\mathbf{U}\, \varphi_3$, dann gibt es nach Induktionsvoraussetzung Pfade σ' und ρ' mit $\sigma' \vDash \mathbf{X}\,\varphi_3 \wedge \rho' \nvDash \mathbf{X}\,\varphi_3$. Wähle σ und ρ so, daß $\sigma = \sigma'$ sowie $\rho = \rho'$ und $\rho^i \nvDash \varphi_3$ für alle $i \geq 1$. □

Definition 4.11
Zwei Modelle $\mathcal{M}$ und $\mathcal{M}'$ werden genau dann blockweise äquivalent genannt, wenn gilt:

1. *$\mathcal{M}$ und $\mathcal{M}'$ haben gleiche Startzustandsmengen.*
2. *Zu jedem Pfad σ von $\mathcal{M}$, der in einem Startzustand s von $\mathcal{M}$ beginnt, existiert ein Pfad σ' von $\mathcal{M}'$ mit $\sigma \sim_{bl} \sigma'$, der ebenfalls in s beginnt.*
3. *Zu jedem Pfad σ' von $\mathcal{M}'$, der in einem Startzustand s' von $\mathcal{M}'$ beginnt, existiert ein Pfad σ von $\mathcal{M}$ mit $\sigma' \sim_{bl} \sigma$, der ebenfalls in s' beginnt.*

Satz 4.12 (Korollar)
Seien $\mathcal{M}$ und $\mathcal{M}'$ blockweise äquivalente Modelle. Dann gilt für alle LTL$_{-X}$-Formeln $\mathbf{A}\,\varphi$ und jeden Startzustand s von $\mathcal{M}$:

$$\mathcal{M}, s \vDash \mathbf{A}\,\varphi \quad \Longleftrightarrow \quad \mathcal{M}', s \vDash \mathbf{A}\,\varphi$$

Wir wollen nun versuchen, den Reduktionsalgorithmus für LTL$_{-X}$ zu konkretisieren. Dazu fehlt nur noch die Berechnungsvorschrift für $ample(s)$. Nach Satz 4.11 muß das reduzierte Modell blockweise äquivalent zum ursprünglichen sein, also für jeden möglichen Pfad einen blockweise äquivalenten Pfad enthalten. Das spiegelt sich in den folgenden Bedingungen für $ample(s)$ wider.

Definition 4.13 ($ample$)
Gegeben sei das Modell eines reaktiven Systems. Dann ist die Funktion $ample : S \to \mathfrak{P}(\mathcal{T})$ wie folgt definiert:

C0 *Für alle Zustände s gilt: $ample(s) = \emptyset \overset{def}{\Longleftrightarrow} enabled(s) = \emptyset$*

C1 *Entlang jedes Pfades des ursprünglichen Modells, der im Zustand s beginnt, gilt: eine Transition, die von einer Transition in $ample(s)$ abhängt, kann nicht ausgeführt werden, bevor zuerst eine Transition aus $ample(s)$ ausgeführt worden ist.*

C2 *Wenn $amples(s) \neq enabled(s)$, dann ist jede Transition $\alpha \in ample(s)$ unsichtbar.*

C3 *Für jede zyklische Transitionsfolge π gilt: Zu jedem Zustand s in π und jeder Transition $\alpha \in enabled(s)$ gibt es einen Zustand s' in π mit $\alpha \in ample(s')$.*

Satz 4.14 (Totale Korrektheit)
Sei $\mathcal{M}$ ein Modell eines reaktiven Systems.

1. *Wenn der Algorithmus aus Abbildung 4.4 mit der Berechnungsvorschrift für $ample(s)$ aus Definition 4.13 terminiert, wird der Zustandsgraph einer zu $\mathcal{M}$ blockweise äquivalenten* Kripke*-Struktur ausgegeben.*
2. *Der Algorithmus terminiert für alle Eingaben $\mathcal{M}$. Insbesondere ist für jeden Zustand s von $\mathcal{M}$ eine Menge $ample(s)$ berechenbar.*

Der vollständige Beweis dieses Satzes erfordert einige technische Vorbereitungen und ist deshalb sehr langwierig. Er ist aber zum Beispiel in [9] zu finden.

Wir wollen im folgenden die in Def. 4.13 angegebenen Bedingungen für $ample(s)$ untersuchen. Bezeichnen $\mathcal{M}$ das ursprüngliche und $\mathcal{M}'$ das für alle Zustände s um die Transitionen aus $enabled(s) \setminus ample(s)$ reduzierte Modell. Wir betrachten jeweils den Fall $ample(s) \neq enabled(s)$ und überprüfen, ob sich daraus Einschränkungen für die blockweise Äquivalenz zwischen $\mathcal{M}$ und $\mathcal{M}'$ ergeben.

Angenommen $enabled(s) = \emptyset$ und $ample(s) \neq \emptyset$. Dann gibt es unnötigerweise Transitionen in $\mathcal{M}'$, die von s ausgehen, ohne ein Verhalten von $\mathcal{M}$ zu repräsentieren. Angenommen $enabled(s) \neq \emptyset$ und $ample(s) = \emptyset$. Dann gibt es Transitionen in $\mathcal{M}$, die von s ausgehen, aber nicht in $\mathcal{M}'$ repräsentiert werden.

Aus **C1** folgt, daß alle Transitionen aus $enabled(s) \setminus ample(s)$ unabhängig von allen Transitionen aus $ample(s)$ sind. Angenommen, es gibt Transitionen $\gamma \in enabled(s) \setminus ample(s)$ und $\delta \in ample(s)$ mit $(\gamma, \delta) \in D$. Da $\gamma \in enabled(s)$, gibt es einen Pfad in $\mathcal{M}$, der mit γ beginnt. Dann ist eine Transition, die von einer Transition in $ample(s)$ abhängt, vor der Ausführung einer Transition aus $ample(s)$ ausgeführt worden. Das ist ein Widerspruch zu **C1**.

Da der Algorithmus nur Pfade betrachtet, deren Zustände aus $ample(s)$ stammen, werden genau diejenigen Pfade nicht untersucht, die einen Präfix $\beta_0, \ldots, \beta_m, \alpha$ mit $\alpha \in ample(s)$ oder die Form $\beta_0, \beta_1, \ldots$ besitzen, wobei jeweils alle Transitionen β_i unabhängig von allen Transitionen aus $ample(s)$ sind.

Wir betrachten den ersten Fall, daß es in $\mathcal{M}$ eine Transitionsfolge $\beta_0, \ldots, \beta_m$, α von s zu r gemäß Abbildung 4.6 gibt. Da alle β_i von α unabhängig sind, kann durch m-malige Anwendung von Def. 4.7 gezeigt werden, daß es mit $\alpha, \beta_0, \ldots, \beta_m$ auch eine Transitionsfolge von s zu r gibt, für die **C1** erfüllt ist. Diese wird vom Algorithmus berücksichtigt. Wenn α unsichtbar ist, sind die Pfade $\sigma = s_0, \ldots, s_m, r$ und $\rho = s, r_0, \ldots, r_m$ außerdem blockweise äquivalent, da dann $L(s_i) = L(r_i)$ für alle $0 \leq i \leq m$.

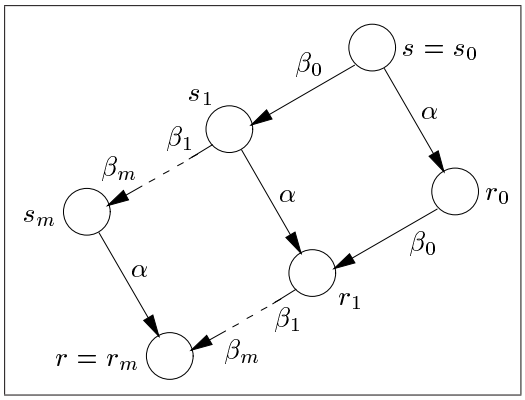

Abbildung 4.6: kommutierende Transitionsfolgen [9]

Nun wird der zweite Fall betrachtet, daß es in $\mathcal{M}$ eine Transitionsfolge $\beta_0, \beta_1, \ldots$, beginnend mit s gibt. Da diese Folge keine Transitionen aus $ample(s)$ enthält, sind nach **C2** alle Transitionen aus $ample(s)$ unsichtbar. Sei $\alpha \in$

ample(s). Da alle β_i von α unabhängig sind, gibt es nach Def. 4.7 eine Folge $\alpha, \beta_0, \beta_1, \ldots$, die in s beginnt und vom Algorithmus berücksichtigt wird. Da α unsichtbar ist, sind auch in diesem Fall die Transitionsfolgen blockweise äquivalent und ermöglichen eine Reduktion.

Es ist möglich, daß die Bedingungen **C0**, **C1** und **C2** erfüllt sind, und trotzdem Verhaltensweisen von $\mathcal{M}$ nicht in $\mathcal{M}'$ präsent sind. Dazu wird das Modell aus Abbildung 4.7 betrachtet. Die Aktionen α_1, α_2 und α_3 sollen jeweils voneinander abhängig aber von der Aktion β unabhängig sein. Dann ist *ample*$(s_1) = \{\alpha_1\}$, *ample*$(s_2) = \{\alpha_2\}$ und *ample*$(s_3) = \{\alpha_3\}$ eine zulässige Reduktion, obwohl β nie ausgeführt wird. Allerdings verletzt sie Bedingung **C3**.

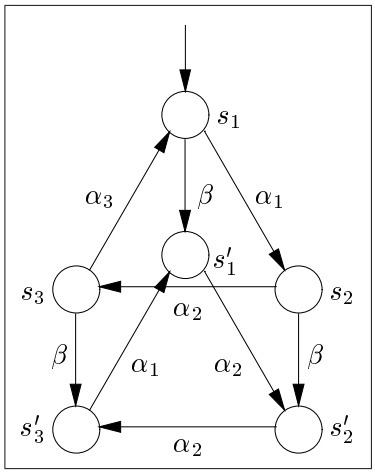

Abbildung 4.7: nebenläufiges System mit Transitionszyklus [9]

Bemerkung 4.15 (Komplexität)
Seien s ein Zustand und $\mathcal{T} \subseteq$ enabled(s) eine Menge von Transitionen eines Modells $\mathcal{M}$. Dann ist die Überprüfung von ***C1*** *für s mindestens so schwer, wie das Erreichbarkeitsproblem für $\mathcal{M}$.*

In der Praxis würde die Überprüfung beliebiger Teilmengen von *enabled*(s) daher am Zustandsexplosionsproblem scheitern, das ja gerade umgangen werden sollte. Man setzt deshalb Heuristiken ein, um einfacher zu überprüfende Mengen *ample*(s) zu erhalten.

4.1.3 Fixpunktanalyse

In JPF werden statische Analyse und Halbordnungsreduktion so kombiniert, daß sich die Vorteile beider Verfahren gegenseitig verstärken [5]. Die zur Konstruktion von *ample*(s) verwendete Methode stellt stärkere Forderungen als nach Definition 4.13 notwendig. Der dadurch entstehende zusätzliche Verifikationsaufwand wird aber durch die einfachere Konstruktion von *ample*(s) kompensiert.

Das hier verwendete Konzept basiert auf der Ermittlung sicherer Transitionen. Wir wollen eine Transition *sicher* nennen, wenn ihre Auswahl in der

Halbordnungsreduktion zulässig ist. Da wir Programme betrachten, sind Transitionen einzelne Anweisungen.

Definition 4.16 (Sicherheit)
*Eine Anweisung α wird genau dann **sicher** genannt, wenn sie unsichtbar und unabhängig von allen Anweisungen anderer Threads ist. Andernfalls wird sie **unsicher** genannt.*

Das Vorgehen zur Bestimmung von $ample(s)$ ist in JPF wie folgt. Wenn es eine sichere Anweisung $\alpha \in enabled(s)$ gibt, ist $ample(s) := \{\alpha\}$, ansonsten ist $ample(s) := enabled(s)$. Um unsichere Anweisungen zu erkennen, wird folgende statische Analysemethode verwendet.

Bemerkung 4.17
*Eine Anweisung α ist **unsicher**, wenn eine der folgenden Eingenschaften gilt:*

1. *α ist **abhängig** von einer anderen Anweisung β, das heißt, α und β gehören zu unterschiedlichen Threads aber besitzen gemeinsam genutzte Datenelemente.*

2. *α ist eine Anweisung zum Betreten eines Monitors oder zum Zugriff auf ein Sperrobjekt.*

Obwohl die zweite Eigenschaft durch statische Analyse entscheidbar ist, kann dies bei der ersten nicht gelingen – die gegenseitige Abhängigkeit zweier Anweisungen ist eine dynamische Eigenschaft. Um die Ermittlung unsicherer Anweisungen zu präzisieren, können aber externe Abhängigkeitsinformationen genutzt werden.

Der Model-Checking-Algorithmus kann feststellen, ob unterschiedliche Threads Referenzen des gleichen Datenelementes besitzen. Dagegen fehlt ihm aber eine globale Sicht auf sichere Berechnungspfade. Da er zur Halbordnungsreduktion nur sichere Anweisungen verwenden darf, benötigt auch er externe Informationen.

Daraus ergibt sich folgende kombinierte Strategie. Das statische Analyseverfahren nutzt bekannte Abhängigkeiten, sogenannte *Aliase*, um eine maximale Menge voneinander unabhängiger Anweisungen zu bestimmen. Daraus ergibt sich eine maximale Menge sicherer Anweisungen. Da immer weniger Anweisungen für Halbordnungsreduktion sicher bleiben, erschließt der Model-Checking-Algorithmus nach und nach immer weitere Teile der Zustandsmenge. Die dabei erkannten Aliase dienen wieder als Eingabe für die statische Analyse.

Definition 4.18
Sei R die Menge aller Referenzen eines Programmes.

1. *Ein Paar $(u, v) \in R \times R$ von Referenzen, die sich auf das gleiche Datenelement beziehen, wird **Alias** genannt.*

2. *Die Menge der zum Zeitpunkt einer Anweisung α geltenden Aliase wird mit $A(t)$ bezeichnet, wobei die Funktion $A : \mathcal{T} \to \mathfrak{P}(R \times R)$ die Aliase des gesamten Programmes beschreibt.*

3. *Die Menge aller möglichen Aliasbelegungen des Programmes wird mit* $\mathcal{A} \stackrel{def}{=} \{A \mid A : \mathcal{T} \to \mathfrak{P}(R \times R)\}$ *bezeichnet.*

4. *Die Funktion* $f_s : \mathcal{A} \to \mathfrak{P}(\mathcal{T})$ *ordnet jeder Menge von Aliasen eine maximale Menge sicherer (unabhängiger) Anweisungen zu.*

5. *Die Funktion* $f_r : \mathfrak{P}(\mathcal{T}) \to \mathfrak{P}(S)$ *ordnet jeder Menge sicherer Anweisungen eine minimale Menge noch zu verifizierender Zustände zu.*

6. *Die Funktion* $f_a : \mathfrak{P}(S) \to A$ *ordnet jeder Menge* S *verifizierter Zustände die Menge aller daraus extrahierten Aliase zu.*

Unter den Werten der Funktionen f_s, f_r und f_a sind die Ergebnisse der Ausführung entsprechender Algorithmen zu verstehen. Das sich ergebende iterative Verfahren wird Fixpunktanalyse genannt und ist in Abbildung 4.8 dargestellt.

Eingabe: ein Programm
Initialisierung: $A_0 := S_0^r := S_0 := \emptyset$, $I_0 := \mathcal{T}$

1. Berechne die Menge unabhängiger Anweisungen $I_{k+1} := f_s(A_k)$ durch statische Analyse. Wende Model Checking auf die Zustandsmenge S_k an und erhalte die Menge S_{k+1} verifizierter Zustände.
2. Berechne daraus $S_{k+1}^r := f_r(I_{k+1})$ durch Halbordnungsreduktion.
3. Ermittle die Menge der Aliase $A_{k+1} := f_a(S_{k+1})$ aus den während des Model Checking beobachteten Referenzen.
4. Falls $A_{k+1} \supset A_k$, gehe zu Schritt 1. Andernfalls terminiere mit Ausgabe von S^r.

Abbildung 4.8: Algorithmus zur Fixpunktanalyse [5]

Satz 4.19 (Totale Korrektheit)

1. *Wenn der Algorithmus terminiert, wurden alle relevanten Pfade durch Model Checking verifiziert.*

2. *Der Algorithmus terminiert für alle eingegebenen* Java*-Programme.*

Beweis

1. Angenommen, es gibt relevante Pfade, die nicht verifiziert worden sind. Sei π ein solcher Pfad. Dann muß es in π eine Anweisung α geben, die von der Halbordnungsreduktion nicht berücksichtigt wurde, da es bereits eine andere sichere Anweisung α' auf einem Pfad $\pi' \neq \pi$ gab. Dann sind $\pi = \pi_1 \stackrel{\alpha}{\to} \pi_2$ und $\pi' = \pi_1' \stackrel{\alpha'}{\to} \pi_2'$, so daß $\alpha, \alpha' \in \mathit{enabled}(s(\pi_1))$, wobei π_1, π_2 sowie π_2' Pfade und $s(\pi)$ den letzten Zustand von π bezeichnen. Da π relevant ist, wurde α' fälschlich als sicher markiert obwohl es unsicher ist. α' kann keine Anweisung nach Def. 4.16(2) sein, da es sonst in Schritt 1 als

unsicher markiert worden wäre. Also ist α' abhängig. Das bedeutet, daß α und α' ein gemeinsam genutztes Datenelement besitzen, der entsprechende Alias aber niemals ermittelt worden ist. Bei der Ausführung von $s(\pi_1)$ wurden aber alle Aliase, die sich auf α und α' auswirken können, entdeckt.

2. $\mathcal{A}$, $\mathcal{T}$ und damit auch S und S^r sind endliche Mengen. Aliase beschreiben monotone Funktionen:

$$\forall (A, B) \in \mathcal{A} : A \subseteq B \Leftrightarrow (\forall \alpha \in \mathcal{T} : A(\alpha) \subseteq B(\alpha))$$

Die Funktion f_s und f_r sind antimonoton, f_a ist monoton:

$$\begin{aligned} \forall (X, Y) \in \mathcal{A} \times \mathcal{A} : &\quad X \subseteq Y \Rightarrow f_s(X) \supseteq f_s(Y) \\ \forall (X, Y) \in \mathfrak{P}(\mathcal{T}) \times \mathfrak{P}(\mathcal{T}) : &\quad X \subseteq Y \Rightarrow f_r(X) \supseteq f_r(Y) \\ \forall (X, Y) \in \mathfrak{P}(S) \times \mathfrak{P}(S) : &\quad X \subseteq Y \Rightarrow f_a(X) \subseteq f_a(Y) \end{aligned}$$

Daraus folgt, daß $A_k \subseteq A_{k+1}$, $I_k \supseteq I_{k+1}$, $S_k^r \subseteq S_{k+1}^r$ und $S_k \subseteq S_{k+1}$. Mit der Abbruchbedingung folgt, daß die Berechnung terminiert. □

Das gezeigte Verfahren enthält bemerkenswerte neue Ideen. Die Kombination von statischer Analyse und Model Checking ist eine effiziente Umsetzung der in [9] vorgeschlagenen Heuristiken. Der Übergang von unsicheren Zwischenergebnissen zu einem sicheren Endergebnis ist neu im Model Checking. Auch mit diesem Algorithmus ist es möglich, daß das Zustandsexplosionsproblem eine vollständige Durchsuchung des Modells verhindert. Dem kann zum Beispiel durch den Einsatz von Kompositionstechniken oder durch Laufzeitanalyse generierte Startzustände (4.3) entgegengewirkt werden.

4.2 Abstraktion

Obwohl *Abstraktion* [7, 9] ein Oberbegriff für mehrere verschiedene Methoden ist, soll hier nur die Theorie der Datenabstraktion mit ihren Anwendungen betrachtet werden. Wir halten uns im ersten Teilabschnitt an die Darstellung von Clarke [9].

4.2.1 Datenabstraktion

Bei der Modellierung von Programmen wird schnell klar, daß die vom Programmierer verwendeten internen Datenstrukturen der Programmiersprache nicht unbedingt die für Model Checking effizientesten sind. Die benutzten Datenbereiche sind meist größzügiger als notwendig gewählt oder es gibt einfache Korrelationen zwischen Zuständen.

Dann besteht die Möglichkeit, *abstrakte Datenbereiche* zu verwenden, die nur so groß sind, wie es die Überprüfung der jeweiligen Spezifikation erfordert. Eine surjektive *Abstraktionsabbildung* $h : D \to A$ weist jedem konkreten Datum ein abstraktes Datum zu. Zum Beispiel wäre es möglich, daß für eine Spezifikation nur das Vorzeichen einer Variablen interessant ist. Dann genügt

ein abstrakter Datenbereich, der nur Elemente für „negativ“, „neutral“ und „positiv“ besitzt. Die Verwendung solcher kleinerer Datenbereiche ermöglicht die Konstruktion von Modellen über kleineren Mengen $\mathcal{AP}$ *abstrakter atomarer Aussagen.*

Definition 4.20 (ideale Abstraktion)
Sei $\mathcal{M} = (S, S_0, \rightarrow, L)$ eine KRIPKE*-Struktur über einer Menge $\mathcal{AP}_r$ abstrakter atomarer Aussagen. Die* KRIPKE*-Struktur $\mathcal{M}_r = (S_r, S_0^r, \rightarrow_r, L_r)$ wird genau dann* ***ideale Abstraktion*** *von $\mathcal{M}$ genannt, wenn gilt:*

1. $S_r \stackrel{def}{=} \{L(s) \mid s \in S\}$
2. $s_r \in S_0^r \stackrel{def}{\Longleftrightarrow} \exists s \in S_0 : s_r = L(s)$
3. $L_r(s_r) \stackrel{def}{=} s_r$
4. $s_r \rightarrow_r t_r \stackrel{def}{\Longleftrightarrow} \exists s, t \in S : s_r = L(s) \wedge t_r = L(t) \wedge s \rightarrow t$

Obwohl ideale Abstraktionen praktisch nicht relevant sind, eignen sie sich, um wesentliche Konzepte zur Untersuchung von Abstraktionen zu erläutern.

Definition 4.21 (Simulation, Bisimulation)
Seien $\mathcal{M} = (S, S_0, \rightarrow, L)$ und $\mathcal{M}' = (S', S_0', \rightarrow', L')$ KRIPKE*-Strukturen über den Mengen $\mathcal{AP} \supseteq \mathcal{AP}'$.*

1. *Eine Relation $H \subseteq S \times S'$ wird genau dann* ***Simulation*** *von $\mathcal{M}$ durch $\mathcal{M}'$ genannt, wenn für alle Zustände $s \in S, s' \in S'$ mit $H(s, s')$ gilt:*
   - $L(s) \cap \mathcal{AP}' = L'(s')$
   - *Für jeden Zustand $t \in S$ mit $s \rightarrow t$ gibt es einen Zustand $t' \in S'$ mit $s' \rightarrow' t'$ und $H(t, t')$.*
2. *Eine Simulation H von $\mathcal{M}$ durch $\mathcal{M}'$ wird genau dann* ***Bisimulation*** *zwischen $\mathcal{M}$ und $\mathcal{M}'$ genannt, wenn auch H^{-1} eine Simulation von $\mathcal{M}$ durch $\mathcal{M}'$ ist.*
3. *Eine* KRIPKE*-Struktur $\mathcal{M}'$* ***simuliert*** *genau dann eine* KRIPKE*-Struktur $\mathcal{M}$, notiert $\mathcal{M} \preceq \mathcal{M}'$, wenn es eines Simulation H von $\mathcal{M}$ durch $\mathcal{M}'$ gibt, so daß es für jeden Zustand $s \in S_0$ einen Zustand $s' \in S_0'$ mit $H(s, s')$ gibt.*
4. *Zwei* KRIPKE*-Strukturen $\mathcal{M}$ und $\mathcal{M}'$ werden genau dann* ***bisimulationsäquivalent*** *genannt, notiert $\mathcal{M} \equiv \mathcal{M}'$, wenn es eine Bisimulation B zwischen ihnen gibt, so daß es für jeden Zustand $s \in S_0$ einen Zustand $s' \in S_0'$ mit $B(s, s')$ und $B(s', s)$ gibt.*

Man sieht, daß $\mathcal{M}_r$ das Verhalten von $\mathcal{M}$ simuliert.

Satz 4.22
Für jede KRIPKE*-Struktur $\mathcal{M}$ gilt:*

$$\mathcal{M} \preceq \mathcal{M}_r$$

Beweis $H = \{(s, s_r) \mid s_r = L(s)\}$ ist eine Simulation[1] von $\mathcal{M}$ durch $\mathcal{M}_r$. □

Lemma 4.23
Seien H eine Simulation der KRIPKE*-Struktur $\mathcal{M}$ durch die* KRIPKE*-Struktur $\mathcal{M}'$ sowie s und s' Zustände von $\mathcal{M}$ bzw. $\mathcal{M}'$ mit $H(s, s')$. Dann gibt es zu jedem Pfad $s_0, s_1, \ldots$ mit $s = s_0$ einen* ***korrespondierenden*** *Pfad $s'_0, s'_1, \ldots$ mit $s' = s'_0$, so daß $H(s_i, s'_i)$ für alle $i \geq 0$.*

Beweis Seien H eine Simulation zwischen den KRIPKE-Strukturen $\mathcal{M}$ und $\mathcal{M}'$ sowie s und s' Zustände von $\mathcal{M}$ bzw. $\mathcal{M}'$ mit $H(s, s')$. Sei $\pi = s_0, s_1, \ldots$ ein Pfad mit $s = s_0$. Da $s_0 \to s_1$ auf π liegt, gibt es nach Def. 4.21 einen Folgezustand s'_1 von $s' = s'_0$ mit $H(s_1, s'_1)$. Durch Induktion folgt, daß diese Transition zu einem Pfad $\pi' = s'_0, s'_1, \ldots$ erweitert werden kann, der zu π korrespondiert.□

LTL-Formeln treffen Aussagen über alle Verhaltensweisen (Pfade) in einem System. Wenn jedes Verhalten von $\mathcal{M}$ durch ein Verhalten von $\mathcal{M}'$ simuliert wird, ist jede in $\mathcal{M}'$ wahre LTL-Formel auch in $\mathcal{M}$ wahr.

Satz 4.24
Seien $\mathcal{M}$ und $\mathcal{M}'$ KRIPKE*-Strukturen mit $\mathcal{M} \preceq \mathcal{M}'$ und $\mathcal{AP}'$ die Menge atomarer Aussagen von $\mathcal{M}'$. Dann gilt für alle LTL-Formeln[2] φ über $\mathcal{AP}'$:*

$$\mathcal{M}' \vDash \varphi \quad \Longrightarrow \quad \mathcal{M} \vDash \varphi$$

Beweis Seien $\mathcal{M}$ und $\mathcal{M}'$ KRIPKE-Strukturen mit $\mathcal{M} \preceq \mathcal{M}'$. Dann gibt es eine Simulation H mit $H(s_0, s'_0)$ für alle Startzustände s_0 und s'_0 von $\mathcal{M}$ bzw. $\mathcal{M}'$. Nach Lemma 4.23 gibt es dann für alle Startzustände s_0 und s'_0 von $\mathcal{M}$ bzw. $\mathcal{M}'$ korrespondierende Pfade $s_0, s_1, \ldots$ in $\mathcal{M}$ und $s'_0, s'_1, \ldots$ in $\mathcal{M}'$.

Sei $\mathcal{AP}'$ die Menge atomarer Aussagen von $\mathcal{M}'$ und $\varphi = \mathbf{A}\,\psi$ eine LTL-Formel über $\mathcal{AP}'$ mit $\mathcal{M}' \vDash \varphi$. Dann gilt $\mathcal{M}', s'_0 \vDash \varphi$ für alle Startzustände s'_0 von $\mathcal{M}'$, also $\mathcal{M}', \pi' \vDash \psi$ für alle Pfade $\pi' = s'_0, s'_1, \ldots$ in $\mathcal{M}'$. Durch strukturelle Induktion über den Aufbau der Pfadformel ψ wird nun gezeigt, daß daraus $\mathcal{M}, \pi \vDash \psi$ für alle Pfade $\pi = s_0, s_1, \ldots$ von Startzuständen s_0 von $\mathcal{M}$ folgt.

1. Wenn $\psi \in \mathcal{AP}'$, dann ist $\psi \in L'(s'_0)$ für alle Startzustände s'_0 von $\mathcal{M}'$. Nach Def. 4.21 ist dann $\psi \in L(s_0)$ für alle Startzustände s_0 von $\mathcal{M}$, also $\mathcal{M} \vDash \psi$.

2. Wenn $\psi = \neg\psi_1$, dann gilt $\mathcal{M}' \nvDash \psi_1$. Nach Induktionsvoraussetzung folgt daraus $\mathcal{M} \nvDash \psi_1$, also $\mathcal{M} \vDash \psi$. Die Fälle $\vee$, $\mathbf{X}$ und $\mathbf{U}$ sind analog dazu. □

Um die ideale Abstraktion eines Systems zu bestimmen, muß dessen KRIPKE-Struktur unreduziert vorliegen. Da dies in vielen Fällen zu aufwendig ist, verfolgt man für Programme einen effizienteren Ansatz. Man stellt die Anweisungen des Programmes durch logische Formeln $\mathcal{S}_0$ und $\mathcal{R}$ dar und beschreibt dann das reduzierte Programm durch entsprechende Formeln $\widehat{\mathcal{S}_0}$ und $\widehat{\mathcal{R}}$.

[1] H^{-1} ist aber im allgemeinen keine Simulation von $\mathcal{M}$ durch $\mathcal{M}_r$.

[2] Dieser Satz gilt sogar für alle ACTL*-Formeln. ACTL* ist die Teilmenge der Formeln von CTL* ohne den Operator $\mathbf{E}$, die in Negationsnormalform vorliegen.

Definition 4.25 (ideale Abstraktion)
Sei $\mathcal{M} = (S, S_0, \rightarrow, L)$ die KRIPKE*-Struktur eines Programmes, so daß gilt:*

1. *$x_1, \ldots, x_n$ sind Variablen über dem* ***Datenbereich*** *D. $\widehat{x_1}, \ldots, \widehat{x_n}$ sind* ***abstrakte Variablen*** *über dem* ***abstrakten Datenbereich*** *A.*

2. *$h : D \rightarrow A$ ist die surjektive* ***Abstraktionsabbildung****.*

3. *Die Zustandsmenge $S \stackrel{def}{=} D^n$ ergibt sich aus den Variablenwerten, wobei genau dann $s = (d_1, \ldots, d_n)$, wenn für alle $i = 1, \ldots, n$ die Variable x_i im Zustand s den Wert d_i hat.*

   S_0 und $\rightarrow$ sind durch Formeln $\mathcal{S}_0$ bzw. $\mathcal{R}$ der Prädikatenlogik erster Stufe gegeben. Deren atomare Teilformeln entsprechen genau den atomaren Operationen des Programmes.

   Die Beschriftungsfunktion L ist wie folgt definiert: Sei $a_i \stackrel{def}{=} h(d_i)$. Bezeichne $\widehat{x_i} = a_i$ die atomare Aussage, daß x_i den abstrakten Wert a_i hat. Dann ist $L(s) \stackrel{def}{=} \{\widehat{x_i} = a_i \mid i = 1, \ldots, n\}$.

4. *Die* KRIPKE*-Struktur $M_r = (S_r, S_0^r, \rightarrow_r, L_r)$, beschrieben durch $\widehat{\mathcal{S}_0}$ und $\widehat{\mathcal{R}}$, wird genau dann* ***ideale Abstraktion*** *von $\mathcal{M}$ genannt, wenn gilt:*

$$\begin{aligned}
\widehat{\mathcal{S}_0} &\stackrel{def}{=} \exists x_1, \ldots, x_n : (\bigwedge_{i=1}^{n} h(x_i) = \widehat{x_i} \wedge \mathcal{S}_0(x_1, \ldots, x_n)) \\
\widehat{\mathcal{R}} &\stackrel{def}{=} \exists x_1, \ldots, x_n, x_1', \ldots, x_n' : \\
&\qquad (\bigwedge_{i=1}^{n} h(x_i) = \widehat{x_i} \wedge \bigwedge_{i=1}^{n} h(x_i') = \widehat{x_i}' \wedge \mathcal{R}(x_1, \ldots, x_n, x_1', \ldots, x_n'))
\end{aligned}$$

5. *Sei Φ eine Formel der Prädikatenlogik erster Stufe mit den freien Variablen $x_1, \ldots, x_m$. Die* ***Abstraktionsoperation*** *$[\cdot]$ bestimmt die dazugehörige Formel $[\Phi]$ über A^m:*

$$[\Phi](\widehat{x_1}, \ldots, \widehat{x_m}) = \exists x_1, \ldots, x_m : (\bigwedge_{i=1}^{n} h(x_i) = \widehat{x_i} \wedge \Phi(x_1, \ldots, x_m))$$

Der Zusammenhang zwischen dem ursprünglichen Modell $\mathcal{M}$ und seiner idealen Abstraktion $\mathcal{M}_r$ wird nun durch $\widehat{\mathcal{S}_0} = [\mathcal{S}_0]$ und $\widehat{\mathcal{R}} = [\mathcal{R}]$ deutlich. Wegen der Größe von $\mathcal{M}_r$ ist es aber immer noch schwierig, daraus S_0^r und R_r zu berechnen.

Man konstruiert stattdessen eine Approximation $\mathcal{M}_a$ für $\mathcal{M}_r$. Dazu wird eine *Transformation* rekursiv auf Φ angewendet. Wir nehmen an, daß sich Φ in Negationsnormalform befindet, also Negation nur auf atomare Formeln direkt angewendet wird.

Definition 4.26 (Transformation)
Die ***Transformation*** *$\mathcal{A}(\Phi)$ einer Formel Φ der Prädikatenlogik erster Stufe ist wie folgt definiert:*

1. *Wenn Φ eine atomare Formel oder deren Negation ist, dann:*
   $\mathcal{A}(\Phi(x_1,\ldots,x_m)) \stackrel{def}{=} [\Phi](\widehat{x_1},\ldots,\widehat{x_m})$
2. $\mathcal{A}(\Phi_1 \vee \Phi_2) \stackrel{def}{=} \mathcal{A}(\Phi_1) \wedge \mathcal{A}(\Phi_2)$ *(analog für $\wedge$)*
3. $\mathcal{A}(\exists x\, \Phi) \stackrel{def}{=} \exists \widehat{x}\, \mathcal{A}(\Phi)$ *(analog für $\forall$)*

Durch Berechnung von $\mathcal{A}(\mathcal{S}_0)$ und $\mathcal{A}(\mathcal{R})$ läßt sich nun die Approximation $\mathcal{M}_a$ eindeutig bestimmen.

Definition 4.27 (Approximation)
Sei $\mathcal{M} = (S, S_0, \rightarrow, L)$ die KRIPKE*-Struktur eines Programmes mit n abstrakten Variablen. Die* KRIPKE*-Struktur $\mathcal{M}_a = (S_a, S_0^a, \rightarrow_a, L_a)$ wird genau dann* ***Approximation*** *von $\mathcal{M}$ genannt, wenn gilt:*

1. $S_a \stackrel{def}{=} A^n$
2. $S_0^a \stackrel{def}{=} \mathcal{A}(\mathcal{S}_0)$
3. $R_a \stackrel{def}{=} \mathcal{A}(\mathcal{R})$
4. $L_a((a_1,\ldots,a_n)) \stackrel{def}{=} \{\widehat{x_i} = a_i \,|i = 1,\ldots,n\}$

Ein Nachteil dieser Vorgehensweise ist, daß die Approximation $\mathcal{A}(\Phi)$ im allgemeinen nicht zu Φ äquivalent ist. Es gilt aber der folgende Satz.

Satz 4.28
Für alle prädikatenlogischen Formeln Φ erster Stufe gilt:

$$[\Phi] \quad \Longrightarrow \quad \mathcal{A}(\Phi)$$

Beweis Der Beweis wird durch strukturelle Induktion über den Aufbau von Φ geführt. Wir vereinbaren, daß $x_1,\ldots,x_m$ die freien Variablen von Φ sind und verwenden die abkürzenden Notationen $\Psi \stackrel{def}{=} \Psi(x_1,\ldots,x_m)$ für alle verwendeten Formeln Ψ.

1. Wenn Φ eine atomare Formel oder deren Negation ist, gilt $\mathcal{A}(\Phi) = [\Phi]$.
2. Wenn $\Phi = \Phi_1 \vee \Phi_2$, dann ist $[\Phi]$ äquivalent zu

   $$\exists x_1,\ldots,x_m : \textstyle\bigwedge_{i=1}^m (h(x_i) = \widehat{x_i} \wedge (\Phi_1 \vee \Phi_2))$$

   Daraus folgt

   $$\begin{array}{rl} \exists x_1,\ldots,x_m : & \bigwedge_{i=1}^m (h(x_i) = \widehat{x_i} \wedge \Phi_1) \\ \vee\, \exists x_1,\ldots,x_m : & \bigwedge_{i=1}^m (h(x_i) = \widehat{x_i} \wedge \Phi_2) \end{array}$$

   Das ist äquivalent zu $[\Phi_1] \vee [\Phi_2]$. Nach Induktionsvoraussetzung folgt daraus $\mathcal{A}(\Phi_1 \vee \Phi_2)$. Dieser Beweisschritt ist vollkommen analog für $\wedge$.

3. Wenn $\Phi = \exists x\, \Phi_1$, dann ist $[\Phi]$ äquivalent zu

$$\exists x_1, \ldots, x_m : \textstyle\bigwedge_{i=1}^{m} (h(x_i) = \widehat{x_i} \wedge \exists x\, \Phi_1(x, x_1, \ldots, x_m))$$

Daraus folgt

$$\exists x : (\, \exists x_1, \ldots, x_m : \textstyle\bigwedge_{i=1}^{m} (h(x_i) = \widehat{x_i} \wedge \exists x\, \Phi_1(x, x_1, \ldots, x_m)))$$

Da die Abstraktionsabbildung surjektiv ist, gilt

$$\exists \widehat{x} : (\, \exists x, x_1, \ldots, x_m : \textstyle\bigwedge_{i=1}^{m} (h(x_i) = \widehat{x_i} \wedge \exists x\, \Phi_1(x, x_1, \ldots, x_m)))$$

Das ist äquivalent zu $\exists \widehat{x}\, \mathcal{A}(\Phi_1)$. Nach Induktionsvoraussetzung folgt daraus $\mathcal{A}(\exists x\, \Phi_1)$. Dieser Beweisschritt ist vollkommen analog für $\forall$.

Satz 4.29
Für jede KRIPKE*-Struktur $\mathcal{M}$ gilt:*

$$\mathcal{M} \preceq \mathcal{M}_a$$

Beweis $H = \{((d_1, \ldots, d_n), (a_1, \ldots, a_n)) \,|\, \forall i = 1, \ldots, n : h(d_i) = a_i\}$ ist eine Simulation zwischen $\mathcal{M}$ und $\mathcal{M}_a$. □

Falls also eine LTL-Formel im approximierten Modell $\mathcal{M}_a$ gilt, dann gilt sie nach Satz 4.24 auch im ursprünglichen Modell $\mathcal{M}$. Falls nicht, ist es immer noch möglich, daß sie in $\mathcal{M}$ gilt. Das liegt daran, daß $\mathcal{M}_a$ zwar alle Verhaltensweisen von $\mathcal{M}$ simuliert, aber zusätzliche Verhaltensweisen haben kann, die dort nicht vorkommen.

Zum Abschluß betrachten wir den Fall, daß sich das Verhalten des abstrakten nicht von dem des konkreten Programmes unterscheidet. Diese Situation liegt vor, wenn unterschiedliche konkrete Variablen nur dann eine gemeinsame abstrakte Variable besitzen, wenn sie sich bereits unter allen atomaren Operationen des konkreten Programmes gleich verhalten haben.

Definition 4.30
Jede Abstraktionsabbildung $h_x : D_x \to A_x$ induziert eine Äquivalenzrelation $\sim_x \subseteq D_x \times D_x$:

$$d \sim_x e \overset{def}{\Longleftrightarrow} h_x(d) = h_x(e)$$

Bemerkung 4.31
Die Äquivalenzrelationen $\sim_{x_i}$ mit $i = 1, \ldots, m$ sind genau dann Kongruenzrelationen bezüglich einer prädikatenlogischen Formel $\Phi \subseteq D_{x_1} \times \cdots \times D_{x_m}$ erster Stufe, wenn gilt:

$$\forall d_1, \ldots, d_m, e_1, \ldots, e_m : (\bigwedge_{i=1}^{m} d_i \sim_{x_i} e_i \Rightarrow (\Phi(d_1, \ldots, d_m) \Leftrightarrow \Phi(e_1, \ldots, e_m)))$$

Satz 4.32 (exakte Approximation)
Genau dann, wenn alle $\sim_{x_i}$ mit $i = 1, \ldots, m$ Kongruenzrelationen bezüglich der atomaren Teilformeln der prädikatenlogischen Formel Φ erster Stufe sind, gilt:

$$[\Phi] \quad \Longleftrightarrow \quad \mathcal{A}(\Phi)$$

Beweis Der Beweis ist analog zum Beweis von Satz 4.28. □

Satz 4.33 (Korollar)
Sei $\mathcal{M}$ die Kripke*-Struktur über $\mathcal{AP}$ eines Programmes mit den Datenbereichen $D_1, \ldots, D_m$, gegeben durch prädikatenlogische Formeln $\mathcal{S}_0$ und $\mathcal{R}$. Genau dann, wenn alle $\sim_{x_i}$ mit $i = 1, \ldots, m$ Kongruenzrelationen bezüglich der atomaren Teilformeln von $\mathcal{S}_0$ und $\mathcal{R}$ sind, gilt:*

1. $\mathcal{M} \equiv \mathcal{M}_a$
2. $\mathcal{M} \models \varphi \Longleftrightarrow \mathcal{M}_a \models \varphi$ *für alle CTL*-Formeln* φ

4.2.2 Prädikatabstraktion

Bei der Modellierung von Java-Programmen entstehen häufig unendliche Datenstrukturen. In diesem Fall müssen endliche Approximationen erfolgen.

In Java PathFinder kann der Nutzer Abstraktionskriterien durch prädikatenlogische Formeln, sogenannte *Abstraktionsprädikate* [33], spezifizieren. Daraus wird mit Hilfe der Software Stanford Validity Checker (SVC) [3] ein abstraktes Programm erzeugt.

Zum Beispiel vergleicht eine Anweisung zwei Integervariablen x und y auf Gleichheit. Diese kann durch den Wert des Prädikates $P : x = y$ abstrahiert werden. Wenn nun Anweisungen den Wert von x oder y ändern, resultiert das in Änderungen von P. Je nach verwendetem Prädikat können sich diese Änderungen nichtdeterministisch auf dessen Wert auswirken. Sei zum Beispiel $P = false$ und es erfolgt die Anweisung $y := y + 1$. Dann ändert sich der Wert des Prädikates entweder zu $true$ (wenn $x = y - 1$) oder er bleibt $false$. Wenn zusätzlich eine Programminvariante $x \leq y$ gilt, verschwindet der Nichtdeterminismus, man erhält eine exakte Approximation.

Da die Bestimmung von Abstraktionsprädikaten im allgemeinen ein tiefes Verständnis des jeweiligen Programmes voraussetzt, ist sie sehr zeitintensiv. Deshalb berechnet JPF automatisch Abstraktionsprädikate, wenn das oben beschriebene Muster vorliegt. Dabei werden auch Abstraktionen zwischen verschiedenen Objekten bzw. Klassen sowie Prädikate zwischen dynamischen Objekten berücksichtigt.

Seien zum Beispiel C und D Klassen mit den Variablen x bzw. y und gelte $C.x \leq D.y$. Dann berechnet JPF ein Prädikat $P : C.x = D.y$. Daraus ermittelt SVC folgende Transformationen vom konkreten in das abstrakte Programm:

$$\begin{aligned} \texttt{C.x} == \texttt{D.y} &\rightsquigarrow \texttt{P} \\ \texttt{C.x} = \texttt{D.y} &\rightsquigarrow \texttt{P} = \texttt{true} \\ \texttt{D.y} = \texttt{D.y} + 1 &\rightsquigarrow \texttt{P} = \texttt{false} \end{aligned}$$

In der Praxis ist es nicht notwendig, daß für jede Kombination dynamischer Objekte ein Prädikat berechnet wird. In JPF wurden deshalb Optimierungen vorgenommen, so daß nur Prädikate zwischen wirklich miteinander interagierenden Objekten ermittelt werden. Durch den Einsatz statischer Analysemethoden sind hier weitere Effizienzsteigerungen denkbar.

4.3 Laufzeitanalyse

Verfahren zur Laufzeitanalyse von Programmen [17, 28] nutzen einen einzelnen Durchlauf[3] eines Programmes zur Ermittlung bestimmter interessanter Eigenschaften. Dafür kommt eine Kombination statischer und dynamischer Analysemethoden zum Einsatz. Laufzeitmethoden können selbst dann Fehler erkennen, wenn der beobachtete Durchlauf fehlerfrei war. Da es sich um nichtformale Methoden handelt, sind aber auch falsche Ergebnisse möglich.

4.3.1 Erkennung zeitkritischer Abläufe

Zur Erkennung zeitkritischer Abläufe wird der von SAVAGE ET. AL. entwickelte Algorithmus LOCKSET [28] eingesetzt.

Ein zeitkritischer Ablauf (siehe 2.1) kann entstehen, wenn mehrere nebenläufige Threads ohne Synchronisation auf eine gemeinsame Variable zugreifen. Die Idee der hier verwendeten Methode ist, die Verwendung mindestens einer Sperre[4] bei jedem konkurrierenden Zugriff zu garantieren.

Für jede Variable v wird eine Menge $S(v)$ der möglichen Sperren verwaltet. In dieser Menge sind alle Sperren enthalten, die jeder bis zu diesem Zeitpunkt auf v zugreifende Thread besaß. Zu Beginn wird $S(v)$ mit der Menge aller existierenden Sperren initialisiert. Wenn ein neuer Thread t auf v zugreift, ergibt sich $S(v) := S(v) \cap locks\_held(t)$. Wenn $S(v) = \emptyset$, wird eine Warnung abgegeben, denn dann ist keine Sperre zur Synchronisation der Zugriffe auf v erkennbar.

Da Zugriffe, die während der Erzeugung eines Objektes erfolgen, nur vom erzeugenden Thread stammen können, werden dort zumeist keine Sperren verwendet. Das gleiche gilt für den ersten Zugriff auf ein Objekt. Außerdem bedeuten nebenläufige Zugriffe keine Gefahr, solange alle von ihnen lesend sind. Der Algorithmus bestimmt deshalb nur dann Mengen $S(v)$, wenn sich v im Zustand $shared - modified$ von Abbildung 4.9 befindet.

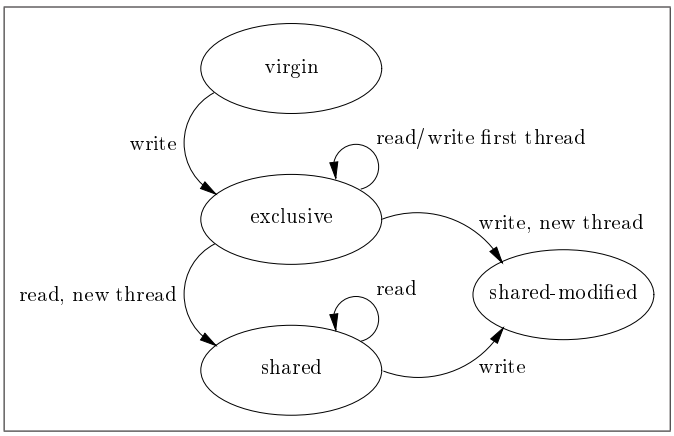

Abbildung 4.9: Zustände einer Variablen in LOCKSET [28]

[3] Bei reaktiven Programmen ist stattdessen eine gewisse Laufzeit notwendig.

[4] Bezogen auf JAVA sind unter diesem Begriff Sperrobjekte und durch Monitore bedingte Sperren zu verstehen.

Zuletzt besteht auch die Möglichkeit, daß spezielle Sperren für Schreiber und Leser benutzt werden. Hier wird der in Abbildung 4.10 angegebene verbesserte Algorithmus angewendet. Wenn ein Schreibvorgang durchgeführt wird, entfernt er diejenigen Sperren aus $locks\_held(t)$, die nur von lesenden Threads gehalten werden. Dann werden nur Sperren aus $write\_locks\_held(t)$ berücksichtigt.

Eingabe: ein Programm, die Menge S aller Sperren des Programms

1. Für jede neu erzeugte Variable initialisiere $S(v) := S$.
2. Arbeite das Programm ab. Wenn dabei ein Thread t im Zustand $shared - modified$ auf eine Variable v zugreift, dann:
   (a) Falls t lesend ist, setze $S(v) := S(v) \cap locks\_held(t)$.
   (b) Falls t schreibend ist, setze $S(v) := S(v) \cap write\_locks\_held(t)$.
   (c) Falls $S(v) = \emptyset$, gib „Warnung“ aus.

Abbildung 4.10: der Algorithmus LOCKSET

Da der Algorithmus schon während des Programmlaufes Ergebnisse erhält, eignet er sich besonders für den Einsatz während des Model Checking.

4.3.2 Erkennung von Verklemmungen

Zur Erkennung von Verklemmungen wird der in [17] vorgestellte Algorithmus GOODLOCK verwendet.

Eine Verklemmung (siehe 2.1) liegt vor, wenn eine Menge nebenläufiger Threads gemeinsame Sperren jeweils in unterschiedlicher Reihenfolge aufgenommen hat. Zum Beispiel benötigen zwei Threads die Sperren L_1 und L_2, um fortgesetzt zu werden. Wenn nun der eine L_1 und der andere L_2 aufgenimmt, warten beide Prozesse für immer auf die Freigabe der jeweils zweiten Sperre.

Betrachten wir das Beispiel in Abbildung 4.11.

```
t1: while(true) {                t2: while(true) {
      synchronized(L1) {                 synchronized(L1) {
        synchronized(L3) {                 synchronized(L2) {
          synchronized(L2) {};               synchronized(L3) {}
          synchronized(L4) {}              }
        }                                  };
      };                                   synchronized(L4) {
      synchronized(L4) {                     synchronized(L3) {
        synchronized(L2) {                     synchronized(L2) {
          synchronized(L3) {                 }
        }                                  }
      }                                  }
    }
```

Abbildung 4.11: Beispiel zur Synchronisation nebenläufiger Threads [17]

Man sieht, daß es zu einer Verklemmung kommen kann, wenn t_1 die obere Sperre $L3$ und t_2 die Sperre $L4$ aufnehmen. Weitere Probleme, zum Beispiel

nach $L3$ im oberen Teil von t_1 und $L2$ im oberen Teil von t_2, werden durch die Sperre $L1$ verhindert.

Der Algorithmus versucht, Verklemmungen durch Analyse sogenannter *lock trees* zu erkennen. Für jeden Thread stellt ein solcher beliebig verzweigender Baum die Reihenfolge dar, in der er Sperren belegt und wieder freigibt. Gleiche Teilbäume, die durch mehrfache Abarbeitung von Programmteilen entstehen, werden nur einmal aufgenommen. In Abbildung 4.12 ist die Konstruktionsvorschrift für lock trees dargestellt.

Eingabe: ein Programm

1. Ordne jedem Thread t den Baum b_t zu, der nur aus einem Wurzelknoten besteht. Ordne ihm einen Zeiger z_t auf diesen Knoten zu.
2. Arbeite das Programm ab:
   - Für jede Anweisung eines Threads t, die eine Sperre L belegt, füge einen mit L beschrifteten Sohn zu z_t hinzu, sofern es noch keinen gibt. Setze z_t auf diesen Sohnknoten.
   - Für jede Anweisung eines Threads t, die eine Sperre L freigibt, setze z_t auf seinen Vaterknoten.

Abbildung 4.12: Algorithmus zur Konstruktion eines lock trees

Für das vorhin betrachtete Beispiel ergeben sich die in Abbildung 4.13 gezeigten Bäume.

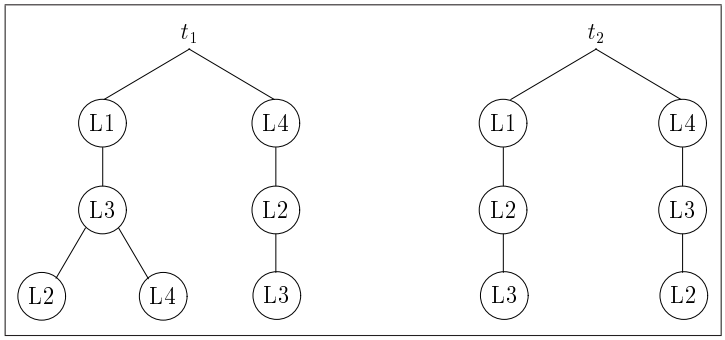

Abbildung 4.13: lock trees zu Abbildung 4.11 [17]

Um eine Menge von Threads auf Verklemmungen zu untersuchen, wird auf deren lock trees das in Abbildung 4.14 angegebene Verfahren angewendet. Dabei wird für alle Paare (t_1, t_2) von lock trees und jeden Knoten n_1 von t_1 überprüft, ob es einen Knoten n_2 von t_2 gibt, der die gleiche Beschriftung wie n_1 und einen Vorfahren mit der gleichen Beschriftung wie ein Nachfahre von n_1 besitzt. Diese Folgen gemeinsamer Sperren führen nur dann nicht zu einer Verklemmung, wenn ihre Ausführung durch eine zusätzliche Sperre synchronisiert wird (wie im obigen Beispiel, wo $L2$ und $L3$ durch $L1$ geschützt sind – in diesem Fall wird $L1$ vom Algorithmus markiert).

Eingabe: eine Menge von Threads, gegeben durch ihre lock trees

- Betrachte alle Paare (b_1, b_2) von lock trees: Führe $prüfe(b_1', b_2)$ für alle Söhne b_1' von b_1 aus.
- $prüfe(s, b)$ ist wie folgt definiert:
  1. Ermittle die Menge S aller Nachfahren von b, die die gleiche Beschriftung wie s und keinen markierten Vorgänger besitzen.
  2. Führe $prüfe\_knoten(s, s')$ für alle $s' \in S$ aus.
  3. Markiere alle Knoten aus S.
  4. Führe $prüfe(s', b)$ für alle Söhne s' von s aus.
  5. Lösche die Markierungen aller Knoten aus S.
- $prüfe\_knoten(s_1, s_2)$ ist wie folgt definiert: Betrachte alle Söhne s_1' von s_1: Bezeichne L die Beschriftung von s_1'. Wenn es einen mit L beschrifteten Vorfahren von s_2 gibt, der nicht markiert ist, gib „Warnung“ aus. Andernfalls führe $prüfe\_knoten(s_1', s_2)$ aus.

Abbildung 4.14: Algorithmus zur Verklemmungsanalyse zweier lock trees

Der Algorithmus benötigt mindestens einen Programmlauf, um zu Ergebnissen zu kommen, da er zunächst die lock trees konstruieren muß. Trotzdem kann er während des Model Checking eingesetzt werden.

Kapitel 5

Das System Java PathFinder

Im Mittelpunkt dieses Kapitels steht die Umsetzung der vorgestellten theoretischen Grundlagen im Softwarewerkzeug Java PathFinder.

Der erste Abschnitt gibt einen Überblick über die Systemarchitektur, die intern verwendete Zustandsdarstellung und dort benutzte dynamische Reduktionsmethoden. Im zweiten Teil erfolgt eine Bewertung des Systems nach den Kriterien Korrektheit, Leistungsvermögen und Handhabbarkeit.

5.1 Systemaufbau

5.1.1 Architektur

Der Kern von JPF besteht aus der speziell für Model Checking entworfenen Java Virtual Machine[1] JVM^{JPF} und einem Algorithmus zur Tiefensuche [4, 32]. Die JVM soll sich weniger durch hohe Geschwindigkeit als vielmehr eine gute Speicherverwaltung auszeichnen. Der Algorithmus arbeitet mit einer hochstrukturierten expliziten Zustandsrepräsentation, die im nächsten Abschnitt beschrieben wird. Beide Einheiten sind selbst in Java programmiert und werden von einer kommerziellen JVM ausgeführt. Zur Manipulation von Klassendateien wird das Paket JavaClass[2] genutzt.

Zu überprüfende Spezifikationen können von der Kommandozeile oder aus einer separaten Datei als LTL-Formeln eingelesen oder durch Methoden der Klasse `Verify()` als Invarianten formuliert werden. Spezifikation und Programm müssen als kompilierter Bytekode eingegeben werden.

Der Algorithmus zur Tiefensuche traversiert den Zustandsgraphen, indem er der JVM Anweisungen gibt, einen Schritt vor oder zurück zu gehen oder Boolesche Ausdrücke auszuwerten. Damit er nichtdeterministische Entscheidungen treffen kann, stellt die Klasse `Verify` die Methoden `random()` und `randomBool()` zur Verfügung. Die JVM führt dafür das Programm schrittweise aus, wobei die Atomizität auf einzelne Bytekodeanweisungen, Java-Anweisungen oder Programmblöcke eingestellt werden kann.

[1] So nennt man eine virtuelle Maschine, welche Java-Bytekode ausführt.

[2] Das Paket ist jetzt Teil der Bytecode Engineering Library, siehe `http://bcel.sourceforge.net/`.

In JPF integrierte Teile des BANDERA-Projektes ermöglichen statisches Slicing, wobei die Slicingkriterien direkt aus der LTL-Formel einer Spezifikation gewonnen werden. Da BANDERA dem Autor nicht rechtzeitig zur Verfügung stand, konnte das nicht überprüft werden.

Die Traversierung des Zustandsgraphen erfolgt unter Verwendung des in 4.1.3 vorgestellten Verfahrens zur Fixpunktanalyse. Durch die variable Atomizität können dabei sichere Blöcke übersprungen werden.

Nutzerdefinierte Abstraktionsprädikate können mit der Methode `Abstract.addBoolean()` definiert werden, `Abstract.remove()` entfernt Objekte aus dem abstrakten Modell. Der STANFORD VALIDITY CHECKER ermittelt daraus das abstrakte Programm. Die automatische Berechnung von Prädikaten anhand bekannter Muster – wie in 4.2.2 beschrieben – stand in der dem Autor vorliegenden Programmversion noch nicht zur Verfügung. Auch die Auswertung nutzerdefinierter Prädikate gelang nicht, da JPF hier auf BANDERA angewiesen ist.

Die Algorithmen zur Laufzeitanalyse werden eingesetzt, um einen günstigen Startbereich für anschließendes Model Checking zu ermitteln. Model Checking wiederum erhöht die Genauigkeit der Algorithmen durch das Ausschließen fehlerhafter Warnungen. Selbst wenn ein Modell für Model Checking zu groß ist, kann eine separat durchgeführte Laufzeitanalyse Probleme erkennen.

5.1.2 Modellierung von JAVA-Programmen

JAVA-Programme sind im allgemeinen nebenläufige, reaktive Systeme. Da sie nicht als Entwurf sondern bereits in ihrer Implementierung vorliegen, ist eine effiziente Zustandsmodellierung schwierig. Potentiell unendlich große dynamische Datenstrukturen[3] müssen auf endliche Modellstrukturen abgebildet werden. Sie führen zu riesigen Zustandsräumen, die effizient kodiert werden müssen, um handhabbar zu sein. Dazu werden in Kapitel 4 angegebenen Verfahren zur Abstraktion auf endliche Modelle sowie zur statischen Reduktion des Zustandsraumes verwendet. Im folgenden wird ein effizientes Schema zur Modellierung von JAVA-Programmen vorgestellt [23].

Da zur Überprüfung einiger Eigenschaften von JAVA-Programmen – insbesondere wenn man zeitkritische Abläufe betrachtet – die Unterscheidung einzelner Bytekodeanweisungen notwendig ist (nur diese Anweisungen sind für die JVM atomar [15], siehe Beispiel im nächsten Abschnitt), muß auch die Zustandskodierung diese Granularität aufweisen. Jeder Zustand eines JAVA-Programmes (in der JVM) besteht im wesentlichen aus drei Teilen:

Feld für statische Daten Werte statischer Felder, Monitore und Sperren jeder Klasse

Feld für dynamische Daten Werte statischer Felder, Monitore und Sperren jedes Objektes

Threadliste Status, Schedulinginformation usw. für jeden Thread

[3] Genauer gesagt handelt es sich jeweils um beliebig große endliche Strukturen.

Auf diese Weise lassen sich theoretisch beliebig große Zustände repräsentieren. Diese dynamischen Zustände können also nicht durch beschränkt viele BOOLEsche Variablen beschrieben werden – ihr Speicherbedarf ist immer noch enorm. Deshalb bedient man sich hier einer Variante der von SPIN verwendeten Methode *Collapse* [20]. Dabei werden komplexe Zustände auf Indices „kollabiert“, man erzeugt geordnete Mengen bereits bekannter Zustandskomponenten (eine Menge für jede der drei genannten Komponenten), so daß man nur noch mit deren Indices arbeiten muß. Damit kann jeder Zustand als Vektor von Integerwerten gespeichert werden. Auch der Vergleich zweier Zustände, um bereits besuchte Zustände zu ermitteln, vereinfacht sich erheblich.

Um den Speicherbedarf weiter zu verringern, werden die Zustände komprimiert gespeichert. Da sich benachbarte Zustände oftmals nur geringfügig unterscheiden, faßt man folgende Komponenten vor der Kompression zusammen: Felddaten, Monitordaten (jeweils statische und dynamische gemeinsam) sowie Methodenkellerelemente und selten geänderte Threadinformationen wie der Threadstatus.

Ein weiterer neuer Ansatz ist die Verwendung von *optimistischem Backtracking*. Anstatt jeden bereits besuchten Zustand zu speichern, werden nur noch Referenzen auf die komprimierten Zustände gespeichert und bei Bedarf rekonstruiert (nur geänderte Komponenten werden dekomprimiert). Ein Effizienzvergleich der hier besprochenen Verfahren ist in Abbildung 5.1 zu sehen. Die verwendeten Beispiele sind unter `http://ase.arc.nasa.gov/jpf/` zu finden.

`RemoteAgent`	**Zustände**	**Transitionen**	**Komponenten**
	66425	148825	1373
	Speicher (MB)	**Zeit** (s)	**Zustandsgröße** (byte)
ohne Kompression	180,79	227,83	2854
Collapse	12,08	138,65	191
optim. Backtracking	12,08	54,39	191

`BoundedBuffer`	**Zustände**	**Transitionen**	**Komponenten**
	105682	275988	583
	Speicher (MB)	**Zeit** (s)	**Zustandsgröße** (byte)
ohne Kompression	504,82	665,90	5009
Collapse	28,17	297,40	445
optim. Backtracking	28,16	76,82	445

Abbildung 5.1: Effizienzvergleich von Zustandskodierungsverfahren [23]

5.1.3 Dynamische Reduktion

Die hier vorgestellten *dynamischen Reduktionsverfahren* [23] beziehen sich auf die interne Zustandsrepräsentation von JAVA PATHFINDER. Ihr Ziel ist es, negative Auswirkungen dynamischer Effekte von JAVA-Programmen auf die Leistungsfähigkeit der JAVA Virtual Machine von JPF zu minimieren.

Symmetriereduktion

Die Zustände eines JAVA-Programmes werden innerhalb von JPF komponentenweise in dynamischen Feldern gespeichert. Durch nichtdeterministische Effekte kann es vorkommen, daß Klassen bzw. Objekte zu verschiedenen Zeitpunkten jeweils an unterschiedlichen Positionen eines Feldes abgelegt werden. Das führt zu einer erheblichen Vergrößerung des Modells um neue Zustände und Transitionen.

Um dies zu verhindern strebt man eine kanonische Repräsentation der internen Zustände an. Da diese oftmals aufwendig zu realisieren ist, berechnet man sie hier nicht vorab sondern dynamisch während des Model Checking. Dazu weist man jedem Objekt bzw. jeder Klasse beim erstmaligen Laden (nach der Erzeugung) eine feste Position im entsprechenden Feld zu. Die Abbildungsvorschrift bleibt auch nach dem Backtracking erhalten.

Diese Abbildung ist wie folgt definiert: Die Position jeder Klasse wird anhand ihres Namens bestimmt. Da Objekte keine Namen besitzen müssen, werden sie anhand ihrer Bytekode-Anweisungen identifiziert. Nun kann es immer noch geschehen, daß Anweisungen auf einem Berechnungspfad mehrfach ausgeführt werden. Deshalb wird zu jedem Objekt zusätzlich ein Zähler gespeichert und die kanonische Position aus Bytekode und Zähler berechnet. Da in nebenläufigen System der gleiche Bytekode in unterschiedlichen Threads vorkommen kann, wird auch der ein Objekt bzw. eine Klasse erzeugene Thread mit in die Abbildungsvorschrift aufgenommen.

Für das unter `http://ase.arc.nasa.gov/jpf/` angegebene Programm `BoundedBuffer` (6-elementiger Puffer mit je 10 Erzeugern und Verbrauchern) wurde eine Symmetriereduktion von 2.502.761 auf 105.682 Zustände (also 25-fach) bei einem zeitlichen Mehraufwand für die Reduktion von einem Prozent erreicht [23]. Da das Verfahren nur auf die Repräsentation in JPF, nicht aber auf das Modell selbst wirkt, ist es orthogonal zu vielen klassischen Reduktionsverfahren für Symmetrien, die zum Beispiel in [9] zur Sprache kommen.

Garbage Collection

Eine Eigenheit von JAVA ist, daß es keine Anweisung zum expliziten Löschen nicht mehr benötigter Objekte gibt. Das erledigt in der JVM der Prozeß *Garbage Collection*. Dieser entfernt Objekte, die nicht mehr durch Variablen referenziert werden, nach einiger Zeit[4]. In der JVM von JAVA PATHFINDER wurden zwei klassische Algorithmen zur Garbage Collection implementiert [23].

Der Algorithmus REFERENCE COUNTING [22] zählt alle Referenzen auf Objekte und speichert diese in einer Tabelle. Wenn eine Objektreferenz in einer Variablen gespeichert wird, inkrementiert er den Zähler dieses Objektes und dekrementiert den des vorher dort gespeicherten. Wenn der Referenzzähler für ein Objekt Null ist, wird es entfernt. Eine Schwäche dieses Algorithmus ist, daß er Objekte, die sich nur noch gegenseitig referenzieren, aber nicht mehr durch eine Variable angesprochen werden können, nicht löscht.

Der Algorithmus MARK AND SWEEP [22] betrachtet Referenzen und Objekte als Knoten eines gerichteten Graphen, dessen Wurzelknoten die Variablen

[4]Für eine genauere Betrachtung siehe zum Beispiel S.246f. in [15].

sind. Wenn ein Knoten von keinem Wurzelknoten aus erreichbar ist, wird er entfernt. Diese Erreichbarkeitsprüfung muß nach jeder Anweisung, die Garbage verursachen kann, durchgeführt werden.

5.2 Bewertung

Die nachfolgenden Untersuchungen wurden mit der Version „JPF 0.1, JVM 0.1, Build 10 (02/27/02)“ unternommen. Die Software kam auf einem PC mit einem Prozessor vom Typ Pentium 3 mit 1133 MHz, 512 MB RAM und 64 MB Speicher für die JVM zum Einsatz. Alle Programmbeispiele sind in Anhang A zu finden. Einige der Befehle von JPF werden in der teilweise veralteten Anleitung [16] erklärt.

Die Überprüfung beliebiger LTL-Formeln gelang nicht. Nach dem erfolgreichen Umsetzen der Spezifikation $\mathbf{A}\,\varphi$ zu $\mathbf{E}\,\neg\varphi$ stoppte die anschließende Verifikation für jede Formel mit einer `ArrayIndexOutOfBoundsException` von JPF. Der Autor vermutet, daß die Einbindung von BANDERA oder eine aktuelle Version von JPF hier Abhilfe schaffen könnte.

Es konnten aber erfolgreich Invarianten formuliert werden (siehe Abbildung 5.2). Dazu dienen die in Abbildung 5.2 gezeigten Methoden, die an beliebigen Stellen des zu überprüfenden Programmes aufgerufen werden. Die Kombination mehrerer Invarianten ist ungefähr so ausdrucksstark wie eine eingeschränkte Pfadformel. Durch die erheblich höhere Beschreibungskomplexität, vermutlich schlechtere Effektivität der Auswertung und mangelnde Reduktionsmöglichkeiten sinkt jedoch die tatsächliche Leistungsfähigkeit.

`Verify.assert(boolean b)`	Deklaration einer Invarianten `b`
`Verify.beginAtomic()`	Beginn eines für JPF atomaren Bereiches
`Verify.endAtomic()`	Ende eines für JPF atomaren Bereiches

Abbildung 5.2: Methoden zur Deklaration von Invarianten

Mit Model Checking wurde das Programm `Datarace`, welches einen zeitkritischen Ablauf enthält, auf eine entsprechende Invariante überprüft. JPF findet nach 64 untersuchten Zuständen, 1,5 s Rechenzeit und mit 1 MB Speicherplatz einen zeitkritischen Pfad der Länge 63. Im Simulationsmodus – der auf Heuristiken und Laufzeitanalyse basiert – werden mit ungefähr gleichem Aufwand kürzere Pfade zu Gegenbeispielen ermittelt (schwankende Ergebnisse, da nichtdeterministisch), zum Beispiel eines nach 22 Anweisungen.

Außerdem sollten Verklemmungen im Programm `Deadlock1` erkannt werden. Mit Model Checking konnte nach 29 untersuchten Zuständen, 1,4 s Rechenzeit und mit 490 KB Speicherplatz ein Pfad der Länge 27 zu einer Verklemmung gefunden werden. Der Simultationsmodus findet einen unerwünschten Pfad der Länge 15, benötigt aber fast doppelt soviel Speicherplatz. Im abgewandelten Programm `Deadlock2` sind Verklemmungen nur auf der Bytekodeebene zu erkennen, da jeder Thread alle seine Sperren innerhalb einer Anweisungszeile belegt. Das Model Checking benötigt die in Abbildung 5.3 angegebenen Ressourcen.

Atomizität	`Zustände`	`Speicher`	`Zeit`
Programmzeilen	49	800 KB	1,3 s
Bytekodeanweisungen	182	1 MB	1,5 s

Abbildung 5.3: Effektivität bei verschiedener Atomizität

Für abschließende Effizienzbetrachtungen soll eine parametrisierte Variante des Philosophenproblems – implementiert im Programm `Philosophers` – betrachtet werden. Dieses Problem sei kurz beschrieben: n Philosophen sitzen um einen runden Tisch, auf dem sich n Teller Spaghetti und n Gabeln – jeweils eine zwischen zwei benachbarten Tellern – befinden. Sie beschäftigen sich abwechselnd mit Essen und Denken. Zum Essen benötigen sie zwei Gabeln. Wenn nun jeder gleichzeitig seine rechte Gabel zum Essen greift, kann eine Verklemmung entstehen, da dann niemand mehr seine linke Gabel nehmen kann. Die Untersuchungen dazu sind in Abbildung 5.4 dargestellt.

Es zeigt sich, daß der Model-Checking-Algorithmus ohne Slicing und Abstraktion schnell an die durch das Zustandsexplosionsproblem gegebenen Grenzen stößt. Die ermittelten Gegenbeispiele sind vergleichsweise sehr groß. Die optionale Nutzung der Fixpunktanalyse führt in diesem Beispiel dazu, daß der benutzte Speicher auf niedrigem Niveau konstant bleibt, allerdings ist die dafür benötigte Laufzeit nicht akzeptabel.

Modus[5]	`Zustände`	`Speicher`	`Zeit`
$n = 5$, MC	1778	3,2 MB	3,1 s
$n = 10$, MC	-	`OutOfMemoryError`	18 s
$n = 10$, MC, HOR	30.000.000	370–890 kByte	60 Minuten[6]
$n = 5$, SIM (5-mal)	57–273	0,6–1,4 MB	1,4–1,7 s
$n = 50$, SIM (5-mal)	1690–2448	11–15 MB	4,2–4,9 s
$n = 100$, SIM (5-mal)	3418–4192	36–52 MB	8–11 s
$n = 200$, SIM (5-mal)	-	`OutofMemoryError`	12 s

Abbildung 5.4: Effektivität von Model-Checking und Simulation

Die Laufzeitanalyseverfahren haben in allen Konfigurationen von JPF die enthaltenen Fehler gefunden. Sie konnten nahezu optimale Gegenbeispiele vorweisen. Durch Einschränkung der Suchtiefe können allerdings im Simulationsmodus Fehler unerkannt bleiben.

Zum Schluß soll die Bedeutung der internen Garbage Collection veranschaulicht werden. Als Beispiel dient das Programm `Garbage`, das einen unendlichen Zustandsraum hat (siehe Abbildung 5.5).

Garbage Collection	`Zustände`	`Speicher`	`Zeit`
aktiviert	7	530	1,2 s
deaktiviert	-	`OutOfMemoryError`	8 s

Abbildung 5.5: Effektivität der Garbage Collection

[5] MC... Model Checking, SIM... Simulation, HOR... Halbordnungsreduktion
[6] Dieser Versuch wurde mangels Ergebnis nach 60 Minuten manuell abgebrochen.

Kapitel 6

Zusammenfassung

Java PathFinder integriert aktuelle Verifikationsmethoden und vorher unbekannte Ansätze zur direkten Anwendung auf Java-Programme. Die wichtigsten davon sind:

- Model Checking von Bytekode, vollständige Abdeckung der Programmiersprache Java; eigene Java Virtual Machine, Nichtdeterminismus
- hochstrukturierte, leistungsfähige Zustandsdarstellung; Anwendung dynamischer Reduktionsmethoden bei Symmetrien und Garbage Collection
- neuartige Kombination von statischer Analyse und Halbordnungsreduktion, die iterativ von unsicheren zu einem sicheren Ergebnis führt
- Datenabstraktion über nutzerdefinierte und automatisch generierte Prädikate, die sowohl statische als auch dynamische Datenelemente abdecken
- kombinierter Einsatz von Laufzeitanalyse und Model Checking

Einige dieser Verfahren können erst in Kombination mit Bandera genutzt werden. Die Entwickler von JPF behaupten, daß Programme bis zu einer Länge von 10.000 Zeilen halbautomatisch überprüft werden können, wobei eine vollständige Verifikation von durchschnittlich 1.000 Zeilen pro Person und Tag realistisch ist.

Für die weitere Entwicklung von JPF existiert folgende Planung:

- Integration von Bandera zur Verarbeitung von LTL-Formeln und zur Kombination von Slicingmethoden mit Abstraktion und Laufzeitanalyse
- höherer Automatisierungsgrad, nutzerfreundlichere Bedienung; bessere Skalierbarkeit, Generierung von *Coverage*-Statistiken
- verteiltes Model Checking durch dynamische Partitionierung des Zustandsraumes [23]
- Generierung kurzer plausibler Gegenbeispiele [27]
- Konzentration auf den Nachfolger Java PathExplorer (JPaX) [19], der auf dem Termersetzungssystem Maude aufbaut und unter anderem theoretisch beliebige Logiken verwenden kann

Interessante Themen für weiterführende Arbeiten sind:

- Untersuchung von Spezifikationsmustern [12, 13] zur Unterstützung formaler Entwurfsmethoden und automatischer Abstraktion
- Untersuchung weiterer Verfahren zur Effizienzsteigerung, zum Beispiel Symmetriereduktion, Komposition und Induktion
- Vergleich verschiedener Softwarewerkzeuge, zum Beispiel Spin, Bandera, JPF, JPaX; dabei Nutzung von Softwaremetriken zur Auswahl repräsentativer Beispiele und statistischer Methoden zur Auswertung praktischer Versuche

Anhang A

Beispiele

A.1 Datarace

```
1  import gov.nasa.arc.ase.jpf.jvm.Verify;
2
3  public class Datarace {
4    private static int value = 0;
5
6    public static void main(String[] args) {
7      Task task1 = new Task();
8      Task task2 = new Task();
9      task1.start();
10     task2.start();
11     while (true) {
12       Verify.assert(value >= task1.getValue() + task2.getValue());
13     }
14   }
15
16   public static void addValue(int value) {
17     Datarace.value = value;
18   }
19
20   public static int getValue() {
21     return value;
22   }
23 }
24
25 class Task extends java.lang.Thread {
26   private int value;
27
28   public void run() {
29     while(true) {
30       Datarace.addValue(1);
31       value = value + 1;
32     }
33   }
34
35   public int getValue() {
36     return value;
37   }
38 }
```

A.2 Deadlock1

```
1  import gov.nasa.arc.ase.jpf.jvm.Verify;
2
3  public class Deadlock1 {
4    public static void main(String[] args) {
5      new Task(0).start();
6      new Task(1).start();
7    }
8  }
9
10 class Task extends java.lang.Thread {
11   private int id;
12   private static Object l0 = new Object();
13   private static Object l1 = new Object();
14
15   public Task(int id) {
16     this.id = id;
17   }
18
19   public void run() {
20     while(true) {
21       if (id == 0) {
22         synchronized(l0) {
23           synchronized(l1) {}
24         }
25       }
26       else {
27         synchronized(l1) {
28           synchronized(l0) {}
29         }
30       }
31     }
32   }
33 }
```

A.3 Deadlock2

```
1  import gov.nasa.arc.ase.jpf.jvm.Verify;
2
3  public class Deadlock2 {

   ⋮

19   public void run() {
20     while(true) {
21       if (id == 0)
22         synchronized(l0) { synchronized(l1) {}}
23       else
24         synchronized(l1) { synchronized(l0) {}}
25     }
26   }
27 }
```

A.4 Philosophers

```
1  import gov.nasa.arc.ase.jpf.jvm.Verify;
2
3  public class Philosophers {
4    private final static int N = 5;
5    private final static Object[] FORK = new Object[N];
6    private static Philosopher[] philArray = new Philosopher[N];
7
8    static {
9      for (int i = 0; i < N; i++) {
10       FORK[i] = new Object();
11       philArray[i] = new Philosopher(i);
12     }
13   }
14
15   public static void main(String[] args) {
16     for (int i = 0; i < N; i++)
17      philArray[i].start();
18   }
19
20   public static void eat(int id) {
21     synchronized(Philosophers.FORK[id % N]) {
22       synchronized(Philosophers.FORK[(id + 1) % N]) { // eat
23       }
24     }
25   }
26 }
27
28 class Philosopher extends java.lang.Thread {
29   private int id;
30
31   public Philosopher(int id) {
32     this.id = id;
33   }
34
35   public void run() {
36     while (true) { // think
37       Philosophers.eat(id);
38     }
39   }
40 }
```

A.5 Garbage

```
1  import gov.nasa.arc.ase.jpf.jvm.Verify;
2
3  public class Garbage {
4    public static void main(String[] args) {
5      while (true)
6        newObject();
7    }
8    public static Object newObject() {
9      return new Object();
10   }
11 }
```

Abbildungsverzeichnis

Literaturverzeichnis

[1] Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*, Toronto, Kanada, 2001.

[2] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, Portland, USA, 2002.

[3] Clark Barrett, David Dill, and Jeremy Levitt. Validity Checking for Combinations of Theories with Equality. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, USA, 1996.

[4] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Java PathFinder - Second Generation of a Java Model Checker. In *Proceedings of the Workshop on Advances in Verification (WAV '00)*, Chicago, USA, 2000.

[5] Guillaume Brat and Willem Visser. Combining Static Analysis and Model Checking for Software Analysis. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*, San Diego, USA, 2001.

[6] Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[7] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*, Albuquerque, USA, 1992.

[8] Edmund M. Clarke, Orna Grumberg, and David E. Long. Verification Tools for Finite-State Concurrent Systems. In *Proceedings of 'A Decade of Concurrency: Reflections and Perspectives' (REX School/Symposium)*, volume 803 of *Lecture Notes in Computer Science*, Noordwijkerhout, Niederlande, 1993.

[9] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, USA, London, UK, 2001.

[10] Edmund M. Clarke and Jeanette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[11] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the International Conference on Software Engineering (ICSE '00)*, Limerick, Irland, 2000.

[12] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP '98)*, Clearwater Beach, USA, 1998.

[13] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, Los Angeles, USA, 1999.

[14] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Robby, Hongjun Zheng, and Willem Visser. Tool-Supported Program Abstraction for Finite-State Verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, Toronto, Kanada, 2001.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition*. Addison-Wesley, Reading, USA, 2000.

[16] Klaus Havelund. Java PathFinder User Guide. Technical report, NASA Ames Research Center, Recom Technologies, Moffett Field, USA, 1999.

[17] Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking of Software (SPIN '00)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford University, Stanford, USA, 2000.

[18] Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 1999.

[19] Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop on Runtime Verification (RV '01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, Paris, Frankreich, 2001.

[20] Gerard J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings of the 3rd International SPIN Workshop (SPIN '97)*, Twente University, Enschede, Niederlande, 1997.

[21] Michael R. A. Huth and Mark D. Ryan. *Modelling and Reasoning about Systems*. Logic in Computer Science. Cambridge University Press, Cambridge, UK, 2000.

[22] Radu Iosif and Riccardo Sisto. Using Garbage Collection in Model Checking. In *Proceedings of the 7th International SPIN Workshop on Model Checking of Software (SPIN '00)*, volume 1885 of *Lecture Notes in Computer Science*, Stanford University, Stanford, USA, 2000.

[23] Flavio Lerda and Willem Visser. Addressing Dynamic Issues of Program Model Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*, Toronto, Kanada, 2001.

[24] Orna Lichtenstein and Amir Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of the 12th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85)*, New Orleans, USA, 1985.

[25] Kenneth L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, USA, 1992.

[26] Stefan Middendorf and Reiner Singer. *Programmierhandbuch und Referenz für die Java-2-Plattform*. dpunkt-Verlag, Heidelberg, 1999.

[27] Corina S. Păsăreanu, Matthew B. Dwyer, and Willem Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *Lecture Notes in Computer Science*, Genua, Italien, 2001.

[28] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, Saint-Malo, Frankreich, 1997.

[29] Colin Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer, Berlin, 2001.

[30] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Hanser, München, 1995.

[31] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[32] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE '00)*, Grenoble, Frankreich, 2000.

[33] Willem Visser, Seung-Joon Park, and John Penix. Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP '00)*, Portland, USA, 2000.