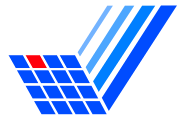
UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK



Marco Bakera

Pläne & Spiele
Eine Anwendung spielbasierten Model Checkings

Diplomarbeit

8. August 2006

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl 5 (Programmiersysteme und Übersetzerbau)
Fachbereich Informatik
Universität Dortmund

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Markus Müller-Olm

GERMANY · D-44221 DORTMUND

Hiermit erkläre ich, Marco Bakera, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Zitate habe ich stets kenntlich gemacht.

______________________ ___________________ ______________________________________

Inhaltsverzeichnis

1 Einleitung und Motivation **9**

2 Model Checking **11**

2.1 Kripke-Strukturen und Transitionssyteme 12

2.1.1 Kripke-Strukturen 12

2.1.2 Labeled Transition Systems 13

2.1.3 Kripke-Transitionssysteme 14

2.2 Temporale Logiken 15

2.2.1 Linearzeitlogiken 16

2.2.2 Branching-Time Logik 19

3 Sichten, Konzepte und Aspekte **29**

3.1 Konzepte und Aspekte 32

3.2 Beispiele für einen Wechsel der Sichten 32

3.2.1 Binomialkoeffizienten 33

3.2.2 Null 35

3.2.3 Model Checking 37

4 Spielgraphen **39**

4.1 Synthese von Formel und Modell 42

4.2 Gewinnbedingungen 44

4.3 Vom Spielergebnis zum Model Checking Ergebnis 44

4.3.1 Besonderheiten für $\exists\mu$ 47

5 Pläne 49

5.1 Property Oriented Expansion 50

5.1.1 Beispiel 1 – Eingabenvalidierung von Funktionen 51

5.1.2 Beispiel 2 – Newtonverfahren zur Fixpunktberechnung . 54

5.1.3 Beispiel 3 – Spielgraphgenerierung auf Mμ 55

5.1.4 Beispiel 4 – POE 59

5.1.5 Einbettung in einen applikativen Kontext 64

5.2 Pläne 66

5.2.1 Eine Taxonomie für Pläne 66

5.2.2 Graphexploration mittels POE 70

5.2.3 Eigenschaften von POE und seinen Transformern 72

5.3 Beispielanwendung – Model Checking auf $\exists\mu$ 75

5.4 Beispielanwendung – Toolkombinationen 79

6 Implementierung 85

6.1 POE 85

6.1.1 Imperativ 86

6.1.2 Deklarativ/Applikativ 89

6.2 Integration in das Java ABC 93

7 Ausblick 97

8 Fazit 99

A Implementierungsdetails 103

A.1 POE 103

A.2 Transformer 105

A.3 Teilformelbestimmung 107

A.4 Bindungsbestimmung für Fixpunktvariablen 108

B Beweise 111

B.1 Wohldefiniertheit von POE 111

B.2 Terminierungsnachweis von $f_{X^1_G}$ 113

Kapitel 1

Einleitung und Motivation

Formale Verifikation von Hard- und Softwaresystemen, insbesondere im Kontext von Lösungen in sicherheitskritischen Umfeldern, hat in den letzten Jahrzehnten zunehmend Bedeutung erlangt. Eine Möglichkeit, dieser neuen Herausforderung an Softwaresysteme gerecht zu werden, ist das Model Checking.

Hierbei ist es dem Entwickler nun möglich noch während des Designs, während der Implementierung, kurz während des gesamten Entwicklungsprozesses, Kriterien an die zu erstellende Software zu stellen, die im Rahmen eines Tools (des Model Checkers) berücksichtigt und überprüft werden. Der Funktionsumfang und die Korrektheit der erstellten Lösung sind somit nicht nurmehr einem komplexen Beweisverfahren oder dem intuitiven Vorgehen des Entwicklers zuzuschreiben, sondern sind im Rahmen einer formalen temporalen Sprache (einer temporalen Logik) spezifizier- und beweisbar.

Die Integration eines Model Checkers in den Software-Entwicklungsprozess birgt somit das immense Potential, Fehler zu vermeiden, noch bevor sie tiefgreifende Auswirkungen auf den weiteren Verlauf der Entwicklung haben konnten, die Entwicklung damit schneller zu betreiben und eine mit den Vorstellungen der Funktionalität einer Anwendung konsistentere Sicht zu ermöglichen.

Insbesondere im Bereich der Interpretation der Ergebnisse stießen die verwendeten Model Checker jedoch schnell an ihre Grenzen, wenn die zu verifizierenden Eigenschaften des Ablaufverhaltens der zugrundeliegenden modellierten Anwendung den gewünschten Spezifikationen nicht genügten, das gelieferte Ergebnis des Model Checkers jedoch nur unzureichend Hilfestellung zur Anpassung des Modelles oder der Eigenschaft gab. Aus diesem Umstand erwuchs somit schnell der Bedarf für umfassendere Interpretationsspielräume des gelieferten Ergebnisses. Das Ergebnis als schlichte Ja/Nein-Antwort reichte bei vielen, insbesondere praxisnahen, und damit großen, Beispielszenarien nicht mehr aus.

Somit etablierten sich interaktive Ansätze, sich dieser Einschränkung zu bemächtigen. Erste fruchtbare Kandidaten dieser Art fassten Model Checking als einen interaktiven Prozess auf, in dem das zu prüfende System und seine Spezifikation in eine gemeinsame, beide Aspekte berücksichtigende Struktur, synthetisiert

wurden. Diese neue Verschmelzung ermöglicht es dem Entwickler, ein im Sinne der Spezifikation nicht erwünschte Modellierung besser zu erkennen, indem nicht nur eine Ja/Nein-Aussage bzgl. der Erfüllbarkeit einer Eigenschaft für eine konkrete modellierte Probleminstanz getätigt wird, sondern ebenso eine intuitive Anleitung zur Anpassung von Modellierung oder Eigenschaft vorgeschlagen werden kann.

Ein erster Ansatz derartiger qualifizierter Aussagen ist es, nicht nur *einen* Kandidat für die Erfüllbarkeit oder Nicht-Erfüllbarkeit einer Eigenschaft anzugeben, sondern dem Anwender gleich eine ganze Fülle solcher Kandidaten zu offenbaren. In vielen Fällen hilft es nicht, wenn die Modellierung eine Eigenschaft verletzt, ein zugehöriges Gegenbeispiel jedoch zu groß gewählt wird, als dass es dem Anwender beim Auffinden des Fehlers hilfreich sein könnte. Die Möglichkeit der Wahl geeigneter Gegenbeispiele oder auch belegenden Kandidaten soll in dieser Arbeit betrachtet werden. Hierzu wird das Model Checking Problem auf einen Ansatz reduziert, der es als Paritätsspiel auffasst. Ein daraus resultierender (so genannter) Spielgraph wird anschließend als Kontrollflussstruktur interpretiert und die Lösungsmenge dem Nutzer präsentiert. Hiermit ist es dem Anwender nun möglich, gezielt nach Eigenschaften zu suchen, die der Intention der Modellbildung am nächsten liegen und dahingehend die Modellierung anzupassen.

Kapitel 2

Model Checking

Das folgende Kapitel liefert eine kleine Übersicht über den Themenkomplex Model Checking. Der Begriff selbst setzt sich aus den beiden Themenkomplexen der *Modelle* und der in diesen zu überprüfenden (meist temporalen) *Eigenschaften* zusammen. Diese beiden Begriffe spielen im Laufe der gesamten Arbeit, wie auch im Kontext des Model Checking selbst eine zentrale Rolle und werden in diesem Kapitel formal eingeführt. Weniger formale Beschreibungen werden in dem sich anschließenden Kapitel 3 vorgestellt und behandelt.

Formal können wir Model Checking als Frage danach auffassen, ob eine gegebene Struktur M Modell einer Formel ϕ ist. Ob also gilt

$$M \models \phi$$

Hierbei ist M gewöhnlich eine endliche Struktur, die an Graphen erinnert. Insbesondere ist die genaue Ausgestaltung sowohl der zu untersuchenden Struktur, wie auch der auf ihr geltenden Eigenschaft nicht näher festgelegt. Lediglich der Umstand, dass die modellierte Struktur M für gewöhnlich einer imperativen und prozeduralen Beschreibung eines zustandsbasierten Systems und die Eigenschaft ϕ einem funktionalen, zustandslosen und auswertungsorientierten Ausdruck zugrundeliegen, lässt sich als invarianter Kern dieses Verfahrens beschreiben.

Dabei ist jedoch zu beachten, dass die Modellierung der Struktur für den Model Checker immer eine Abstrahierung ist, die die Gefahr birgt, gewisse Aspekte des betrachteten Problems auszusparen oder neue Aspekte in die Modellierung aufzunehmen, die nicht das zu untersuchende Objekt widerspiegeln. Wenn Model Checking also eine Aussage betrifft, ob eine Eigenschaft gilt oder nicht, hilft ein Gegenbeispiel (falls die Eigenschaft nicht gilt) bzw. ein Beweis dafür (falls die Eigenschaft gilt), diese Aussage auf das konkrete Problem zu übertragen und die Aussage des Model Checkers wert zu schätzen. Wichtig in diesem Zusammenhang ist also, sich als Anwender des Verfahrens immer über die zugrunde liegende Abstraktion des untersuchten Systems in eine Modellstruktur und die Abstraktion der gewünschten Eigenschaft (der Spezifikation) in eine (temporal-

logische) Formel bewusst zu sein und das Ergebnis des Model Checking Prozesses in diesem Kontext zu werten.

Die nächsten beiden Abschnitte etablieren die Begriffe *Modell* und *Eigenschaft* im Kontext des klassischen Model Checkings in formaler Art und Weise. Hierbei werden zunächst die klassischen Vorstellungen von Modellen als Transitionsgraphen mit (ggf. vorhandenen) Annotationen an Kanten und Knoten, wie auch sowohl lineare als auch verzweigende temporale Logiken vorgestellt.

2.1 Kripke-Strukturen und Transitionssyteme

Bisher wurden noch keinerlei Aussagen dazu getroffen, wie die Strukturen, für die Eigenschaften nachgewiesen werden, genau aussehen sollen. Hier existiert auch keine einheitliche Definition – was auch nicht weiter verwundert, will man ein breites Spektrum von zu modellierenden Problemstellungen berücksichtigen – als vielmehr verschiedene Modelle, die abhängig von der betrachteten Problemstellung verwendet werden können und jeweils ihre eigenen Vor- und Nachteile bergen.

Im Folgenden stelle ich drei dieser Vertreter genauer vor.

- Kripke-Strukturen (im Folgenden kurz KS)
- Labeled Transition Systems (im Folgenden kurz LTS)
- Kripke-Transitionssysteme (im Folgenden kurz KTS)

Intuitiv kann man sich darunter graphartige Gebilde vorstellen, bei denen jeweils die Knoten (bei Kripke-Strukturen), die Kanten (bei Labeled Transition-Systems) oder die Knoten *und* die Kanten (bei Kripke-Transitions-Systemen – siehe [MOSS99]) beschriftet sind.

Alle drei Vertreter sind diskrete Modelle, die von einem diskreten endlichen Zustandsraum ausgehen und auf diesem automatenartige Transitionen vollziehen. Denkbar sind zudem nicht-diskrete (siehe [Alu99] und [Hen94]) oder unendliche (siehe [SB92]) Modelle, die hier jedoch nicht weiter betrachtet werden.

2.1.1 Kripke-Strukturen

Eine *Kripke-Struktur* (kurz KS) über einer Menge atomarer Aussagen (atomarer Propositionen) AP ist ein Tripel (S, R, I) mit den Eigenschaften:

- S ist eine Menge von Zuständen
- R ist eine Relation $R \subseteq S \times S$, die wir *Transitionsrelation* nennen
- I ist eine Abbildung $I : S \mapsto 2^{AP}$, die wir *Interpretation* nennen.

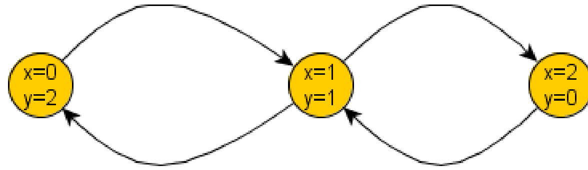


Abbildung 2.1: Beispiel einer Kripke Struktur.

Hierbei repräsentieren die atomaren Propositionen Symbole, die wiederum atomare Aussagen widerspiegeln, die an den Zuständen gelten können. Die Abbildung I weist jedem Zustand eine Menge solcher atomarer Propositionen zu, die in diesem Zustand gelten. Weiter wird angenommen, dass in der Menge AP der atomaren Propositionen die Aussagen **wahr** und **falsch** enthalten sind[1]. Ferner soll gelten

$$\forall s \in S : \textbf{wahr} \in I(s) \text{ und } \textbf{falsch} \notin I(s).$$

Eine *Kripke-Struktur* heißt *total*, wenn ihre Transitionsrelation R total ist. Wenn also gilt

$$\forall s \in S : \exists t \in S : (s,t) \in R.$$

Wir nennen eine *Kripke-Struktur partiell,* wenn sie nicht *total* ist.

In unserer Betrachtung sind sowohl S als auch AP endlich. Ein Beispiel für eine *Kripke-Struktur* ist in Abbildung 2.1 zu sehen. Diese beschreibt eine abstrakte Darstellung eines Kontrollflussgraphen, wobei der Zustand, in Form einer Variablenbelegung zweier Variablen x und y, beschrieben wird und die Kanten die Zustandsübergänge durch Änderung dieser Variablenbelegungen beschreiben.

2.1.2 Labeled Transition Systems

Wie der Name schon vermuten lässt, sind *Labeled Transition Systems* (kurz LTS) an den Kanten beschriftete Graphen. Formal ist ein solches System als Tripel $(S, Act, \rightarrow)$ gegeben, wobei die folgenden Eigenschaften gelten.

- S ist eine endliche Menge von Zuständen
- *Act* ist eine Menge von *Aktionen*
- $\rightarrow$ ist eine *Transitionsrelation* $\rightarrow \subseteq S \times Act \times S$.

[1]Dies wird später bei der Definition einiger Operatoren aus den Temporallogiken von Nutzen sein.

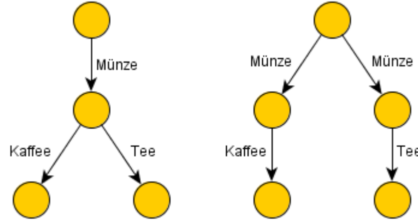


Abbildung 2.2: Ein Modell eines Kaffee- und Tee-Automaten.

Für $\rightarrow$ existiert die folgende einfache Schreibweise: Gilt $(s, a, s') \in \rightarrow$, so schreibt man auch $s \xrightarrow{a} s'$. Verstehen lässt sich diese Notation derart, dass aus einem Zustand s in den Folgezustand s' übergegangen werden kann, wobei die Aktion a mit der Umgebung interagiert bzw. in ihr ausgeführt wird. In einem solchen Fall nennen wir s' einen *a-Nachfolger* von s.

Wichtig ist hierbei, dass an den Kanten jeweils nur eine Aktion stehen kann. Aus einem Zustand kommen wir also nur über *genau eine* Aktion – anstatt über eine *Menge* von Aktionen – in den Folgezustand[2].

Ein Beispiel aus dem Alltag für ein LTS modelliert einen Automaten, der nach Münzeinwurf sowohl Kaffee als auch Tee ausgeben kann. Zwei Varianten der Modellierung eines solchen Kaffee-Tee-Automaten zeigt Abbildung 2.2.

Die beiden Automaten unterscheiden sich hinsichtlich bestimmter Eigenschaften, wenn man Anforderungen an das Modell stellt, die einerseits durch eine Linearzeitlogik und andererseits durch eine Branching-Time-Logik beschrieben werden. Bei beiden Maschinen ergeben sich als maximale Berechnungspfade die Mengen {Münze, Kaffee} und {Münze, Tee}. Eine Linearzeitlogik kann also nicht zwischen den beiden Modellierungen unterscheiden. Eine Branching-Time-Logik dagegen kann jedoch sehr wohl entscheiden, ob in jedem Fall nach einer Münze-Aktion eine Kaffee-Aktion folgt, was wir in einem späteren Abschnitt noch sehen werden.

2.1.3 Kripke-Transitionssysteme

Kripke-Transitionssysteme (kurz KTS) vereinen schließlich die Eigenschaften von Kripkestrukturen und Labeled-Transitionssystemen. Hier sind nun Beschriftungen sowohl an den Kanten als auch an den Knoten zulässig.

Formal lässt sich ein *Kripke-Transitionssystem* T über einer Menge atomarer Propositionen AP als ein Tupel $T = (S, Act, \rightarrow, I)$ mit den folgenden Eigenschaften beschreiben.

- S ist eine endliche Menge von *Zuständen*.

[2]Bemerke: Bei Kripke-Strukturen, waren wir in der Lage, an den Knoten Mengen von Eigenschaften zu notieren.

- *Act* ist eine Menge möglicher *Aktionen*.
- $\rightarrow$ ist eine *Transitionsrelation* $\rightarrow \subseteq: S \times Act \times S$.
- I ist eine *Interpretation* $I : S \mapsto 2^{AP}$.

In vielen Anwendungen werden die Mengen *Act* und *AP* disjunkt angenommen.

Die Transitionsrelation von Kripke-Transitionssystem und Labeled-Transitionssystemen gleichen sich stark. Neu hinzugekommen ist lediglich eine Interpretation I, die jedem Zustand eine Menge atomarer Propositionen zuordnet, die in ihm gilt.

Kripke-Transitionssysteme sind Verallgemeinerungen der beiden vorherigen Vertreter. Bei einer Kripke-Struktur ist die Menge der Aktionen *Act* leer, bei einem Labeled-Transition-System ist die Interpretation I, die triviale Interpretation.

Anwendung finden Kripke-Transitions-Systeme etwa bei der Modellierung imperativer Programme für die Datenflussanalyse. An den Knoten werden in diesem Anwendungskontext Ergebnisse der Datenflussanalyse notiert, Kanten hingegen tragen Informationen über den Programmablauf. Ist etwa das folgende Programm gegeben

```
z = 0;
i = 0;
while (i != y)
    z = z+x;
    i = i+1;
```

und der Untersuchungsgegenstand des Programms betrifft Lebendigkeitseigenschaften von Variablen, so lässt sich daraus das Kripke-Transitions-System aus Abbildung 2.3 erzeugen. Eine in diesem Zusammenhang interessante Anwendung von Model Checking für Datenfluss-Analyse-Probleme ist etwa in [Ste91] zu finden. Hierbei lassen sich die Eigenschaften von Programmstellen durch temporal-logische Formeln auf dem Kontrollflussgraphen beschreiben.

Manchmal wollen wir einen Zustand als den Ausgangspunkt eines Labeled-Transition-Systems, einer Kripke-Struktur oder eines Kripke-Transitions-Systems auffassen. Dann bezeichnen wir diesen ausgezeichneten Zustand mit s_0.

2.2 Temporale Logiken

Lag das bisherige Augenmerk auf der Modellierung der Systeme durch Transitionsgraphen, für die Eigenschaften nachzuweisen waren, liefern die Temporallogiken die geeignete Beschreibungssprache für die Eigenschaften, die wir von den spezifizierten Modellen erwarten. Die formale Beschreibungssprache in Form

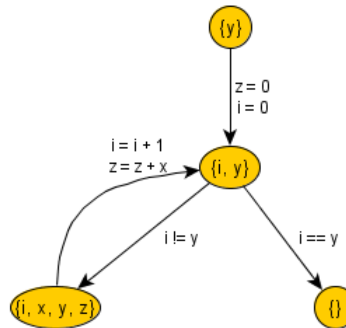


Abbildung 2.3: Beispiel eines Kripke Transitions Systems.

von zeitlichen Logiken, die in der Lage sind, Aussagen über und Veränderungen innerhalb dieses zeitlichen Ablaufes zu treffen, werden nun in diesem Abschnitt vorgestellt und vervollständigen damit das Gesamtbild des Model Checking Kontextes.

Propositionale Logiken machen Aussagen, die sich über die Zeit nicht verändern. Sie sind wahr oder falsch für alle Zeiten. Bei den Kripke-Transitions-Systemen wurden bereits in den Zuständen eine Menge von Knotenannotationen eingeführt, die eine Aussage darüber machen, welche Aussagen in einem Zustand wahr sind und welche falsch. Nun kann es aber auch von Interesse sein, ob ein solcher Zustand überhaupt erreicht wird. Hat z.B. in einem Kripke-Transitions-System ein Knoten die Interpretation *rot* erhalten, so ist sicherlich von Interesse, ob bzw. von welchen Knoten aus dieser rote Knoten erreichbar ist. Oder aber es könnte von Interesse sein, ob auf einem Pfad, der das Kripke-Transitions-System durchläuft, immer ab einer bestimmten Stelle alle nachfolgend besuchten Knoten rot sind.

Doch wie werden solche Eigenschaften in einer formalen Sprache ausgedrückt? Genau dies wird in den beiden folgenden Abschnitten beispielhaft durch einige Kandidaten temporaler Logiken darstellt. Dabei werden zunächst Linearzeitlogiken vorgestellt, die nur für einzelne Berechnungspfade Entscheidungen treffen, ob eine Eigenschaft gilt, nicht jedoch formulieren, ob eine Eigenschaft für *alle* Berechnungspfade gilt bzw., ob es *einen* Berechnungspfad gibt, für den eine gewissen Eigenschaft gilt. Dies ermöglichen erst die Branching-Time-Logiken. Hierbei wird dann auch die Unterscheidung des auf Seite 14 vorgestellten Kaffee-Tee-Automaten möglich.

2.2.1 Linearzeitlogiken

Ein prototypischer Vertreter von Linearzeitlogiken ist die *propositional linear-time logic* (kurz PLTL, manchmal auch LTL oder PTL genannt)¸ die häufig über die eben vorgestellten Kripke-Strukturen definiert wird. Wenn p eine atomare Proposition aus AP ist, so lässt sich eine PLTL-Formel ϕ wie folgt erzeugen:

$$
\begin{aligned}
\pi \models p \quad &\Leftrightarrow \quad p \in I(\pi_0) && (2.1)\\
\pi \models \neg\phi \quad &\Leftrightarrow \quad \pi \not\models \phi && (2.2)\\
\pi \models \phi_1 \vee \phi_2 \quad &\Leftrightarrow \quad \pi \models \phi_1 \text{ oder } \pi \models \pi_2 && (2.3)\\
\pi \models X(\phi) \quad &\Leftrightarrow \quad |\pi| > 1 \text{ und } \pi^1 \models \phi && (2.4)\\
\pi \models U(\phi,\psi) \quad &\Leftrightarrow \quad \exists k, 0 \leq k \ < |\pi| : \pi^k \models \psi && (2.5)\\
&\qquad \text{und} \\
&\qquad \forall i, 0 \leq i < k : \pi^i \models \phi \\
\pi \models F(\phi) \quad &\Leftrightarrow \quad \exists k, 0 \leq k < |\pi| : \pi^k \models \phi && (2.6)\\
\pi \models G(\phi) \quad &\Leftrightarrow \quad \forall k, 0 \leq k < |\pi| : \pi^k \models \phi && (2.7)
\end{aligned}
$$

Abbildung 2.4: Die Semantik einer Linearzeitlogik

$$\phi ::= p|\neg\phi|\phi_1 \vee \phi_2|X(\phi)|U(\phi,\psi)|F(\phi)|G(\phi)$$

Hierbei werden diese Formeln über *Pfade* in Kripke-Strukturen interpretiert. Diese kann man sich als eine Folge von Zuständen vorstellen, die die Kripke-Struktur gemäß ihrer Transitionsrelation durchläuft. Eine solche *endliche* Folge $\pi = \langle \pi_0, \pi_1, \ldots, \pi_{n-1}\rangle$ von Zuständen, für die gilt $\pi_0, \pi_1, \ldots, \pi_{n-1} \in S, \forall 0 \leq i \leq n-1 : (\pi_i, \pi_{i+1}) \in R$ bezeichnen wir als einen *Pfad*. Einen *unendlichen* Pfad bezeichnen dann jene Folgen von Zuständen, für die gilt $\forall i \geq 0 : (\pi_i, \pi_{i+1}) \in R$. Bei solchen Pfaden bezeichnet

- π_i den i-ten Zustand der entsprechenden Zustandsfolge und
- π^i mit $\langle \pi_{i+1}, \pi_{i+2}, \ldots\rangle$ wird als *Rest* bezeichnet.

Durch diese Definitionen lässt sich jetzt bereits die Semantik der beschriebenen Logik angeben, die in Abbildung 2.4 zu sehen ist.

Die formale Semantikbeschreibung soll nun durch ein paar klärende Worte mit etwas Leben gefüllt und eine Vorstellung für die Benutzung gewonnen werden.

1. Eine Zustandsfolge modelliert eine atomare Proposition genau dann, wenn diese Proposition bereits im ersten Zustand gilt.
2. Diese Definition ergibt sich aus der intuitiven Vorstellung des Begriffes *Nicht*.
3. Auch diese Definition folgt der intuitiven Vorstellung des Begriffes *oder*.
4. $X(\phi)$ (genannt *neXt*) verlangt, dass die Eigenschaft ϕ auch für den Rest gelten soll, wenn der erste Zustand der Zustandsfolge entfernt wird. Folglich gilt $X(\pi)$ – gesehen von der aktuellen Position aus – immer für den aktuellen Folgezustand. Da wir eine Logik mit Linearzeit betrachten, ist dieser auch eindeutig. Da unsere Zustandsfolge mindestens die Länge 1 haben muss, ist dieser auch vorhanden.

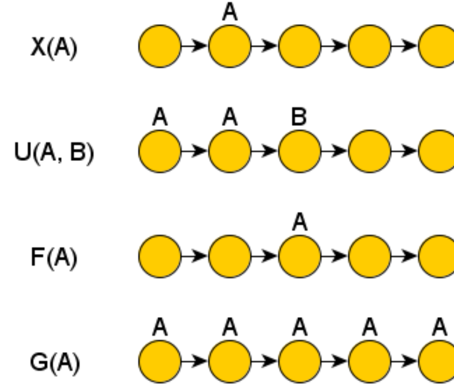


Abbildung 2.5: Ein Anschauung für Modalitäten.

5. $U(\phi, \psi)$ (genannt *Until*) verlangt intuitiv, dass die Eigenschaft ϕ solange gilt, bis irgendwann ψ gilt[3]. Insbesondere erfolgt keine Aussage über die Dauer der Gültigkeit von ψ. Es ist also ausreichend, wenn ψ in lediglich einem Modellknoten gilt.

6. $F(\phi)$ (genannt *Final*) verlangt, dass für irgendeinen Zustand der Zustandsfolge die Eigenschaft ϕ eintreten muss.

7. $G(\phi)$ (genannt *Global*) verlangt, dass für alle Zustände der betrachteten Zustandsfolge die Eigenschaft ϕ gelten muss.

Abbildung 2.5 zeigt eine graphische Vorstellung der Formeln und der obigen Beschreibung der verschiedenen Modalitäten X, U, F und G.

Die vorgestellte Definition ist keinesfalls minimal. Insbesondere reichen die beiden Modalitäten X und U, um die andere damit auszudrücken.

$$F(\phi) = U(\text{wahr}\phi)$$
$$G(\phi) = \neg F(\neg\phi)$$

Dennoch ist die vorgestellte Variante weitaus leichter verständlich und eingängiger.

[3]Diese Variante wird auch *strong-until* genannt, da ψ tatsächlich irgendwann eintreten muss. Gilt dagegen die ganze Zeit ϕ und niemals ψ, so kann eine Variante *weak-until* etabliert werden, die in diesem Falle wahr wäre. Diese Variante lässt sich leicht durch $U_W(\phi, \psi) = U(\phi, \psi) \vee G(\phi)$ definieren.

$$
\begin{array}{rcl}
s \models \text{wahr} & & s \not\models \text{falsch} \\
s \models \phi_1 \wedge \phi_2 & \Leftrightarrow & s \models \phi_1 \text{ und } s \models \phi_2 \\
s \models \phi_1 \vee \phi_2 & \Leftrightarrow & s \models \phi_1 \text{ oder } s \models \phi_2 \\
s \models [a]\,\phi & \Leftrightarrow & \forall t \text{ mit } s \xrightarrow{a} t : t \models \phi \\
s \models \langle a \rangle\,\phi & \Leftrightarrow & \exists t \text{ mit } s \xrightarrow{a} t : t \models \phi
\end{array}
$$

Abbildung 2.6: Semantik von HML

2.2.2 Branching-Time Logik

Eine kanonische Verallgemeinerung der Linearzeitlogiken sind die so genannten *Branching-Time-Logiken*, die auch Aussagen über verzweigte Berechnungspfade machen können und damit eine echte Erweiterung der Linearzeitlogiken darstellen. Sie werden im Folgenden durch ihre Vertreter Hennessy-Milner-Logik (kurz HML), die computation-tree-logic (kurz CTL bzw CTL^* für ihre Erweiterung) und schließlich den modalen μ-Kalkül (oder kurz auch μ-TL) vorgestellt und durch ein paar Beispiele mit Leben gefüllt. Schließlich wird noch eine für den Rest der Arbeit wesentliche neue Logik vorgestellt, die auf der Reduktion des modalen μ-Kalküls basiert.

Hennessy-Milner-Logik (HML)

Die erste Logik ist eine recht einfach zu verstehende, aber, verglichen mit den anderen noch folgenden Vertretern, nicht sonderlich ausdrucksstarke Logik und nennt sich Hennessy-Milner-Logik (kurz HML). Sie ist auf bestimmte Aussagenklassen beschränkt. Eine Erweiterung dieser Logik um Fixpunkte beschreibt den modalen μ-Kalkül, der im Anschluss vorgestellt wird und die stärkste Aussagekraft der vorgestellten Logiken besitzt.

Doch zunächst zu HML. Die Logik ist definiert über einer Menge *Act* von Aktionen, die von einer Variablen a durchlaufen werden. Dann lässt sich die Syntax einer HML-Formel ϕ wie folgt beschreiben.

$$\phi ::= \text{wahr} | \text{falsch} | \phi_1 \vee \phi_2 | \phi_1 \wedge \phi_2 | \, [a]\,\phi | \, \langle a \rangle\,\phi$$

Neu hinzugekommen sind die beiden Modalitäten $[]$ und $\langle\rangle$, die aufgrund ihrer Form auch als *Boxoperator* und als *Diamondoperator* bezeichnet werden.

Die Logik wird über LTS interpretiert. Bei einem gegebenen LTS

$$T = (S, Act, \rightarrow)$$

wird induktiv definiert, wann eine HML-Formel ϕ für einen Zustand $s \in S$ gültig ist ($s \models_T \phi$). Die Semantik einer solchen Formel lässt sich wie in Abbildung 2.6 beschreiben.

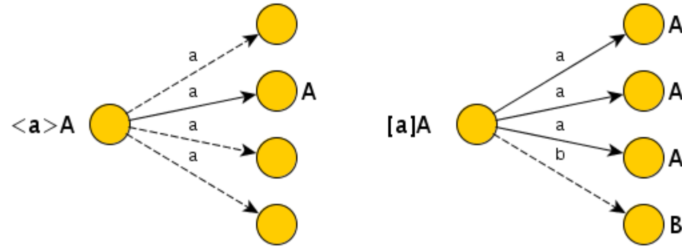


Abbildung 2.7: Anschauung für die Operatoren $\Box$ und $\diamond$

Alle Definitionen folgen unseren Überlegungen, die auch schon bei der PLTL-Logik galten. Lediglich neu hinzugekommen sind der Box- und der Diamond-operator. Hierbei beschreibt der Boxoperator intuitiv das Verhalten, dass im Zustand s für *alle* a-Nachfolger von s immer noch die Formel ϕ gelten muss. Beim Diamondoperator dagegen wird lediglich verlangt, dass es *mindestens einen* a-Nachfolger gibt, für den die Formel gelten muss. Insbesondere ist es auch nicht so, dass fehlende Nachfolger diesen Operator falsifizieren. Analog zu all-quantifizierten Aussagen, erfüllen Knoten ohne Nachfolger jedwede Eigenschaft, die über einen Box-Operator spezifiziert ist. Eine graphische Vorstellung der Operatoren liefert Abbildung 2.7.

Computation Tree Logic

Kommen wir nun zu einer Logik, die eine weitere Erweiterung der bisher bekannten Logiken darstellt. PLTL wird häufig zur Überprüfung von *Korrektheitseigenschaften* von Systemen eingesetzt. Die Frage danach, ob bei einer bestimmten Ausführung eine bestimmte Eigenschaft eintritt, lässt sich jedoch nicht mehr mit/in PLTL ausdrücken. Solche Eigenschaften sind dagegen gut in CTL (computational tree logic) beschreibbar. Die in ihr beschriebenen Formeln sind sehr anschaulich und ermöglichen uns eine große Menge von zeitlogischen Aussagen. Doch wie sehen solche CTL-Formeln aus?

Ist p eine atomare Proposition, dann lässt sich die Syntax von CTL-Formeln wie folgt beschreiben.

$$\phi ::= \phi_1 \vee \phi_2 | \phi_1 \wedge \phi_2 | EX\phi | EG\phi | \phi_1 EU \phi_2$$

Die Operatoren werden in der Praxis jedoch noch häufig um die folgenden Operatoren ergänzt.

$$
\begin{aligned}
T, s \models v \text{ für } v \in V &\Leftrightarrow v \in s(V) \\
T, s \models EX\phi &\Leftrightarrow \exists\pi : \pi_0 = s \text{ und } T, \pi_1 \models \phi \\
T, s \models EG\phi &\Leftrightarrow \exists\pi : \pi_0 = 2 \text{ und } \forall k \geq 0 : T, \pi_k \models \phi \\
T, s \models \phi_1 EU \phi_2 &\Leftrightarrow \exists\pi \text{ mit } \pi_0 = s \text{ und } k \geq 0 : \\
& \quad\; T, \pi_k \models \phi_2 \text{ und} \\
& \quad\; \forall 0 \leq j < k : T, \pi_j \models \phi_1
\end{aligned}
$$

Abbildung 2.8: Semantik von CTL

$$
\begin{aligned}
EF\phi &= \text{wahr } EU\phi \\
AX\phi &= \neg EX \neg\phi \\
AG\phi &= \neg EF \neg\phi \\
AF\phi &= \neg EG \neg\phi \\
\phi_1 AU \phi_2 &= \neg(\neg\phi_2 EU \neg\phi_1 \wedge \neg\phi_2) \wedge AF(\phi_2)
\end{aligned}
$$

Wie PLTL-Formeln, werden die aus der obigen Vorschrift gewonnenen CTL-Formeln über Kripke-Strukturen interpretiert. Ist eine Kripke-Struktur T und ein Zustand $s \in S$ gegeben, dann ist die Semantik für $T, s \models \phi$ wie in Abbildung 2.8 induktiv definiert, wobei π hier ein Berechnungspfad ist.

Die Semantik für die boolschen Verknüpfungen ist wie gewöhnlich definiert.

Hierbei beschreibt $EX\phi$, dass es *einen* Pfad gibt, auf dem ϕ im nächsten Schritt gilt, $AX\phi$ dagegen verlangt diese Gültigkeit im nächsten Schritt für *alle* solchen Nachfolger. $EF\phi$ verlangt mindestens einen Pfad, auf dem schließlich irgendwann ϕ gilt, $AF\phi$ verlangt dies dagegen für alle Pfade. $\phi_1 EU \phi_2$ verlangt, dass es einen Pfad gibt, auf dem ϕ_1 so lange gilt bis ϕ_2 irgendwann gilt, $\phi_1 AU \phi_2$ verlangt dies analog wieder für alle Pfade. Schließlich soll bei $EG\phi$ eine globale Gültigkeit von ϕ für einen Pfad herrschen, $AG\phi$ dagegen fordert gar eine globale Gültigkeit auf allen Pfaden und damit im gesamten Modell.

A, E bezeichnet man demnach als *Pfadquantoren*, da diese Aussagen machen, auf welchen Pfaden die Formel gelten soll, während G, F, U, X als *temporale Operatoren* bezeichnet werden.

Das Zusammenspiel von Pfadquantoren und temporalen Operatoren ist in Abbildung 2.9 zu sehen. Die Spitze des beschnittenen Kegels symbolisiert hierbei einen möglichen Startpunkt für Modellexplorationen entlang eines oder mehrerer Pfade, wobei die Farben[4] jeweils der Gültigkeit der untersuchten Eigenschaft entsprechen. So fordert AG etwa die Gültigkeit der Eigenschaft auf allen Pfaden. AF fordert die Gültigkeit auf allen Pfaden in jeweils mindestens einer Stelle. Die existenziellen Operatoren EF und AF schließlich fordern die Gültigkeit für nur mindestens einen der von der Spitze des Kegels startenden Pfade. Die Until-Operatoren EU und AU besitzen zwei Argumente und ermöglichen die

[4]In einer nicht-farbigen Version dieser Arbeit ist das rot in einer dunkleren, das blau dagegen in einer helleren Grauschattierung zu erkennen.

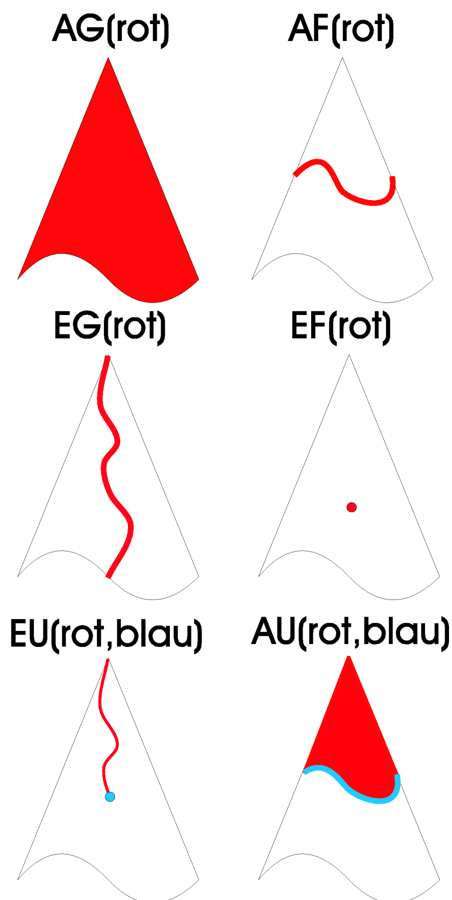


Abbildung 2.9: Eine anschauliche Vorstellung der CTL-Operatoren.

Beschränkung des ersten Arguments entlang eines oder aller Pfade durch das zweite Argument.

Zwei Beispiele für die Verwendung von CTL-Formeln sollen die Handhabung verdeutlichen und weiter veranschaulichen, in welchen Bereichen diese Operatoren Verwendung finden:

- $AG\neg(owns_1 \wedge owns_2)$ ist ein Beispiel für mutual-exclusion, bei der ein Ressourcenverwalter, der zwei Prozesse bedient, die gleiche Ressource niemals gleichzeitig zwei verschiedenen Anforderern zur Verfügung stellen darf.
- $AG(EF(init))$ besagt für ein Prädikat $init$, das den Startzustand eines Systems beschreibt, dass es von jedem Punkt aus einen Weg wieder zurück in eben diesen Startzustand gibt, dass das System also zurücksetzbar ist.

CTL kränkelt ein wenig in der Ausdruckskraft, da sich Pfadquantoren (A, E) und temporale Operatoren (X, U, G, F) in CTL-Formeln abwechseln müssen. Eine Erweiterung von CTL, genannt CTL*, umgeht diese Einschränkung und

lässt nun Formeln wie etwa $AFG(p)$ zu. Damit wird CTL* echt mächtiger und gewinnt somit an Ausdruckskraft – es lässt sich zeigen, dass CTL* eine Obermenge von CTL und PLTL ist (vgl. hierzu [CD89], [EH86] und [Lam80] sowie Abbildung 2.11 auf Seite 25).

Eine Erweiterung von CTL* schließlich, die sich jedoch nicht auf die Menge der verwendeten Operatoren, als vielmehr auf die Ausdruckskraft bezieht, ist der modale μ-Kalkül, der im nächsten Abschnitt vorgestellt wird.

Der modale μ-Kalkül (μ-TL)

Der modale μ-Kalkül erweitert die bereits vorgestellte Hennessy-Milner-Logik um Fixpunktoperatoren. Die Syntax von μ-Kalkül-Formeln ergibt sich aus der folgenden Beschreibung.

$$\begin{array}{rccccc} \phi ::= & \text{wahr} & \mid & \text{falsch} & \mid & \\ & [a]\,\phi & \mid & \langle a\rangle\,\phi & \mid & \\ & \phi_1 \vee \phi_2 & \mid & \phi_1 \wedge \phi_2 & \mid & \\ & \mu X.\phi & \mid & \nu X.\phi & \mid & X \end{array}$$

Hierbei läuft a über die Menge der Aktionen Act und X ist eine Variable aus Var, der Menge der Fixpunktvariablen. Die beiden Fixpunktoperatoren sind μ/ν und bezeichnen jeweils den kleinsten/größten Fixpunkt.

Die Grammatik lässt keine Negationen zu und sicherlich ist dies nicht in allen Fällen wünschenswert. Dennoch ist dies keine echte Einschränkung, da sich die Negationen bis auf die Ebene der atomaren Propositionen durch die Regel von de Morgan und die Äquivalenzen

$$\begin{aligned} \neg\langle a\rangle\,\phi &= [a]\,\neg\phi \\ \neg(\mu X.\phi) &= \nu X.\neg\phi[\neg X/X] \end{aligned}$$

umformen lassen[5].

Formeln, die entsprechend der oben angegebenen Grammatik konstruiert werden, werden über ein LTS $T = (S, Act, \rightarrow)$ interpretiert, dessen Zustände aus S die geschlossene Formel ϕ wahr machen. Um die Bedeutung von nicht-geschlossenen Formeln zu erklären, führen wir nun *Umgebungen* ein. Dies sind partielle Abbildungen $\rho : Var \overset{part}{\mapsto} 2^S$, die die freien Variablen in ϕ als Teilmengen von S interpretieren; dann ist $M_T(\phi)(\rho)$ gerade die Menge von Zuständen des LTS T, die die μ-Kalkül-Formel ϕ unter Berücksichtigung der Umgebung ρ erfüllen. Die Bedeutung einer geschlossenen Formel dagegen ist unabhängig von einer konkreten Umgebung und gilt daher für alle Umgebungen gleichermaßen.

[5]Hierbei können in bestimmten Fällen Probleme beim Entfernen von Negationen auftreten, die wir hier jedoch nicht betrachten wollen, da diese sich aus Monotoniegründen auch nur für Formeln ergeben, in denen ϕ nicht monoton von X abhängt und damit solche Formeln die Wohldefiniertheit der Semantik gefährden.

$$
\begin{array}{ll}
M_T(\text{wahr})(\rho) & = S \\
M_T(\text{falsch})(\rho) & = \emptyset \\
M_T([a]\,\phi)(\rho) & = \{s|\forall s' : \\
 & \quad s \xrightarrow{a} s' \Rightarrow s' \in M_T(\phi)(\rho)\Big\} \\
M_T(\langle a\rangle\,\phi)(\rho) & = \{s|\exists s' : \\
 & \quad s \xrightarrow{a} s' \wedge s' \in M_T(\phi)(\rho)\Big\} \\
M_T(\phi_1 \vee \phi_2)(\rho) & = M_T(\phi_1)(\rho) \cup M_T(\phi_2)(\rho) \\
M_T(\phi_1 \wedge \phi_2)(\rho) & = M_T(\phi_1)(\rho) \cap M_T(\phi_2)(\rho) \\
M_T(X)(\rho) & = \rho(X) \\
M_T(\mu X.\phi)(\rho) & = \text{fix}_\mu F_{\phi,\rho} \\
M_T(\nu X.\phi)(\rho) & = \text{fix}_\nu F_{\phi,\rho} \\
 & \quad \text{mit } F_{\phi,\rho}(x) = M_T(\phi)(\rho[X \to x])
\end{array}
$$

Abbildung 2.10: Semantik des modalen μ-Kalküls

Somit gilt für eine μ-Kalkül-Formel ϕ und einen Zustand $s \in S$: $s \models^T \phi$, wenn $s \in M_T(\phi)(\rho)$ für eine beliebige Umgebung ρ.

Die Semantik von μ-Kalkül Formeln relativ zu einer Umgebung lässt sich somit wie in Abbildung 2.10 beschreiben.

Propositionale Aussagen, wie auch der Box- und der Diamond-Operator haben die gleiche Bedeutung, wie auch schon in HML. Neu hinzugekommen, sind lediglich die Operatoren zur Bestimmung kleinster (μ) und größter (ν) Fixpunkte, wie auch Variablen, über die der Fixpunkt bestimmt werden soll. Zu diesem Zweck wird eine Umgebung ρ genutzt, um über die Belegung der Fixpunktvariablen Buch zu führen. Die genaue Art und Weise der Bestimmung der Fixpunkte mittels fix_μ und fix_ν bleibt hierbei jedoch offen.

Die μ-Kalkül-Formeln beschreiben in ihrer Ausdrucksstärke eine echte Obermenge von CTL* (und somit auch von CTL), lassen sich doch alle CTL-Formeln auch als μ-Kalkül-Formeln darstellen. So gilt etwa (nach [CGP99])

$$
\begin{array}{rcl}
AG(\phi) & = & \nu X.\phi \wedge [\,]\, X \\
EG(\phi) & = & \nu X.\phi \wedge \langle\rangle\, X \\
EF(\phi) & = & \mu X.\phi \vee \langle\rangle\, X \\
AF(\phi) & = & \mu X.\phi \vee [\,]\, X.
\end{array}
$$

Stellen wir abschließend noch einmal alle bisher betrachteten Logiken bzgl. ihrer Ausdruckskraft in einem Venn-Diagramm gegenüber, so ergibt sich das Bild aus Abbildung 2.11.

Zwei Fragmente des μ-Kalküls: $\exists\mu$ und $\forall\nu$

Da der in Abschnitt 2.2.2 vorgestellte modale μ-Kalkül die anderen vorgestellten Logiken in seiner Ausdrucksstärke umfasst, wenngleich Umformungen in

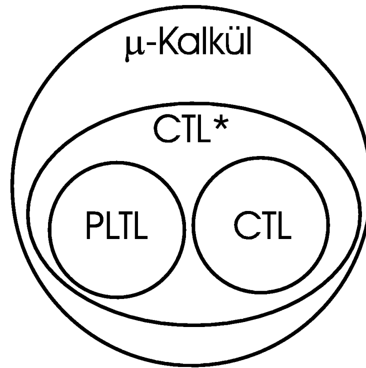


Abbildung 2.11: Ein Vergleich der Ausdrucksstärken verschiedener Temporallogiken.

den modalen μ-Kalkül Formeln unter Umständen exponentiell in ihrer Größe anwachsen lassen, und damit in gewisser Form eine Universalität temporal logischer Eigenschaften etabliert, ist anzunehmen, dass sich weitere stärkere Resultate nur durch Einschränkung dieser Mächtigkeit erzielen lassen. Diese Arbeit beruht grundlegend auf einer Einschränkung des modalen μ-Kalküls und bezieht ihre Ergebnisse gerade aus diesem Verlust der Ausdruckskraft. Andererseits muss die Logik dennoch ausdrucksstark genug bleiben, um weiterhin einen hinreichend großen Fundus von Aussagen bereit zu stellen.

Die Einschränkung, auf die im Folgenden immer wieder Bezug genommen wird, besteht in einer partiellen Partitionierung der Operatoren des μ-Kalküls in *konjunktive* und *disjunktive* Operatoren. Diese Partitionierung ist partiell in dem Sinne, dass sich bestimmte Operatoren beiden Seiten – also sowohl der disjunktiven, wie auch der konjunktiven – zuschreiben lassen. Gemeint sind damit z.B. die atomaren Propositionen, die weder eine eindeutig konjunktive, noch eindeutig disjunktive Gestalt besitzen – ja vielmehr, diese erst im Zusammenhang mit einem zu überprüfenden Modell annehmen.

Diese Partitionierung in disjunktive und konjunktive Operatoren, die wir im Folgenden $\exists\mu$ (für den um die konjunktiven Operatoren reduzierten μ-Kalkül) und $\forall\nu$ (für den um die disjunktiven Operatoren reduzierten μ-Kalkül) nennen möchten, wird im Folgenden kurz vorgestellt.

Wie für die atomaren Propositionen, gibt es auch für die Fixpunktoperatoren natürlicherweise mehrere Möglichkeiten, diese in einem konjunktiven wie auch disjunktiven Kontext zu interpretieren. Diese Doppeldeutigkeit folgt aus der Interpretation der Fixpunktoperatoren jeweils in einem applikativen deklarativen Kontext und einem imperativen Kontext. Was damit gemeint ist, sollen die beiden folgenden bekannten Sätze von Knaster, Tarski und Kleene genauer klären.

Satz 1 (Knaster-Tarski (vgl. [Tar55]) *Sei $f : D \to D$ eine monotone Funktion eines vollständigen Verbandes $(D, \sqsubseteq)$ und P die Menge aller Fixpunkte von f. Dann ist P nicht leer und es gilt*

$$\bigcap P = \bigcap \{x \in D | f(x) \sqsubseteq x\}\,.$$

Insbesondere ist $\bigcap P$ *der kleinste Fixpunkt von* f*.*

Satz 2 (Kleene (vgl. [Kle52])) *Sei* $f : D \to D$ *eine stetige Funktion auf einer kettenvollständigen Halbordnung* $(D, \sqsubseteq)$ *mit kleinstem Element* $\bot$*. Dann ist*

$$\bigcup \{f^n \bot | n \geq 0\}$$

der kleinste Fixpunkt von f*.*

Während der Fixpunktsatz von Kleene eine natürlicherweise explizite Darstellung für einen imperativen Algorithmus liefert, zielt die Darstellung, Fixpunkte als Supremum von Prefixpunkten zu interpretieren, auf eine eher deklarative Interpretation des Fixpunktbegriffes ab. Obwohl beide Sätze ihrem Wesen nach das gleiche Konzept (siehe hierzu auch Abschnitt 3.1) beschreiben, tun sie dies jedoch einmal in einer eher prozedural imperativen Art und Weise (weshalb Satz 2 auch als Kleenes Fixpunkt*iteration* bezeichnet wird), während Satz 1 einen eher deklarativen Ansatz zur Darstellung des Fixpunktbegriffes liefert. Insbesondere lassen sich die beiden Sätze in dualer Weise ebenso auf *größte* Fixpunkte anwenden und führen dort zu dualen Resultaten. Da die den kleinsten Fixpunkten auf Kontrollflussstrukturen zugrundeliegenden Livenesseigenschaften jedoch eher einem imperativen Konzept entspringen und ein zu findender Pfad seinem Wesen nach eine prozedurale Beschreibung dieses Phänomens darstellt, fällt die Wahl an dieser Stelle ebenso auf die explizite Beschreibung von Fixpunkten und etabliert somit den kleinsten Fixpunkt μ als einen disjunktiven, den größten Fixpunkt ν als einen konjunktiven Operator, wenngleich diese Einteilung keinesfalls so kanonisch erfolgt, wie in den anderen Fällen.

Die Operatoren $\vee$ und $\wedge$ entstammen einer propositionalen Logik und sind daher natürlicherweise (wenn nicht gar ausschließlich) in einem deklarativen Kontext verankert und es ergibt sich die ebenso natürliche Zuteilung von $\vee$ zu den disjunktiven und $\wedge$ zu den konjunktiven Operatoren.

Schließlich bleiben noch die Operatoren $\Box$ und $\Diamond$ (die Box- und Diamonoperatoren, die über keine Kanteneinschränkungen spezifiziert sind). Diese beiden Operatoren sind (neben den atomaren Propositionen), die wohl am „stärksten" modellabhängigen Operatoren der Logik – über die Modellabhängigkeit der Fixpunktoperatoren werden wir uns an späterer Stelle in Abschnitt 5.1.3 noch vertiefend Gedanken machen. Diese Modellabhängigkeit fundiert somit auch die Betrachtungsweise als imperativ explizit motivierte Operatoren. Der $\Diamond$-Operator übernimmt hierbei das Analogon zum disjunktiven Operator einer propositionalen (und damit auch deklarativen) Logik auf imperativer modellabhängiger Ebene. Sowohl $\Diamond$ als auch $\vee$ beschreiben in jeweils unterschiedlichen Betrachtungsweisen ein atomar selektives Verhalten, zum einen auf Modell-, zum anderen auf Teilformelebene. Fordert $\vee$ die Existenz eines Kandidaten mit gewünschten

$$\begin{aligned} \phi ::= \quad & wahr \mid falsch \mid ap \\ & \mid \diamond \phi \\ & \mid \phi_1 \vee \phi_2 \\ & \mid \mu X.\phi \end{aligned}$$

Hierbei bezeichnet ap eine atomare Proposition und X eine Fixpunktvariable.

Abbildung 2.12: Operatoren von $\exists\mu$.

Eigenschaften in den zugehörigen strukturellen und hierarchischen nächst kleineren Stufen, so fordert $\diamond$ analog die Existenz eines Kandidaten im zugehörigen nächstmöglichen Schritt eines zugrundeliegenden Modells (vgl. Abschnitt 3 für eine genaue Beschreibung der Begriffe *Schritt* und *Stufe*). Die natürliche Interpretation von $\diamond$ begründet sich somit in einem disjunktiven Kontext. Analog lässt sich dies auf einen Vergleich von $\wedge$ und $\square$ übertragen, der schließlich in der Auffassung einmündet, $\square$ als einen konjunktiven Operator anzunehmen.

Eine besondere Rolle spielt in diesem Zusammenhang die Negation. Diese argumentiert, ähnlich den propositionalen Operatoren, über Teilformeln – genauer: über genau *eine* Teilformel. Sowohl das propositionale $\vee$ und $\wedge$, wie auch das Modell orientierte $\square$ und $\diamond$ argumentieren jedoch im Allgemeinen über mehrere Teilformeln bzw. Nachfolger. In beiden Fällen besitzen diese quasi eine „Abhängigkeitsarität“ von mindestens zwei. Die Negation argumentiert jedoch lediglich über genau eines seiner Argumente. Somit ist es nur schwerlich möglich, die Negation eindeutig einer konjunktiven oder disjunktiven Seite zuzuordnen, die ihre Bedeutung aus der unterschiedlichen Behandlung der abhängenden Teilkonstrukte (also Teilformeln oder Nachfolger) beziehen und nehmen sie deshalb in keine der beiden eingeschränkten Logiken $\exists\mu$ bzw. $\forall\nu$ auf. Mehr noch würde die Zuteilung der Negation zu einer der beiden Teillogiken die gesamte Trennung erübrigen. Durch die Regeln von de Morgan ließen sich die Operatoren der einen Klasse in die der andere Klasse überführen.

Zusammenfassend ergibt sich also für die Logik $\exists\mu$ die Auswahl der Operatoren, die in Abbildung 2.12 zusammengefasst ist.

Angemerkt sei hierzu noch, dass diese Operatoren genau die Analoga zu den häufig von Programmiersprachen geforderten Konzepte der Primitiven, der Kombinations- und der Abstraktionsmöglichkeit beschreiben. Hierbei übernehmen die Propositionen die Aufgabe der Primitiven. Die Operatoren $\diamond$ und $\vee$ erlauben die Kombination von Eigenschaften im Sinne von Alternativen, wie von Pfaden im Sinne einer Pfaderzeugung aus einzelnen Bausteinen. Die Bestimmung der kleinsten Fixpunkte mittels μ erlaubt schließlich die Abstraktion von explizit beschriebenen Eigenschaften bzw. Pfaden hin zu impliziten Beschreibungen.

Kapitel 3

Sichten, Konzepte und Aspekte

Die später folgenden Abschnitte etablieren einen Standpunkt bzgl. des Model Checkings, der, insbesondere bei einer erstmaligen Konfrontation, nicht intuitiv erscheint. Hierbei wird Model Checking in einen Spielkontext eingebettet, der es erlaubt, ein Model Checking Ergebnis aus dem Resultat eines Zwei-Personen-Spieles abzuleiten, wobei der zugrunde liegende Spielplan eine Verschmelzung aus Modell und Formel des aktuellen Model Checking Problems darstellt.

Um diesen Paradigmenwechsel einzuführen, möchte ich zunächst auf den folgenden Seiten allgemein die Interpretation von *Konzepten* und *Sichten* bzw. *Aspekten* vorstellen. Darunter können wir uns eine Metaabstraktion auf die in Informatik und Mathematik gängigen Konzepte vorstellen. Insbesondere stellt das Konzept selbst wiederum ein *Konzept* in der vorgestellten Interpretation dar und ermöglicht somit eine – wenn auch auf den ersten Blick nicht triviale – Anwendung auf sich selbst.

Atomos ist griechisch und beschreibt, speziell in Rahmen von physikalischen Systemen, die kleinsten Bestandteile, aus denen dieses zusammengesetzt ist. Hierbei beschreiben die Atome eine kleinste nicht weiter teilbare Einheit – insbesondere eine physikalische Einheit – die Grundlage und Bestandteil all dessen ist, was uns an Materie umgibt. Dieses Prinzip, kleinste unteilbare Einheiten zur Gestaltung größerer Systeme und Objekte – und dies müssen nicht zwangsläufig physikalische Objekte sein – heranzuziehen, findet ebenso in der Mathematik ihre Anwendung, etwa bei der Erzeugung von Ordnungsstrukturen und damit auch z.B. bei Zahlenmengen. Die kleinste unteilbare Einheit der natürlichen Zahlen ist demnach die 1. Die kleinste unteilbare Einheit der reellen Zahlen (eine infinitesimale Größe) dagegen ist nicht eindeutig beschreibbar und deren Anwesenheit sowie Abwesenheit bereitet in gewissen Kontexten der Infinitesimalrechnung (insbesondere innerhalb der Integration[1]) Schwierigkeiten. Gerade

[1]Fasse man etwa das Integral einer Funktion als grenzwertige Annäherung an die Fläche zwischen dem Graphen der Funktion und der Abszisse des Koordinatensystems auf, so lässt sich diese Fläche durch eine unendliche große Menge von Rechtecken unterhalb des Graphen

dieser Umstand zeigt eine scharfe Grenze der Reduzierbarkeit abzählbarer und überabzählbarer Strukturen auf ihre unteilbaren Einheiten auf, die auch in der Geschichte der Mathematik für vielerlei unterschiedliche Interpretationen dieses Umstandes gesorgt hat. Diskrete Strukturen besitzen damit die Eigenschaft, sich in abzählbarer Form (in gewisser Weise imperativ) zu beschreiben und zu erzeugen. Stetige Strukturen dagegen gewinnen ihre Form nicht aus der Zusammensetzung, als vielmehr aus ihrer (deklarativen) Beschreibung.

Löse man sich nun von einer rein *struktur*orientierten Beschreibung und verallgemeinere dieses Prinzip, Größeres aus Kleinerem zu erzeugen, so ergeben sich natürlicherweise drei Kategorien, auf die diese Art der Unterteilung anwendbar ist.

1. Struktur
2. Verhalten
3. Hierarchie

Struktur Der erste dieser Punkte wurde bereits am Beispiel physikalischer Systeme und der Vorstellung kleinster unteilbarer Einheiten (genannt *Atome*) etabliert. Eine Anwendung in informationsverarbeitenden Prozessen besteht nun gerade in der Strukturierung von Daten. In der Informatik werden Bits zu einem Byte, Elemente zu einer Liste, Zeichen zu einem Wort zusammengefasst. Hierbei ist dieses Prinzip genau ein Analogon, eine Abstrahierung der physikalischen Vorstellung, Größeres aus Kleinerem zusammensetzen zu können.

Verhalten meint in diesem Sinne, eine Tätigkeit – nicht notwendigerweise eine Veränderung – über die Zeit. Im imperativen Sinne möge man sich dies als Ausführung eines Programmes, in realen physikalischen Rechensystemen als die Folge von Schaltoperationen und in einem elektronischen Kreislauf als Fluss von Elektronen vorstellen. Eine kleinste unteilbare Einheit eines Verhaltens wird in diesem Sinne als *Schritt* bezeichnet. Im Sinne einer imperativen Programmiersprache ist dies also etwa gerade ein Befehl, im Sinne eines physikalischen Rechenwerks gerade ein Schaltvorgang, im Sinne eines elektronischen Kreislaufes ein Zustandswechsel zwischen zwei diskreten aufeinander folgenden Elektronenkonfigurationen, sofern dieser in einer geeigneten physikalischen Modellbildung existiert.

---

annähern. Doch welchen Flächeninhalt besitzt ein jedes dieser Rechtecke? Beträgt er 0, so wäre auch eine unendliche Aufsummierung dieser Rechtecke stets 0. Ist er dagegen ungleich 0, so könnte eine solche Zahl in unendlicher Aufsummierung kein endliches Ergebnis liefern. Solch eine Zahl müsste also genau irgendwo zwischen 0 und der kleinsten reellen Zahl, die ungleich 0 ist, liegen. Insbesondere wäre diese Zahl keine reelle Zahl mehr (vgl. [Rob74]).
Ein analoges Beispiel beschreibt die Vorstellung eines Käses mit unendlich vielen Löchern. Haben die Löcher eine endliche Größe, so besteht der Käse selbst nur noch aus Löchern und existiert demnach gar nicht mehr. Wie sieht es aber aus, wenn diese Löcher unendlich klein werden? Besteht der Käse dann nur noch aus Löchern oder nur aus Käse, oder gibt es gar einen Zustand irgendwo dazwischen, der unserer Vorstellung eines realen Käses noch am nächsten ist?

Zunächst scheinen die elementaren unteilbaren struktur- und verhaltensbeschreibenden Einheiten (Atom und Schritt) keinerlei Gemeinsamkeiten zu besitzen und gänzlich unterschiedliche Konzepte zu beschreiben. Bei genauerer Betrachtung zeigt sich jedoch, dass z.B. die Struktur selbst ein Verhalten beschreiben kann, gerade in dem Sinne, in dem ein Programm als zusammengesetzte Einheit von Schlüsselbegriffen oder sogar von Worten und Zeichen aufgefasst werden kann. In der Interpretation dieser strukturellen Beschreibung wird das Programm schließlich zu einer ein Verhalten beschreibenden Ausführung. Andererseits kann auch das Verhalten zur Beschreibung von Struktur herangezogen werden. So sind es etwa die Churchschen Numerale (siehe dazu den Abschnitt 3.2.2), die es ermöglichen, die Häufigkeit der Anwendung einer Funktion auf ihr Argument auf die natürlichen Zahlen abzubilden. Dies macht insbesondere deutlich, dass die Trennung zwischen Struktur und Verhalten mehr eine virtuell abstrakte Teilung, denn eine ihr immanent inne wohnende Separierung darstellt. Ob das zugrunde liegende Teilbare nun aus Atomen im Sinne einer Struktur oder aber aus Schritten im Sinne eines Verhaltens zusammengesetzt ist, ist vielmehr eine Interpretation, als eine „Wahrheit“, und kann ebenso in den entsprechend anderen Kontext überführt werden.

Hierarchie Der wichtigste Punkt der oben genannten Aufzählung ist jedoch der der Hierarchie. Hierunter verstehe man die kleinste Einheit dessen, was sich in Kategorien von über- und untergeordnet, von umfassend und eingebettet beschreiben lässt. Das Wesen der kleinsten Einheit der Hierarchie bezeichne man in diesem Zusammenhang als *Stufe*. Die Wichtigkeit und Allgemeingültigkeit dieses Begriffes erschließt sich, wenn man erkennt, dass sie genau das beschreibt, was sie ist.

Gerade diesem Punkt der Verallgemeinerung von Hierarchien unter Einbeziehung von struktur- und verhaltensorientierten Kategorien, widmet sich der Rest dieses Kapitels und beschließt dieses mit einer Aufzählung von Beispielen für diese Kategorie von aus unteilbaren Einheiten zusammengesetzten Strukturen, die ihrem Wesen nach eine Unterteilung, eine Ordnung beschreiben.

Ein Punkt für weitere Betrachtungen ist die Möglichkeit, diese kleinsten unteilbaren Einheiten wiederum zu neuen kleinsten unteilbaren Einheiten „zusammenzusetzen“ – wobei dies nicht zwangsläufig auf strukturelle Art und Weise geschehen muss. Hierbei scheint eine zusammengesetzte unteilbare Einheit zunächst widersprüchlich, ist sie doch eben nicht unteilbar, sondern eben aus Komponenten aufgebaut. Diese Atomizität besteht jedoch nur auf einer gewissen Betrachtungsebene und stellt sich demnach dem Betrachter nicht zwangsläufig als eine solche dar. So ist die 1 in der Betrachtung der natürlichen Zahlen atomar, in den reellen Zahlen dagegen nicht, ohne jedoch die Atomizität der 1 in den natürlichen Zahlen dadurch in Frage zu stellen. Abschnitt 4 über Spielgraphen beschreibt ein solches Beispiel, in dem ein Atom wiederum aus einer Stufe und einem anderen Atom zusammengesetzt und in dieser Betrachtung wiederum als unteilbar betrachtet wird. Hierbei werden Teilformeln und Modellknoten in eine neue Form eines Spielgraphknoten aggregiert und in diesem als unteilbare neue Einheit angenommen.

3.1 Konzepte und Aspekte

Unter einem *Konzept* verstehe man im Folgenden eine Sammlung von Einstellungen, Notationen, Konnotationen, Vorstellungen, Eindrücken und Benennungen im Kontext einer Ordnung, wobei das Konzept selbst eine Form des Übergeordnetseins einnimmt. Hierbei ist das Konzept selbst das zu untersuchende Gegenstandsfeld. Dieses Gegenstandsfeld zerfällt wiederum in Teile, die im nächsten Abschnitt vorgestellt und als *Aspekte* bezeichnet werden. Insbesondere sei hiermit auch nicht ausgeschlossen, dass das Konzept selbst wiederum als Teil – und somit als Aspekt – eines weiteren übergeordneten Konzeptes in Erscheinung tritt. Konzept und Aspekt unterscheiden sich in dieser Sichtweise also nicht in ihrer Struktur oder ihrem Verhalten, sondern ausschließlich in der Relation, in der sie zueinander stehen (in der hierarchischen Ordnung). Ebenso bleibt offen, ob das Konzept selbst wiederum in beliebiger Häufigkeit in weitere Aspekte zerfällt oder aber irgendwann in einer Unteilbarkeit seinen Ursprung manifestiert.

In dieser Betrachtung bildet das Konzept in der Gedankenwelt eines einzelnen Menschen den Erfahrungs- und Intuitionshintergrund, vor dem sich die Vorstellungen, Meinungen und Wertungen abzeichnen. Das Konzept wird schließlich zu einer allein gültigen Vorstellungen dessen, was es eigentlich beschreibt und verdeckt das Wesen dessen, für das es steht.

Aspekte unterscheiden sich ihrem Wesen nach nicht von den Konzepten. Erst in der hierarchischen Abtrennung von den Konzepten erhalten die Aspekte in dieser Betrachtung ihre Bedeutung und Wertigkeit. Wir fassen sie in diesem Kontext als Teile, Partien, Stücke oder Teilmengen eines sie beschreibenden und übergeordneten, sie umfassenden Konzeptes auf. Auch die Relation zu anderen Aspekten wird nicht weiter eingeschränkt, so dass sich die Aspekte auch ihrerseits überschneiden und überlappen können, oder aber keinerlei Übereinstimmungen aufweisen.

3.2 Beispiele für einen Wechsel der Sichten

Die nächsten Abschnitte führen die beiden soeben vorgestellten Begriffe Konzept und Aspekt nun in einigen beispielhaften Anwendungen zusammen, um das Wechselspiel und die Bedeutung der Begriffe mit Leben zu füllen. Es werden einige Bereiche der Mathematik (in ihrer strukturellen Erscheinungsform) und der Informatik (in ihrer verhaltensorientierten Erscheinungsform) in den sie umfassenden Konzepten und die sie konstituierenden Elemente vorgestellt. Es sei noch einmal darauf hingewiesen, dass diese Vorstellung bereits beim Leser ein Konzept aktiviert, das mit den vorgestellten Ideen und Konnotationen verknüpft ist.

3.2.1 Binomialkoeffizienten

Das in diesem Abschnitt nun vorzustellende Konzept ist mit dem Namen *Binomialkoeffizienten* benennbar. Selbst diese scheinbar einfache Benennung ist bereits der erste wichtige Schritt zum Verständnis des Konzeptes. Der Name selbst liefert bereits ein Werkzeug der Greifbarkeit. Sobald das Konzept mit einem Namen ausgestattet ist, kann es in Texten erwähnt und in einem Disput darauf verwiesen werden. Sobald das Konzept benannt wird, ist es „angreifbar" für andere Betrachtungen. Im Besonderen erschließen sich nach einer Benennung auch die weiteren Aspekte des Konzeptes leichter. Die Benennung stellt somit einen ersten Schlüssel für die Zugänglichkeit dessen dar, für das die Benennung (das Benannte) steht. Fraglich – wenn auch eher in psychologisch-philosophischer Hinsicht – ist der Umstand, ob ein Ding, eine Idee, eine Vorstellung überhaupt existiert, wenn es keinerlei Möglichkeit einer Benennung für eben jenes gibt. Am ehesten würde man solche Konzepte wohl als Gefühle, Intuition oder Stimmungen bezeichnen, die, aufkeimend aus scheinbar chaotischen und ungefilterten Sinneskonvulsionen, nur nach und nach ihren Weg in die Bewusstseinsperipherien bis hin zu gewollten und beabsichtigten Entscheidungen bahnen und in eine eng umgrenzte Klassifikation in Form eines Namens einmünden.

Ein ebenso wichtiger Aspekt wie der Name ist – zumindest in mathematischen und formalen Kontexten – die Notation. Der Binomialkoeffizient besitzt hierfür die hinlänglich bekannte Schreibweise $\binom{n}{k}$. Doch bereits an dieser Stelle wird die verzahnte Struktur von Notation und Benennung sichtbar, fragt man sich etwa nach den Namen der Komponenten dieser Notation. So gibt es etwa keinerlei weithin gültige Bezeichnung für die Komponenten der Notation n und k, wenngleich ein Name für das Gesamtkonstrukt besteht. Kann ein Fehlen einer Benennung also ebenso ein Hinweis für die Unwichtigkeit darstellen, wie das Vorhandensein eines Namens ein Hinweis auf die Wichtigkeit darstellt?!

Sobald jedoch das Konzept über eine Notation verfügt, wird es empfänglich für syntaktische Analogien, Transformationen und einer dieser Notation zugrunde liegende Intention. Erst die Notation erlaubt es dem Konzept einer gewissen rhetorischen Beliebigkeit zu entfliehen und sich in einer formalen Sprache einer gewissen strikten Bedeutung zu erfreuen. Ist die Benennung bereits eine Einbettung in ein zum Austausch gedachtes wissenschaftliches Netzwerk, so bereitet die Notation die formale Grundlage für die Verwendung dieses Konzeptes in einer wertfreien Umgebung. Ist die Benennung noch an eventuelle kulturelle Wertigkeiten des verwendeten Begriffes gebunden, so erzeugt die Notation einen in gewisser Weise wertfreien und ethikneutralen Raum, in dem das Konzept losgelöst von einer kulturgeschichtlichen Gebundenheit existieren kann. Jegliche Bezugnahme auf das Konzept ist nun lediglich an dieses formale System gebunden und losgelöst von dem, was es – im semantischen Sinne – beschreibt.

Name und Notation bilden somit die wichtigsten Grundpfeiler eines Konzeptes, da sich erst durch diese Aspekte weitere neue Aspekte etablieren können. Die Notation verhilft dem Konzept zu einer Identität und erlaubt erst jetzt, Aspekte eindeutig zuordnen zu können. Der Name ist der Greifpunkt, unter dem das Konzept – im wahrsten Sinne des Wortes – „begriffen" werden kann. Die Notation schließlich erlaubt eine Einbettung in eine formale Sprache. Erst

mit der Notation ist es möglich, das Konzept mit anderen ähnlichen – oder aber auch gänzlich unterschiedlichen – Konzepten zu vergleichen und neue weitere Aspekte, die auf diese Notation aufbauen, zu etablieren. Liefert die Benennung eine Einbettung in einen nicht-formalen alltäglichen Kontext, so bettet die Notation das Konzept schließlich in einen formalen Raum, von dem aus weitere Betrachtungen möglich werden.

Ein weiterer Aspekt des Konzeptes der Binomialkoeffizienten ist eine Beschreibung. Eine solche Beschreibung eröffnet ein Anwendungsfeld, eine Vorstellung und eine – wenn auch beschränkte – Sicht mit Beispielcharakter für das Konzept. Eine mögliche Beschreibung für Binomialkoeffizienten könnte etwa wie folgt aussehen:

> Der Binomialkoeffizient $\binom{n}{k}$ beschreibt die Anzahl der Möglichkeiten, k Elemente aus einer n-elementigen Grundmenge zu wählen.

Zunächst ist leicht zu erkennen, dass sowohl von Notation, als auch von der Benennung Gebrauch gemacht wird, um diesen nunmehr neuen Aspekt zu etablieren. Ferner erlaubt uns die umgangssprachliche Umschreibung des Konzeptes nun, eine gewisse, wenn auch diffuse, Vorstellung zu erlangen, die in diesem Falle auf scheinbar vorhandene kombinatorische und mengentheoretische Gesichtspunkte zurückgreift. Erstmalig ist dies also ein Aspekt, der auf andere Aspekte – nämlich genau diejenigen, die Mengenauswahl und Kombinatorik betreffen – zugreift und damit eine Brücke zu bereits Bekanntem etabliert.

Wichtig sei an dieser Stelle noch, auf die Gefahr zu verweisen, die mit einer solchen Beschreibung einhergeht. Gewinnt der Aspekt der Beschreibung schließlich definitorischen Charakter, so verdrängt er die anderen bekannten – und auch unbekannten und somit möglichen – Aspekte dieses Konzeptes und reduziert das Konzept selbst schließlich auf eben diese Beschreibung. Es sei noch angemerkt, dass die Gefahr nicht ausschließlich von der eingeschränkten Beispielhaftigkeit des Konzeptes herrührt. Auch eine Definition in einer formalen Beschreibungssprache wie der Mathematik kann ihrem Wesen nach ebenfalls Beispielcharakter besitzen und weitere Aspekte verdecken, indem die Definition den Alleinanspruch als Stellvertreter für das Konzept beansprucht und damit weitergehende alternative Betrachtungsweisen verdeckt. Dass das Konzept aber im umfassenden Sinne auch jenseits dieser Beschreibung existiert und sogar darüber hinausgeht, soll im Folgenden deutlich werden.

Bisher haben wir neben der Benennung und der Notation nur die Beschreibung als Aspekt des Konzeptes, das wir Binomialkoeffizienten nennen, beschrieben. Mit Etablierung der Notation gewinnen wir sogleich eine Fülle von Konnotation zu anderen Bereichen unserer formalen Sprache, die die Sprache der Mathematik ist. So lässt sich etwa auch die folgende Notation ableiten:

$$\frac{n^{\underline{k}}}{k!}$$

Hierbei beschreibt der Term $n^{\underline{k}}$ den Ausdruck[2] $n * (n-1) * \cdots * (n-k+1)$.

[2]Diese Notation hat ihren Ursprung in der diskreten Analysis; einer Theorie, die Integration

Diese Darstellung vermittelt wiederum einen Zusammenhang zum Konzept der Polynome und intendiert, Binomialkoeffizienten eben als solche Polynome aufzufassen. Dieser Wechsel der Begrifflichkeiten – und damit auch Begriffs*welten* – erlaubt es schließlich, Aspekte einer Begriffswelt zu benutzen und deren Auswirkungen auf die Begriffswelt des entsprechenden Konzeptes zu übertragen. Fassen wir Binomialkoeffizienten nun als Polynome auf, erschließen sich *automatisch* Begrifflichkeiten wie die der Nullstellen, Stetigkeit, Monotonie, Graphen etc., deren Übetragbarkeit auf das Konzept des Binomialkoeffizienten überprüft werden kann.

Die obigen Beispiele vermögen einen kleinen Eindruck der Mächtigkeit und Allgemeinheit eines Konzeptes zu geben, lassen sich nur genügend unterschiedliche Aspekte erschließen. Eine kleine Auswahl an weiteren Aspekten zum Konzept der Binomialkoeffizienten sei hier nur exemplarisch aufgelistet. Der interessierte Leser wird schnell weitere Schlüsse ziehen und weitere Aspekte identifizieren können. Ich verzichte daher an dieser Stelle auf eine weitere Erklärung und liefere diese Liste daher ohne weitere Anmerkungen.

- $\binom{n-1}{k} + \binom{n-1}{k-1}$
- $\frac{n!}{k!(n-k)!}$
- $\frac{n^{\underline{k}}}{k!}$
- $\binom{n}{n-k}$
- $(-1)^k \binom{k-n-1}{k}$

3.2.2 Null

Nachdem das letzte Kapitel eine konkrete Vorstellung eines Konzeptes und eines Aspektes anhand eines bekannten Beispiels vorgestellt hat, soll dieses Kapitel versuchen, das Verfahren auf ein weiteres Konzept anzuwenden. Hierbei wird sich herausstellen, dass die Extraktion eines Aspektes und die Konnotation eines Konzeptes mit gesellschaftlichen, manchmal sogar philosophischen, Betrachtungsweisen, schnell in grenzwertige Bereiche führt, deren einziger Ausweg in der Konzentration auf Notation und Transformation besteht. Das Beispiel, dem wir uns im Folgenden widmen möchten, ist das Konzept der *Null* oder auch des *Nichts*. Wir erkennen bereits hier, dass eine Benennung nur schwer, wenn nicht sogar unmöglich ist. Um jedoch eine eindeutige Grundlage für die weiteren Betrachtungen zu haben (das Konzept also bereits in diesem Sinne greifen zu können), wollen wir dasjenige Konzept beschreiben, das in seiner mathematischen Notation mit dem Zeichen 0 beschrieben wird.

Die Benennung des Konzeptes ist, wie wir bereits gesehen haben, äußerst schwierig und entzieht sich weitgehend seiner Behandlung. Beschreibungen wie *Null*

und Differentiation in einen endlichen diskreten Kontext einbettet und insbesondere auch ein Analogon zum Fundamentalsatz der Analysis liefert. Mehr Informationen dazu sind etwa bei [Gra03] zu finden.

oder *Nichts* lassen schnell den Vorwurf aufkeimen, etwas Unbeschreibliches beschreiben zu wollen. Wenn das, was wir beschreiben wollen, jedoch bereits in seiner Definition als nicht beschreibenswert deklariert wird, drehen wir uns offensichtlich im Kreis, etwas genauer definieren zu wollen, was per Definition bereits nicht definierbar ist. Wie können wir dem Nichts einen Namen geben, wenn es ja bereits das Nichts ist und sich damit auch einer Benennung weitgehend entzieht?!

Was bleibt uns jedoch, wenn der Name schon so äußerst schwierig zu fassen ist, um dieses Konzept weiter zu untersuchen? Wie bereits beim Konzept der Binomialkoeffizienten (Abschnitt 3.2.1) zu erkennen war, scheint die Notation ein fruchtbarer Weg für weitere neue Aspekte zu sein, den wir auch an dieser Stelle weiter beschreiten wollen. Da die Einbettung in einen formalen Raum mittels einer Notation in einem ethikfreien und wertneutralem Sinne vollzogen werden kann, wird durch die Notation eine rein syntaktische Behandlung der 0 möglich.

Betrachten wir zunächst wieder die bekannte mathematische Notation 0, so ist auch hier fraglich, ob ein Zeichen für das Konzept, das eigentlich das Nichts beschreiben soll – und somit auch zeichenlos ist – zutreffend ist. Oder lässt sich das Konzept des Nichts nicht vielmehr als Relation aus einem oder mehreren Etwas erzeugen?! Verfolgen wir diesen Weg, so lässt sich die folgende Beschreibung der 0 leicht herleiten (i beschreibt hier die imaginäre Einheit).

$$e^{i*\pi} + 1$$

Wird dieser Gleichung in vielerlei Hinsicht eine gewisse Ästhetik, manchmal gar Magie zugesprochen, da sie weiß, die wichtigsten Konzepte der Mathematik zu vereinheitlichen (neben natürlichen, kommen transzendente und komplexe Zahlen in ihr vor), ist sie dennoch nicht geeignet, das Nichts genügend zu beschreiben, da sie eben dieses aus dem Etwas erzeugt und nur eine Relation zu Bestehendem etabliert[3].

Um den Leser nach so vielen Einschränkungen und Rückschlägen nicht gänzlich im Ungewissen über die Bedeutung des Zeichens 0 zu lassen, betrachten wir zum Schluss noch einen weiteren Aspekt des Konzeptes, der von Alonzo Church etabliert wurde und auf die Church'schen Numerale[4] zurückgeht. Die folgende Notation beschreibt gerade Church‘s Vorstellung des Nichts.

$$\lambda f.\lambda x.x$$

Obige Notation beschreibt eine anonyme Funktion, die bei Eingabe eines Argumentes eine weitere Funktion – nämlich genau die Identität – zurückliefert.

[3] Philosophisch oder auch religiös interessierte Leser mögen sich an dieser Stelle fragen, ob das Etwas nun aus dem Nichts, oder aber das Nichts aus dem Etwas entstanden ist. Scheinbar etabliert das eine genau das andere und liefert somit keinen Ausweg aus dem altbekannten Henne-Ei-Problem.

[4] Die Church'schen Numerale beschreiben eine Einbettung der natürlichen Zahlen in einen Kontext der Funktionsapplikation des λ-Kalküls und wurden von Alonzo Church in [Chu65] vorgestellt.

Nebenbei sei erwähnt, das sich diese Vorstellung konsistent in eine vollständige Arithmetik auf den natürlichen Zahlen einbetten lässt. So lassen sich mit einer entsprechenden Angabe des Konzeptes der 1 (gar des Konzeptes der Existenz?) und der Addition die natürlichen Zahlen vollständig beschreiben und es stellt sich somit die Frage nach der *tatsächlichen* Gestalt der natürlichen Zahlen – der diskreten abzählbar unendlichen formalen Ordnungsstrukturen – wenngleich nunmehr zwei Darstellungen davon existieren. Doch sind diese Darstellungen äquivalent oder muss man sich bei genauerer Betrachtung ebenso Gedanken über das Konzept von Äquivalenz und Gleichheit machen, und zerfließt damit das gesamte formale wissenschaftliche Gefüge in einer Soße aus Beliebig-, Interpretierbar- und Austauschbarkeit? Oder aber ist der Kern dessen, was es zu wissen gilt, genau der, dass es keine absoluten Wahrheiten gibt, sondern sich jede Wahrheit im Rahmen ihrer Einbettung in ihre Konzepte bewegt und sich darin als wahr oder eben falsch behauptet?

Genau hier schließt sich der Kreis zum Model Checking. Denn auch das Model Checking selbst weist Eigenschaften doch nur *relativ* zur formalen Modellbildung nach, nicht jedoch absolut zum dem Modell zugrundeliegenden System. Die Aussage, dass ein mit Model Checking überprüftes System eine bestimmte Eigenschaft besitzt, muss also – selbst in der Annahme eines korrekten Model Checkers – dahingehend relativiert werden, dass der Model Checker eine einer formalen Notation entsprechenden Eigenschaft gegen ein formales Modell des Systems validiert. Die rückbezüglichen Aussagen zum modellierten System und der spezifizierten Eigenschaft müssen also ebenso überprüft werden. Diese Überprüfung muss jedoch – zumindest zur Zeit noch – vom Menschen selbst vollzogen werden. Hier ist wiederum die Wichtigkeit zu erkennen, die sich aus Ergebnissen des Model Checkings ergeben, die über reine Ja/Nein-Aussagen hinausgehen und dem Menschen in Form von Begründungen, Beweisen, Gegenbeispielen und Argumenten anleiten, die Modellbildung bzgl. des System oder der Eigenschaft geeignet zu korrigieren bzw. anzupassen.

3.2.3 Model Checking

Die beiden vorgestellten Konzepte der Binomialkoeffizienten und der Null ließen bereits erahnen, welch neue Sichten und damit auch Möglichkeiten sich bei einem Wechsel des Standpunktes und der Etablierung weiterer Aspekte ergeben können.

Die in Abschnitt 2 vorgestellte Sicht des Model-Checking-Begriffes werde im Folgenden als die *klassische* Sicht bezeichnet und beschreibt den Zusammenhang, der sich aus der Betrachtung eines deklarativen Konzeptes unter einer imperativen prozeduralen Betrachtungsweise natürlicherweise ergibt. Diese Verschmelzung auf einer eher semantischen Ebene – die Probleminstanzen unterscheiden zwischen Modellen und Eigenschaften – lässt sich nun auf einer syntaktischen Ebene fortführen. Die Begriffe *syntaktisch* und *semantisch* beschreiben hierbei auf der einen Seite die konkrete mathematische Problemformulierung und auf der anderen Seite das dahinterliegende intendierte Modell (im Sinne einer Vorstellung), das dieser Problemformulierung zugrunde liegt. Führt man diesen Verschmelzungsprozess auf der syntaktischen Ebene der Problemformulierung

fort – unifiziere man also sowohl die eigenschaftsbeschreibende Temporallogik als auch die verhaltensbeschreibende Modellbildung – so ergibt sich eine spieltheoretische Beschreibung des ursprünglichen Gesamtproblems. Nunmehr sind also nicht Modelle und Eigenschaften, die gegeneinander validiert werden sollen, die Problemeingaben. An ihre Stelle sind eine Mischung aus Regeln, Gewinnbedingungen, Spielern und ein den Spielverlauf regulierendes Instrument des Spielgraphen getreten, wobei die gewünschte Problemlösung von Ja/Nein-Antworten in eine Gewinner/Verlierer-Situation überführt wurde. Eine dieser Neuinterpretationen des Ergebnisses besteht nun jedoch in der Möglichkeit, über den Problemlösungsweg zu argumentieren. In der klassischen Sicht geschieht dies meist über eine Argumentation über Teilformeln und deren Einfluss auf das Gesamtergebnis. Die nunmehr spielbasierte Sicht dagegen, erlaubt es über Spielstrategien, Gewinnmengen und Spielsituationen zu argumentieren. Zwar lassen sich diese Begriffe jederzeit in die klassische Sicht übersetzen, doch bietet der Wechsel in die spielorientierte Sicht an dieser Stelle einen intuitiveren Zugang zum Gesamtproblem.

Wichtig ist also nicht, die Gemeinsamkeiten zwischen der klassischen und der spielbasierten Sicht des Model-Checking-Problems herauszustellen, als vielmehr die den jeweiligen Sichten eigenen und damit neuen Aspekte des Gesamtkonzeptes hervorzuheben. Hiermit wird auch der Begriff der *klassischen* Sicht überflüssig, ist es doch eine rein historische Bezeichnung, die dem Kern des Paradigmenwechsels doch nur in zeitlich-historischer Sicht entspricht, jedoch keinerlei substanzielle Unterschiede aufzeigt.

Eben dieser Paradigmenwechsel von einer ehedem *klassischen* Sicht in eine spielbasierte Sicht übernimmt dieses letzte Kapitel und führt im nächsten Kapitel die Grundlage für diesen Wechsel (die Spielgraphen) ein. Ferner beschreibt der weiteren Verlauf dieser Arbeit eine formalere Fundierung der in diesem Abschnitt eingeführten Konzepte und erläutert den Sichtenwechsel an sich, wie auch seine Möglichkeiten, am Beispiel des eben beschriebenen Wechsels im Model Checking Kontext.

Kapitel 4

Spielgraphen

Im Folgenden soll eine neue Sicht für das Model Checking Problem vorgestellt werden. Hierbei wird Model Checking nicht mehr nur als ein Verfahren gesehen, das auf zwei Strukturen – nämlich der Formel auf der einen und dem Modell auf der anderen Seite – als vielmehr auf nur noch einer Struktur arbeitet. Zu diesem Zweck werden die getrennten Problemteile *Eigenschaft* und *Modell* in eine gemeinsame Struktur überführt. Eine solche Darstellung des Model Checking Verfahrens, die es auf Grundlage eines Zwei-Personen-Spieles[1] auf einem so genannten *Spielgraphen* etabliert hat, wobei das Spielergebnis Auskunft über das Model Checking Ergebnis selbst gibt, wurde erstmalig in [EJS93] vorgestellt. Hierbei werden sowohl das Modell als auch die Formel miteinander zu einem Spielgraphen „verschmolzen", dessen Knoten eine Teilformel und einen Knoten des Ursprungmodells als jeweils kleinste unteilbare Komponenten (siehe dazu auch Kapitel 3) des Ursprungsproblems in sich vereinen. Die Knoten dieses Spielgraphen sind in zwei Mengen partitioniert und beschreiben dadurch, welcher der Spieler am jeweiligen Knoten am Zuge ist. Durch abwechselndes Ziehen an den Knoten wird schließlich ein Gewinner dadurch ermittelt, dass er seinen Gegner in eine Situation bringt, in der dieser nicht mehr ziehen kann, oder aber er konstruiert ein (unendliches) Spiel, bei dem er immer wieder „möglichst günstige" seiner Knoten besucht. Hierbei versuchen die Spieler jeweils die Formel an dem Knoten, an dem sie sich befinden, durch geschickte Wahl der Nachfolger zu beweisen oder aber zu widerlegen. Die beiden Spieler entscheiden sich hierbei während des Spieles an ihren Knoten jeweils immer gleich (wählen an einem Knoten also immer den gleichen Nachfolger – auch wenn sie diesen wiederholt aufsuchen) und insbesondere unabhängig davon, wie sie zu diesem Knoten gelangt sind.

Zwei Beispiele für derartige Spielgraphen zeigt Abbildung 4.1. Die Knoten 1, 3, 5, 6 und 7 gehören dem $\Diamond$-Spieler, die Knoten 2 und 4 dagegen dem $\Box$-Spieler.

[1] Die Bezeichnung dieser beiden Spieler ist in der Literatur relativ uneinheitlich und deckt mit Bezeichnungen von 0/1 über $\Box/\Diamond$ und $\vee/\wedge$ bis zu $\forall$lbert/$\exists$loise ein buntes Spektrum allerlei Assoziationen zu den Bedeutungen der Spielercharaktere ab. Im Folgenden wir aus Gründen der Übersicht und Einfachheit die männliche Form verwendet, wenngleich mit Bezeichnungen wie Adam/Eva oder Albert/Eloise auch durchaus würdige geschlechtsdiskriminierende Varianten existieren.

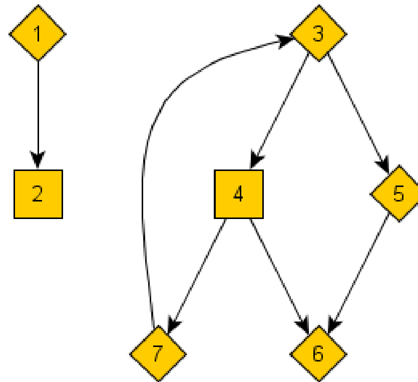


Abbildung 4.1: Ein kleines Beispiel für einen Spielgraphen.

Startet ein solches Spiel nun an dem Knoten 1, so ist Spieler ⋄ am Zug und muss einen Nachfolger wählen. An dieser Stelle existiert nur der Knoten 2 als Nachfolger. Stelle man sich die Knoten als Container vor und hinge an jedem Knoten eine Formel, so wäre es die Aufgabe des ⋄-Spielers nun zu beweisen, dass seine Formel gilt, indem er für mindestens einen Nachfolger (für mindestens eine Teilformel) zeigt, dass dieser die Formel erfüllt. In diesem Falle gibt es nur den Nachfolger 2. An dieser Stelle wäre Spieler □ am Zuge. Spieler □ hat nun nicht die Verpflichtung, die Formel an seinem Knoten durch geschickte Nachfolgerwahl zu beweisen, als vielmehr zu widerlegen, indem er für einen Nachfolger zeigt, dass deren Teilformeln *nicht* erfüllt sind. In diesem Falle gibt es keinen Nachfolger und Spieler □ kann seinen Beweisverpflichtungen nicht nachkommen. Der ⋄-Spieler hat damit gezeigt, dass es mindestens einen Nachfolger (mindestens eine Teilformel) gibt, die er belegen kann – nämlich genau die in Knoten 2 (da Spieler □ wiederum aus Knoten 2 nicht für alle Nachfolgeknoten (für alle Teilformeln) das Gegenteil beweisen kann).

Startet das Spiel dagegen aus Knoten 3, so ist die Wahl des Knotens 5 für den ◇-Spieler keine gute Wahl, da von diesem Knoten wiederum nur ein Zug zu Knoten 6 möglich ist, von dem aus es keine Möglichkeit gibt, den eigenen Beweisverpflichtungen nachzukommen. Wenn Spieler ⋄ also gewinnen (und damit die an Knoten 3 befindliche Formel belegen) möchte, bleibt nur die Wahl für Knoten 4. Hier jedoch kann Spieler □ mit der Wahl von Knoten 6 zeigen, dass sich nicht alle Teilformeln der Formel an Knoten 4 belegen lassen (Knoten 6 ist nämlich genau eine Wahl für ein solches Gegenbeispiel). Spieler ⋄ kann also nicht verhindern, dass Spieler □ gewinnt; oder anders ausgedrückt, Spieler ⋄ kann nicht gewinnen, da Spieler □ in jedem Fall gewinnt.

Kanten zwischen Knoten beschreiben also eine Teilformelrelation, wobei eine Kante von einer Formel auf ihre Teilformeln verweist. Eine Beispiel für eine Formel, die dem linken Spielgraphen aus Abbildung 4.1 entspräche ist etwa $tt \vee tt$. Diese Formel hängt nur von ihren beiden Teilformeln tt und tt ab, die jedoch identisch zueinander sind. Im Speziellen besitzen diese Teilformeln weder weitere

Teilformeln noch sind sie widerlegbar – entsprechen also gerade der dargestellten Situation. Interpretieren wir das Erfüllen einer Formel mit der Belegung 1 und das Widerlegen einer Formel mit der Belegung 0, so entspricht der linke Graph in Abbildung 4.1 ebenso den Formeln[2] $max(1,1)$ oder $1 * 1$.

Eine bislang unbehandelte Kante zwischen den Knoten 7 und Knoten 3 widerspricht scheinbar dem Konzept der Interpretation einer Kante als Teilformelrelation. So glaubt man doch, dass eine Teilformelrelation transitiv sei und eine Formel in Knoten 7 demnach Teilformel des Knotens 3 wäre. Wie aber kann Knoten 3 zur gleichen Zeit Teilformel von Knoten 7 sein, wenn beide Formeln gleichzeitig verschieden sein sollen? Hierfür ist es hilfreich, sich Kanten nicht länger als Beschreibung einer Teilformelrelation vorzustellen, sondern als *Abhängigkeitsrelation*. So gibt es also eine Kante von einer Formel ϕ zu einer Formel ϕ', wenn der Wert von ϕ ausschließlich durch den Wert von ϕ' bestimmt wird. Dies steht im Einklang mit der Interpretation von Kanten als Teilformelrelation, da in applikativen Kontexten der Wert einer Funktionsapplikation vollständig und ausschließlich von ihren Argumenten abhängt. Der Wert einer Funktion f angewendet auf ein Argument x (die Notationen $f(x)$ oder auch fx werden in dieser Arbeit synonym verwendet) hängt also ausschließlich vom Wert der Argumente, vom Wert von x, ab.

Die Bedeutung der Kante von Knoten 7 zu Knoten 3 erschließt sich, wenn man den Knoten 3 als Fixpunktbestimmungsoperator auffasst. Ist man also an einem Fixpunkt einer Funktion interessiert, so lässt sich dieser Operator durch den Y-Operator[3] beschreiben und bestimmen. Ist etwa die Einsfunktion f_1 gegeben (definiert als $f_1 x = 1$), dann gilt $Yf_1 = 1$, da 1 der (insbesondere einzige) Fixpunkt von f_1 ist. Da $Y\phi$ jedoch ein Fixpunkt von ϕ ist, gilt ebenso $\phi(Y\phi) = Y\phi$ (genannt Fixpunkteigenschaft). Hieraus ist erkennbar, dass der Spielgraph für eine solche Fixpunktbestimmung eine Kante $\phi(Y\phi) \to Y\phi$ enthält, da der Wert von ϕ vom Wert seiner Argumente (in diesem Falle $Y\phi$) abhängt. Wegen der Fixpunkteigenschaft von $Y\phi$ lassen sich Quelle und Ziel dieser Kante jedoch umformen zu der neuen Kante $Y\phi \to \phi(Y\phi)$ und es ergibt sich gerade die gleiche Kante mit gegensätzlicher Orientierung. Hierbei ist festzustellen, dass die Funktion ϕ sowohl von ihren Argumenten abhängt, wie auch, dass für bestimmte Argumente (eben Fixpunkte dieser Funktion), diese Argumente von diesem Fixpunkt abhängen (eben identisch zu diesem sind). Angewandt auf das Beispiel aus Abbildung 4.1 beschreiben die Knoten 3, 4 und 7 also – zum Beispiel – eine Fixpunktbestimmung, wobei Knoten 3 den eigentlichen Fixpunkt, Knoten 4 die Formel, über die der Fixpunkt bestimmt werden soll und Knoten 7 schließlich die Variable der Formel, über die der Fixpunkt bestimmt werden soll, repräsentieren. Nicht jedoch nur im Falle einer Fixpunktbestimmung hängt das Argument von der Funktion selbst ab; jede rekursiv beschriebene Funktion erfüllt diesen Tatbestand und wäre demnach ein ebenso geeigneter Kandidat für die Interpretation dieser drei Knoten.

Eine Gewinnstrategie für einen Spieler – also das Wissen, welcher Nachfolger

[2] Ein Fuzzy-Logiker würde an dieser Stelle ebensogut von T- und S-Spielern reden können, wenn er T- und S-Normen als Verallgemeinerungen konjunktiver bzw. disjunktiver Konzepte interpretiert.

[3] In Anlehnung an die von Moses Schönfinkel und Haskell Curry eingeführte *kombinatorische Logik* (vgl. [Fey65]) definiert als $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$.

an einem bestimmten Knoten zwangsläufig zum Sieg führt – entspricht damit dem Model Checking Ergebnis. Die Knoten, von denen aus der $\diamond$-Spieler (also der beweisende Spieler) gewinnt (die also in seiner Gewinnstrategie liegen), repräsentieren gerade jene Knoten im ursprünglichen Modell, für die sich die im Knoten des Spielgraphen enthaltene Eigenschaften nachweisen lässt.

Eine andere Vorstellung mag in der Auffassung eines Spielgraphen als Kontrollflussstruktur einmünden, deren verzweigende konditionale Komponenten „unschlüssig" über ihr Verhalten sind. Der Model Checking Prozess liefert nun Gewissheit für derartige Komponenten und legt sie auf eine Richtung fest. Möge man sich diese Konditionalknoten als Repräsentanten von IF-Statements auffassen, so wird die in diesen Verzweigungen auszuwertende Eigenschaft von einer zunächst nicht-deterministischen Auswahl in eine deterministische transformiert.

Die folgenden beiden Abschnitte beschreiben zunächst eine Formalisierung des oben bereits Geschriebenen und verdeutlichen den Verschmelzungsprozess von Modell und Formel in Abschnitt 4.1 für die reduzierte Logik $\exists\mu$ (siehe Abschnitt 2.2.2). Daran anschließend verknüpft Abschnitt 4.3 die bisherige Intuition des Model Checking Ergebnisses mit dem Ergebnis, das sich aus einem auf diese Art erstellten Spieles ergibt.

4.1 Synthese von Formel und Modell

Spielgraphen für das Model Checking Problem stelle man sich am besten als eine Art Kreuzprodukt über das Modell zum einen, und über die Teilformeln der zu untersuchenden Eigenschaft zum anderen vor. Hierbei entsteht ein neuer Graph (eben der Spielgraph), dessen Knoten jeweils eine Teilformel der ursprünglichen Formel und einen Knoten des ursprünglichen Modells repräsentieren. Der folgende Abschnitt fundiert die Notation und Formalisierung dessen, was unter einem Spielgraphen zu verstehen ist. Hierbei erfolgt eine Beschränkung auf die auf disjunktive Operatoren reduzierte und demnach insbesondere alternierungsfreie Logik $\exists\mu$.

Definition 1 (Spielgraph für $\exists\mu$ (nach [EJS93])) *Ein* Spielgraph *für eine Kripke Struktur $K = (V_k, E_k, I)$ über einer Menge atomarer Propositionen AP und eine (alternierungsfreie) Formel $f \in \exists\mu$ ist ein Tupel $G = (V_\Box, V_\diamond, E)$ mit einer Knotenmenge V und einer Kantenmenge E, wobei die folgenden Eigenschaften gelten.*

$V = V_\Box \cup V_\diamond$ mit $V = \{(s, g) | s \in V_k, g \in SF(f)\}$. $SF(f)$ beschreibt hierbei die Menge aller Teilformeln von f. Ferner gilt

- *$(s, g) \in V_\Box$, wenn $g \in AP \wedge g \in I(s)$.*
- *$(s, g) \in V_\diamond$, wenn*
  - *$g \in AP \wedge g \notin I(s)$*

  - $g = \mu X.g'$
  - $g = g' \vee g''$
  - $g = \Diamond g'$
  - g *ist Fixpunktvariable.*

$E \subseteq (V_\Box \cup V_\Diamond) \times (V_\Box \cup V_\Diamond)$ *mit* $(s, g) \rightarrow (s', g') \in E$ *wenn*

- $g = \mu X.g'$ *und* $s = s'$
- $g = X$, *wobei* X *eine Fixpunktvariable ist,* $s = s'$ *und* $g' = \mu X.g''$ *ist die Fixpunktformel, an die* X *gebunden ist.*
- $g = g' \vee g''$ *und* $s = s'$
- $g = \Diamond g'$ *und* $(s, s') \in E_k$.

Hierbei lässt sich SF leicht induktiv definieren. Für weitere Details und eine formale Beschreibung von SF sei auf den Abschnitt A.3 im Anhang verwiesen.

In dieser Definition beschreiben die Knotenmengen $V_\Box$ und $V_\Diamond$ jeweils konjunktive bzw. disjunktive Operatoren. Die Regeln, nach denen auf einem solchen Spielgraphen gespielt werden kann, ist in der folgenden Definition genauer beschrieben.

Definition 2 (Spiel) *Ein* Spiel *besteht aus einem Spielgraphen* $G = (V_\Box, V_\Diamond, E)$ *und einem Startknoten* $v \in V_\Box \cup V_\Diamond$ *und wird von zwei Spielern* $\Box$ *und* $\Diamond$ *auf folgende Weise gespielt.*

Der Spieler $i \in \{\Box, \Diamond\}$ *wählt vom aktuellen Knoten aus den nächsten Knoten* w *mit* $(v, w) \in E$, *wenn* $v \in V_i$.

Man stelle sich den Spielgraphen also als einen gewöhnlichen gerichteten Graphen mit zwei Knotentypen $\Box$ und $\Diamond$ vor, die den jeweiligen Spielern zugeordnet sind. Ausgehend von einem Knoten darf gerade der Spieler in Richtung der zu diesem Knoten inzidenten Kanten ziehen, dem dieser Knoten gehört. Anschließend ist wiederum derjenige Spieler am Zug, der Eigentümer dieses neuen Knotens ist. Insbesondere wird also kein abwechselndes Ziehen gefordert und ferner wird in den gegebenen Definitionen nicht einmal die Existenz eines Gegenspielers gefordert. Spiele, bei denen die ganze Zeit nur ein Spieler am Zuge ist, stehen also ebenso im Einklang mit dieser Definition. Der Sinn solcher Spiele und die Frage nach der Gerechtigkeit ergeben sich erst aus den Gewinnbedingungen; gerade jenen Bedingungen, die Aussagen über den Ausgang eines Spielverlaufes treffen – wenngleich die Grad der möglichen Einflussnahme natürlich erheblich geringer ist, wenn ein Spieler überhaupt nicht ziehen kann.

Hiermit ist der Vereinigungsprozess der Eingabeinstanz abgeschlossen und die zuvor separaten Teile Modell und temporal-logische Eigenschaft in einem Spielgraphen zusammenfassend vereinigt. Was zu tun bleibt, ist die noch verbliebenen zuvor implizit in den Auswertungsregeln der Operatoren gehaltenen Formalismen als Gewinnbedingungen in das Spielkonzept einzugliedern. Gerade dies wird der nächste Abschnitt über die Gewinnbedingungen leisten.

4.2 Gewinnbedingungen

Mit der Verschmelzung von Modell und Formel steht nun zwar ein Spielgraph zur Verfügung, auf dem abwechselnd von den beteiligten Spielern gezogen werden kann, doch besteht noch keine formale Motivation darüber, nach welchen Kriterien die Spieler eine Auswahl treffen sollen, wenn sich mehrere alternative Zugmöglichkeiten ergeben. Es besteht also noch keine Gewissheit darüber, wann ein Spieler gewinnt bzw. wann er ein Spiel verliert. Diese Bedingungen, unter denen ein Spiel beendet und ein Sieger eindeutig ermittelt werden kann, bezeichne man, dem Titel dieses Abschnittes entsprechend, als *Gewinnbedingungen.* Eine formale Definition dieser Bedingungen wird in der folgenden Definition zusammengefasst. Auch diese Definition (wie auch bereits die Definition für Spielgraphen) berücksichtigt den Umstand, dass die Formel, aus der der Spielgraph gewonnen wurde, aus $\exists\mu$ stammt. Die Abwesenheit von größten Fixpunkten (und damit ebenso von Alternierung) bedingen in diesem Fall eine einfachere und knappere Formulierung.

Definition 3 (Gewinnbedingungen) *Ein Spiel wird von einem Spieler* gewonnen*, wenn eine der folgenden Bedingungen eintritt.*

- *Kann ein Spieler $i \in \{\Box, \diamond\}$ irgendwann im Verlauf des Spieles nicht mehr ziehen, dann verliert dieser Spieler i das Spiel und der jeweils andere Spieler gewinnt.*
- *Entsteht ein unendliches Spiel, bei dem mindestens ein Spieler immer wieder in der Lage ist, zu ziehen, so gewinnt Spieler $\Box$.*

Anzumerken ist, dass im Falle eines unendlichen Spieles immer der Spieler $\Box$ gewinnt. Da der erzeugte Spielgraph für eine Formel aus $\exists\mu$ (wie definiert in Abschnitt 2.2.2) konstruiert wurde, kann diese Formel insbesondere weder eine Fixpunktalternierung noch die Bestimmung eines größten Fixpunktes überhaupt beinhalten. Außerdem besteht die einzige Möglichkeit, einen Kreis in einem solchen Spielgraphen zu schließen, darin, nach Erreichen einer Fixpunktvariablen zurück zu einem Knoten zu ziehen, dessen Funktion die Fixpunktbestimmung (eines kleinsten Fixpunktes) über gerade diese Variable beinhaltet. Dieses scheinbare Ungleichgewicht zwischen den beiden Spielern, wird im folgenden Abschnitt näher erläutert.

4.3 Vom Spielergebnis zum Model Checking Ergebnis

Wurden das Modell sowie die auf ihm zu überprüfende Eigenschaft in einen Spielgraphen synthetisiert, ist ein nach den soeben beschriebenen Regeln geführtes Spiel irgendwann beendet und steht schließlich ein Gewinner fest, so verbleibt es diesem Abschnitt, nunmehr den Bogen zum ursprünglichen Problem (dem eigentlichen Model Checking Problem) zu schließen und das Ergebnis dieses

Spieles als Ergebnis für das ursprüngliche Model Checking Problem zu interpretieren. Wie lässt sich aus der Tatsache, dass Spieler $\Box$ oder Spieler $\diamond$ das Spiel gewinnt, ein Rückschluss darauf ziehen, ob eine gewisse Eigenschaft an einem Knoten im ursprünglichen dem Spielgraphen zugrunde liegenden Modell erfüllt ist oder nicht? Wie lässt sich also aus der Spielinformation wiederum eine Model Checking Information extrahieren?

Zu diesem Zwecke wird der Spielgraph in seiner Interpretation als nicht-deterministische Variante eines Kontrollflussgraphen als Model Checking Ergebnis interpretiert. Die Interpretation des Spielgraphen als Model Checking Ergebnis geht also Hand in Hand mit der Interpretation (im Sinne einer Semantik gebenden Verhaltensbeschreibung) des auf diese Art und Weise entstandenen Kontrollflussgraphen.

Wie schon eingangs in der beispielhaften Einführung von Spielgraphen, betrachte man zunächst einen Spielgraphen, der aus einer fixpunktfreien Formel entstanden ist und demnach einen DAG (directed acyclic graph) beschreibt. Insbesondere handelt es sich nicht um einen Baum, wie der Graph belegt, der aus der Formel $(a \wedge a) \vee a$ (und einem Modell mit nur einem Knoten s ohne Kanten) entsteht und die Kanten $(s, (a \wedge a) \vee a) \rightarrow (s, a \wedge a) \rightarrow (s, a)$, wie auch $(s, (a \wedge a) \vee a) \rightarrow (s, a)$ besitzt. Teilformeln würden in dieser Darstellung also mehrfach benutzt. Ebenso besitzt der Graph keinerlei Kreise, da Kanten in diesem Fall nur von Formeln auf deren Teilformeln verweisen. Erst durch Zulassung von Fixpunktoperatoren beschreiben die Ausdrücke rekursive Eigenschaften, die ihren Wert aus sich selbst heraus beziehen und Kreise im Spielgraph ermöglichen.

In einem ersten Schritt nehmen wir an dieser Stelle zunächst an, dass der Spielgraph aus einer fixpunktfreien Formel und einem beliebigen (jedoch endlichen) Modell entstanden sei. Die beiden Spieler, deren Knoten für den $\Box$-Spieler auf die konjunktiven, für den $\diamond$-Spieler auf die disjunktiven Knoten aufgeteilt wurden, spiegeln ihrem Wesen nach den Beweiser bzw. den Widerleger ihrer Formeln wieder. Diese Zuteilung steht im Einklang mit der Vorstellung der Spieler als *homo oeconomicus*, die ihren Wesen nach den ökonomischen Prinzipien der Aufwandsminimierung bei gleichzeitiger Profitmaximierung handeln.

Der $\diamond$-Spieler übernimmt in diesem Falle die Rolle des Beweisers. Seine Formeln sind von der Form $\phi_1 \vee \phi_2$, $\diamond\phi$ oder aber $\mu X.\phi(x)$. Er kann mit sehr wenig Aufwand Formeln beweisen, jedoch nur mit großem Aufwand Formeln widerlegen. Für eine Formel $\phi_1 \vee \phi_2 \vee \ldots \vee \phi_n$ etwa genügt es – selbst wenn dieser Ausdruck unendlich groß wäre – lediglich *einen* (und damit insbesondere endliche viele) Kandidaten anzugeben, der die gesamte Formel belegt. Würde er dagegen versuchen wollen, die Formel zu widerlegen, so müsste jede der – unter Umständen unendlich vielen – Teilformeln ϕ_i für sich widerlegt werden. Argumentiert man nicht über strukturelle Einheiten der Eigenschaften (also Teilformeln), sondern über deren Gegenstücke im Modell (über die Nachfolger), so lässt sich dieses Argument ebenso auf Formeln der Gestalt $\diamond\phi$ übertragen. Letztendlich folgt die Argumentation ebenso für eine Verknüpfung von strukturellen Einheiten der Eigenschaften und strukturellen Einheiten des Modells in Form von μ-Kalkül-Formeln. Diese verknüpfen den einzelnen Schritt und die Möglichkeit der Wiederholung zu Pfadstrukturen, wobei sich ein Beleg in gerade der Angabe eines

solchen Pfades ausdrückt. Insgesamt bleibt zusammenfassend festzuhalten, dass es dem $\diamond$-Spieler leicht fällt, Belege für seine Formeln zu finden, da diese selbst bei unendlichen großen Modell- resp. Formelstrukturen stets endlich bleiben.

Analog verhält es sich für den $\Box$-Spieler, dessen Kompetenz nun gerade darin besteht, für konjunktive Operatoren Gegenbeispiele zu finden. Gerade der konjuntivische Charakter der Operatoren untermauert die Anfälligkeit dieser Operatoren für fehlerhafte Teilkomponenten. Dies können zum einen Teilformeln, die nicht erfüllbar sind und demnach sofort die gesamte Eigenschaft widerlegen, Nachfolger, denen eine gewisse Eigenschaft fehlt, oder aber Pfadstrukturen sein, deren Existenz zu einem Fehlschlagen der Eigenschaftsvalidierung führen. Analog zum $\diamond$-Spieler lässt sich die Gewinnmotivation des $\Box$-Spielers zusammenfassend als *die Formeln mittels endlicher Gegenbeispiele widerlegend* beschreiben.

Entsprechend ihrer Motivation ziehen die beiden Spieler im Spielgraphen in Übereinstimmung mit der Abhängigkeit innerhalb der Formel, die zu widerlegen bzw. belegen ist, also entweder entlang der Formelstruktur zu Teilformeln, innerhalb des Modells zu Nachfolgern, oder aber einen Pfad etablierend, zu Fixpunkten und deren Formeln.

Das Ergebnis für das ursprüngliche Model Checking Problem entsteht nun also aus der Information, welcher Spieler von welchen Knoten aus gewinnt, und welcher verliert. Da der $\diamond$-Spieler in diesem Szenario die Rolle des Beweisers übernimmt, der sich dafür verantwortlich zeichnet, die Teilargumente der Formel, die er zu belegen hat, nachzuweisen, sind gerade jene Modellknoten aus dem Spielgraphen in der Semantik einer zu belegenden Formeln, wenn er von einem Spielgraphknoten mit eben einer solchen Formel das Spiel gewinnt. Gibt es also z.B. im Spielgraphen einen Knoten (s, ϕ), der aus einer Teilformel ϕ und einem Modellknoten s des ursprünglichen Problems entstanden ist, von dem aus der $\diamond$-Spieler gewinnt, so ist der Knoten s des Modells in der Semantik von ϕ unter Berücksichtigung dieses Modelles enthalten. Insbesondere gewinnt man hierdurch also nicht nur Informationen für das ursprüngliche Problem – nämlich ein Modell gegen eine Formel zu prüfen – als auch die Teilprobleme, die die jeweiligen *Teilformeln* gegen das Modell prüfen.

Gelingt es dem $\diamond$-Spieler nicht, für eine seiner Teilformeln resp. einen seiner Nachfolger eine Eigenschaft nachzuweisen, so verliert er von einem solchen Knoten aus das Spiel und der $\Box$-Spieler gewinnt, weil es ihm nun möglich ist, alle Teilformeln, die der $\diamond$-Spieler eigentlich hätte belegen müssen, zu widerlegen.

Das Ergebnis für die eine zu untersuchende Eigenschaft erfüllenden Knoten eines Modelles manifestiert sich also in gerade jenen Knoten, von denen aus der $\diamond$-Spieler gewinnt (und zwar unabhängig von den Entscheidungen des $\Box$-Spielers). Andererseits ergeben sich die Knoten, die eine solche Eigenschaft nicht besitzen, wiederum aus den Knoten im Spielgraphen, von denen aus der $\Box$-Spieler gewinnt (und zwar auch wiederum unabhängig von der Wahl der $\diamond$-Spielers).

4.3.1 Besonderheiten für $\exists\mu$

Der vorherige Abschnitt hat das Spielergebnis auf das Model Checking Ergebnis reduziert und somit ermöglicht, Model Checking als ein Zwei-Personen-Spiel auffassen zu können. Da die letzten Abschnitte häufig davon ausgegangen sind, dass der Spielgraph, der der Fragestellung zugrunde lag, wiederum einer speziellen μ-Kalkül-Formel (eben einer solchen aus $\exists\mu$) zugrunde lag, soll dieser Abschnitt noch einmal die besonderen Auswirkungen auf das entstehende Zwei-Personen-Spiel beleuchten.

Da der Spielgraph aus einer Formel aus $\exists\mu$ entstanden ist, sind besondere Bedingungen, die sich aus der Alternierung von Fixpunkten ergeben, nicht zu berücksichtigen, da eine solche Formel keine größten Fixpunkte zulässt und somit lediglich Aussagen über (höchstens) kleinste Fixpunkte tätigen kann. Ebenso spielt der $\Box$-Spieler in diesem Szenario keine weitere Rolle, sind gerade dessen Operatoren doch aus der Logik entfernt worden. Dennoch bleibt es ein *Zwei*-Personen-Spiel, ist es dem $\Diamond$-Spieler doch immer noch möglich zu verlieren, wenn er in eine Senke zieht, von der aus er ziehen müsste, es jedoch mangels ausgehender Kanten nicht kann. Dies ist etwa bei Formeln $\Diamond\phi$ der Fall, die nur belegbar sind (und gerade das ist ja die Aufgabe des $\Diamond$-Spielers), wenn entsprechende Kanten im ursprünglichen Modell, aus dem der Spielgraph erzeugt wurde, vorhanden waren. Hieraus ist wiederum leicht zu erkennen, dass sich die Motivation des $\Diamond$-Spielers in dieser neuen Gestalt des Spieles darauf beschränkt, Wege zu Knoten zu finden, an denen der $\Box$-Spieler nicht weiter ziehen kann. Gerade solche Knoten sind entweder Senken im Spielgraph, die auf atomare Propositionen, welche wiederum gültig im jeweiligen Knoten des zu überprüfenden Modells sind, oder aber $\Diamond$-Teilformeln der eben beschriebenen Gestalt verweisen. Gerade dies sind die elementaren Einheiten, die es für den $\Diamond$-Spieler zu belegen gilt, sollen weitere Eigenschaften, in die diese Proposition eingebettet ist, nachgewiesen werden. Im Besonderen ist es dem $\Diamond$-Spieler also weiterhin durchaus möglich, zu verlieren, wenn etwa bestimmte Kanten im ursprünglichen Modell fehlen und damit die Etablierung eines Pfades verwehren, oder aber alle möglichen Wege den $\Diamond$-Spieler in Senken führen, an denen er ziehen müsste.

Hieraus lässt sich damit leicht erkennen, dass der Beleg einer Formel aus $\exists\mu$ gerade darin besteht, einen geeigneten Pfad im Spielgraphen zu einer Senke des $\Box$-Spielers zu finden. Insbesondere ist dem $\Box$-Spieler durch die Einschränkung der Logik die beeinflussende Möglichkeit auf den Spielverlauf genommen. Der $\Box$-Spieler wird weiterhin ausschließlich zur Bestimmung des Gewinners benötigt.

Kapitel 5

Pläne

Nachdem das letzte Kapitel ausführlich Auskunft über die Erstellung und Benutzung von Spielgraphen zur Informationsgewinnung von Model Checking Ergebnissen gegeben hat, soll dieses Kapitel diese bisher nurmehr einfachen Aussagen zu nicht eindeutigen Ergebnissen dahingehend verallgemeinern, dass nun nicht nur *ein* Ergebnis als Antwort auf eine Model Checking Anfrage ausreichen soll, sondern man vielmehr an *allen* möglichen Lösungen interessiert ist. Da die Menge möglicher Lösungen nicht zwangsläufig endlich ist, ist demnach auch keine explizite Angabe (im Sinne einer Aufzählung) tatsächlich aller Lösungen denkbar. Dieser Umstand schließt jedoch nicht aus, dass es durchaus sinnvolle[1] endliche Teilmengen dieser möglicherweise unendlichen Grundmenge gibt, die eine explizite endliche Repräsentation besitzen. Vielmehr soll das Interesse nun also auf bestimmte Kategorien von Lösungen gelenkt werden, die für das gegenwärtige Model Checking Problem relevant erscheinen.

Doch wie kommt es zu dieser Vielzahl von Lösungen? Ein einfaches Beispiel soll hier Klarheit schaffen. Hierzu ziehe man den noch später genauer zu betrachtenden Graphen aus Abbildung 5.17 auf Seite 80 heran. Welche Lösungen gibt es in diesem konkreten Fall für die Eigenschaft, aus Knoten a startend auf irgendeinem Weg zu Knoten d zu gelangen? Neben den direkten Wegen, sind sicher auch Wege, die mehrfach die Knoten b und c besuchen, bevor sie zu Knoten d gelangen, gültig. Insbesondere sind sogar alle endlichen Wege gültige Lösungen, die beliebig häufig diese beiden Knoten besuchen, solange schließlich der Knoten d erreicht wird. Es ergibt sich also eine unendliche Vielzahl von Lösungsmöglichkeiten bereits für dieses einfache Beispiel. Die Vielzahl von Lösungen resultiert demnach maßgeblich aus der Existenz von Kreisen – es lässt sich damit auch die Vermutung ableiten, dass die Anzahl von Kreisen in einem Graph demnach maßgeblich verantwortlich für die Mächtigkeit einer jeden nicht-trivialen endlichen Beschreibung ist.

Andererseits ist auch zu erkennen, dass nach einer gewissen Zeit, keinerlei neue

[1] „Sinnvoll“ beschreibt in diesem Zusammenhang eine Auswahl von Pfaden, die Wiederholungen in Belegen zwar zulässt, jedoch nur, wenn seit der letzten Wiederholung weitere neue Knoten aufgetaucht sind.

Informationen über den Lösungsweg offenbart werden. Es lassen sich also Strukturen erkennen, nach denen die Lösungen aufgebaut zu sein scheinen. Was in diesen kleinen Beispielen noch unmittelbar zu erkennen ist, vermag in größeren Beispielen jedoch für Verwirrung sorgen. Man ist also an einer intuitiven Spezifikation einer (möglicherweise endlichen) Teilmenge dieser Lösungen oder aber an der Menge aller Lösungen interessiert, die durch Nutzerinteraktion erkundet und verstanden werden kann.

Die Grundlage für die Aufzählung aller möglichen Lösungen wird ein gemäß der Beschreibung des vorherigen Abschnitts erzeugter Spielgraph sein. Aus diesem lässt sich bereits durch eine einfache rückwärts gerichtete Breitensuche von den Senken des $\Box$-Spielers aus eine *implizite* Beschreibung aller Gewinnmöglichkeiten des $\diamond$-Spielers, und damit aller Belege der beschriebenen Liveness-Eigenschaft auf dem untersuchten Modell, gewinnen.

Ein mächtiges Werkzeug zur Erzeugung endlicher Teilstrukturen aus impliziten Beschreibungen möglicherweise unendlicher Grundstrukturen, stellt das Property Oriented Expansion-Verfahren dar, das im nun folgenden Abschnitt vorgestellt werden soll. Es fungiert an dieser Stelle als eine Art Filter, der aus unendlichen implizit beschriebenen Beschreibungen diejenigen endlichen Teilstrukturen extrahiert, die gemäß einer funktionalen Eigenschaft von Interesse sind.

5.1 Property Oriented Expansion

Transformer
Annotationen

Property Oriented Expansion (im Folgenden kurz POE genannt) beschreibt ein bereits 1996 von Bernhard Steffen (vgl. [Ste96]) vorgestelltes Verfahren zur Expansion von Graphen auf Grundlage einer gegebenen Funktion – dem so genannten *Transformer*. Hierbei wird der zu expandierende Graph mit Initialwerten (im Folgenden *Annotationen* genannt) an bestimmten Knoten der Graphstruktur versehen und anschließend diese Intialknoten, die bestehende Graphstruktur respektierend, mit Hilfe des Transformers expandiert. Dabei geht der Algorithmus wie folgt vor.

1. Initialisiere eine Menge von Knoten mit Annotationen.
2. Wähle einen noch nicht bearbeiteten Knoten n mit Annotation A, und bezeichne ihn mit (n, A).
3. Bestimme für alle Nachfolger n' von n und den Transformer t die Werte $t(n)(A)$.
4. Erzeuge Kanten von (n, A) nach $(n', t(n)(A))$.
5. Markiere (n, A) als bearbeitet.
6. Wenn noch unbearbeitete Knoten existieren, gehe zu Schritt 2, ansonsten beende den Transformationsprozess.

Das beschriebene Vorgehen, einer Kontrollstruktur folgend und dabei einen Zustand respektierend, kann in dieser Art also ebenso als eine Interpretation des

Graphen im Sinne einer Ausführung gesehen werden. Die Annotationen nehmen in diesem Zusammenhang die Rolle eines globalen Zustandes ein, der von Knoten zu Knoten gereicht und dabei verändert wird. Der Algorithmus selbst beschreibt eine Schrittsteuerung auf dem zu expandierenden Graphen. Der Transformer schließlich realisiert die eigentliche Funktionalität. In Abschnitt 5.1.4 wird eben diese Interpretation (die jedoch nicht die einzige Möglichkeit ist, das Verfahren zu benutzen, wie auch die weiteren Beispiele dieses Abschnittes zeigen werden) genutzt, um das POE-Verfahren selbst zu beschreiben.

Die folgenden Abschnitte zeigen beispielhaft unterschiedliche Verwendungsmöglichkeiten von POE. Das erste Beispiel beschreibt die Möglichkeit, für bestimmte Eingabeinstanzen einer Funktion durch Expansion einen Graph zu erzeugen, der gerade die Zuordnung dieser Eingabeinstanzen zu ihren evaluierten Werten vornimmt und die eigentliche Berechnung der Funktion hiermit überflüssig macht. Wird für die Formel eine boolsche Funktion angenommen, so lassen sich auf diese Weise (natürlich nicht notwendigerweise minimale) BDDs[2] erzeugen. Aber auch im Umfeld von Meta- und Makro-Programmierung hat dieses Beispiel seine natürlichen Anwendungsfelder. Das zweite Beispiel beschreibt die Bestimmung von Fixpunkten einer Funktion durch Expansion eines Graphen. Hierbei wird Newtons Iterationsverfahren zur Bestimmung von Wurzeln benutzt, welches sukzessive die Wurzel als Fixpunkt einer bestimmten Mittelwertbildung nach und nach annähert. Ein weiteres Beispiel beschreibt die automatische Spielgraphgenerierung (wie sie bereits in Abschnitt 4 vorgestellt wurden) für eine eingeschränkte Logik. Ein Beispiel nutzt POE als Verfahren, Kontrollflussgraphen zu interpretieren und somit etwa das POE-Verfahren selbst auszuführen. Schließlich wird in einem weiteren Abschnitt das oben beschriebene Verfahren in einen applikativen Kontext eingebettet, um es damit für weitere formale Betrachtungen zugänglicher zu machen.

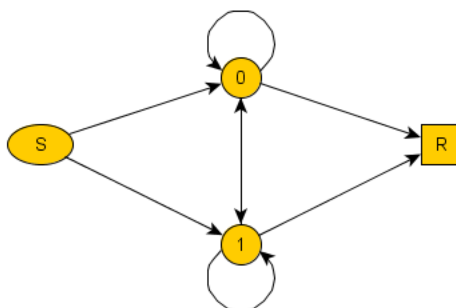
Die im Folgenden verwendeten Transformer werten unter gewissen Umständen zu $\perp$ aus. Dieser Wert kann als rein symbolischer Wert aufgefasst werden, der als unerwünschte Expansion gedeutet wird. Am Ende einer vollständigen Graphexpansion werden Knoten mit einer derartigen Annotation entfernt.

5.1.1 Beispiel 1 – Eingabenvalidierung von Funktionen

Ein Beispiel für die Validierung von Funktionen auf bestimmten Eingaben soll das folgende Beispiel verdeutlichen. Hierbei ist eine Funktion f gegeben, die untersucht werden soll. Ferner beschreibt der Graph in Abbildung 5.1, auf welche Eingabe hin die Funktion getestet werden soll. Der Knoten S beschreibt hierbei einen Startzustand, von dem ausgehend die Eingaben 0 und 1 in beliebiger Reihenfolge und Anzahl gesammelt werden, bis sie schließlich im Knoten R ausgewertet werden.

Der Transformer, der das Aufsammeln der Argumente, wie auch ihre Auswertung übernimmt, ist ebenfalls in Abbildung 5.1 zu sehen. Solange der Auswertungsknoten R nicht erreicht ist, sammelt der Transformer die Argumente

[2]BDD = Binary Decision Diagram



$$t(n)(f,i) = \begin{cases} f(i) & \text{falls } n = \mathbf{R} \\ i : n & \text{sonst} \end{cases}$$

Abbildung 5.1: Eingabenvalidierung einer Funktion auf 0 und 1.

in einer Liste i auf („:“ stehe hierbei für die Listenkonkatenation), bis er sie schließlich am Auswertungsknoten R evaluiert.

Abbildung 5.2 zeigt als Beispiel die Evaluierung einer boolschen *und*-Funktion auf Eingaben 0 und 1, die für die jeweiligen Wahrheitswerte *true* und *false* stehen. Der Graph ist zwar nach unten hin unbeschränkt und wächst mit steigender Operandenzahl beliebig, doch lässt sich der Transformer leicht auch auf eine endliche feste Operandenzahl anpassen.

Der entstandene Graph ermöglicht nun weitere Untersuchungen der Funktion – etwa hinsichtlich der Erfüllbarkeit. Fasse man den expandierten Graphen als ein Kripke Transitions System und die Annotation an den Knoten als Propositionen auf, so beschreibt ein Weg vom Knoten S zu einem Knoten mit der Propositionen *true* eine Belegung der Funktion, die zu gerade eben diesem Ergebnis auswertet. Ist es nun ferner möglich, alle solche Wege zu einer erfüllenden Belegung aufzuzählen (und gerade dies wird diese Arbeit in späteren Abschnitten zeigen), so erhält man den Kern der Funktion f. Ebenso ist die Funktion f selbst nun nicht weiter notwendig, wenn sie lediglich auf Eingaben 0 und 1 definiert ist, da bereits der resultierende Graph die Funktion und ihr Verhalten vollständig beschreibt (was hier jedoch nicht weiter gezeigt werden soll).

Es gibt sicherlich andere zeit- und auch platzeffizientere Verfahren als POE für die Berechnung des Kerns oder aber Auswertung einer boolschen Funktion – wie etwa OBDDs. Vielmehr zeigt die Wahl des Beispiels, dass POE seinem Wesen nach nicht auf Mengenannotation und -behandlung beschränkt ist und, werden diese Zeit- und Platzeinschränkungen akzeptiert, somit, ähnlich den *constraint satisfaction problems*, logischen Problembeschreibungssprachen wie Prolog oder aber generell deklarativen Programmiersprachen, als allgemeines Problembeschreibungsverfahren, dessen Lösung lediglich durch geeignete Wahl von Expansiongraph und Transformer bereits ausreicht, um eine algorithmische Lösung des spezifizierten Problems zu erhalten. Nicht die Lösungs- als vielmehr die Problemmodellierung stehen hiermit im Vordergrund. Lässt sich ein Problem

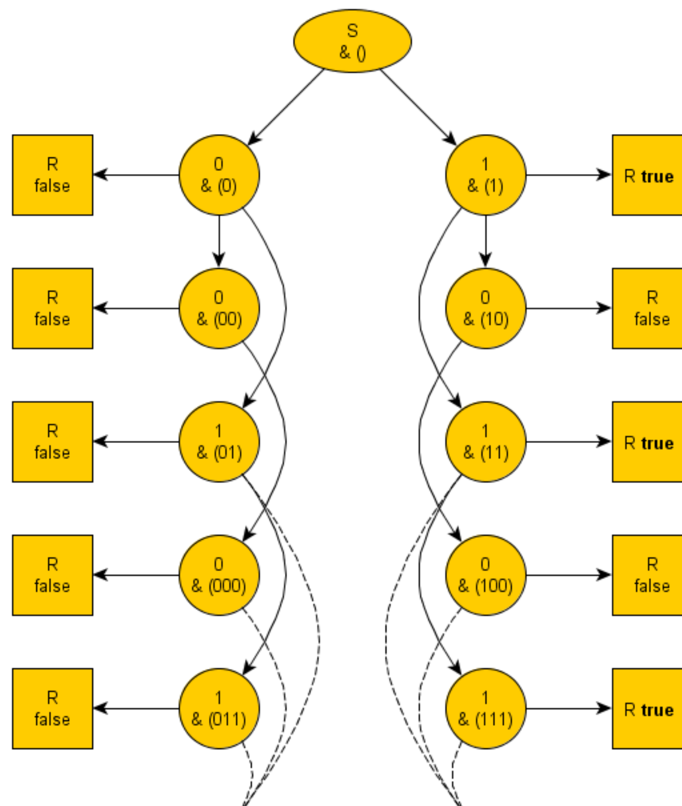


Abbildung 5.2: Validierung von Eingaben für eine boolsche *und*-Funktion.

in eine solche Problembeschreibungssprache überführen, die es ermöglicht das Problem *automatisch* zu lösen, so profitiert jedes auf solche Art beschriebene Problem von einer „Verbesserung" – meist einer Zeit- oder Platzoptimierung – des zugrundeliegenden Verfahrens (in diesem Falle POE).

5.1.2 Beispiel 2 – Newtonverfahren zur Fixpunktberechnung

Eine weitere Anwendung von POE, ist das Iterationsverfahren von Newton zur Bestimmung von Wurzeln. Im Allgemeinen dient dieses Iterationsverfahren – unter gewissen Voraussetzungen – der Bestimmung von Fixpunkten. Genau eben diese Fixpunkte einer Gleichung werden durch das Verfahren nämlich ermittelt. Die Gleichung, deren Fixpunkt bestimmt werden soll, ist die folgende.

$$f(S) = \frac{1}{2}\left(\frac{n}{S} + S\right)$$

Ein Fixpunkt dieser Gleichung ist gerade $\sqrt{n}$, denn es gilt $f(\sqrt{n})^2 = n$. Dies folgt aus

$$\begin{aligned} f(\sqrt{n})^2 &= \\ \left(\frac{1}{2}\left(\frac{n}{\sqrt{n}} + \sqrt{n}\right)\right)^2 &= \\ \left(\frac{1}{2}\frac{2n}{\sqrt{n}}\right)^2 &= \\ \left(\frac{n}{\sqrt{n}}\right)^2 &= n \end{aligned}$$

Als Spezialfall dieser Gleichung gilt also ebenso für $n = 2$, dass $f(\sqrt{2}) = \sqrt{2}$. Der Wert von S in der Ursprungsgleichung beschreibt eine Art Startwert, von dem ausgehend der Iterationsprozess begonnen werden kann, um sich schließlich nach und nach den Fixpunkt anzunähern, indem die Folge der Werte $f(S), f(f(S)), f(f(f(S), \ldots$ berechnet wird. Die folgenden Gleichungen zeigen einen solchen Annäherungsprozess für den Startwert 1.

$$\begin{array}{ll} f(1) & = 1.5 \\ f(f(1)) = f(1.5) & = 1.417 \\ f(f(f(1))) & = 1.414 \\ & \ldots \end{array}$$

Dieser Prozess ist jedoch ebenso mittels POE und einem geeigneten Graphen wie auch einem geeigneten Transformer denkbar. Sei etwa der Graph aus Abbildung 5.3 mit dem folgenden Transformer zu expandieren.

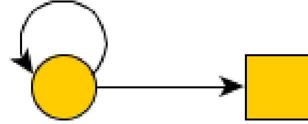


Abbildung 5.3: Der zu expandierende Graph für Newtons Iterationsverfahren

$$\begin{array}{ll} \forall\circ: & f(S) = \frac{1}{2}\left(\frac{2}{S} + S\right) \\ \forall\Box: & f(S) = S^2 \end{array}$$

Hierbei schreitet beim „Abwickeln“ des Graphen an den Kreisknoten der Iterationsprozess fort, während ein Verfolgen der Boxknoten die jeweilige Probe im Sinne eines Quadrierens auf das bisherige Expansionsergebnis macht. Das Ergebnis dieses Expansionsprozesses – und damit auch des Iterations- und Fixpunktbestimmungsprozesses – mit einer Initialisierung von 1 im Kreisknoten zeigt der Graph in Abbildung 5.4.

In diesem Fall handelt es sich sogar um einen unendlichen Graphen (die fehlenden Knoten und Kanten sind durch die gestrichelte Kante angedeutet). Man kann also nicht von einer Terminierung des POE-Verfahrens für beliebige Eingaben ausgehen. Wann eine solche Terminierung jedoch zu erwarten ist, beschreibt der im Anhang unter B.1 vorgestellte Satz über die Wohldefiniertheit von POE unter bestimmten Voraussetzungen – die im Speziellen in diesem Fall nicht gegeben sind.

5.1.3 Beispiel 3 – Spielgraphgenerierung auf Mμ

In Abschnitt 4 wurden Spielgraphen bereits für Formeln der reduzierten Logik $\exists\mu$ vorgestellt (vgl. Definition 1). Beschränkt man die Logik für die temporalen Eigenschaften in einer anderen Richtung, sowie die Modelle, die mit diesen Eigenschaften zu einem Spielgraphen verschmolzen werden sollen, so lässt sich dieser Prozess der Erstellung des Spielgraphen auch durch einen geeigneten Transformer mittels POE durchführen.

Die Beschränkung der Modelle (in Form eines KTS) besteht darin, für alle Knoten des Modells reflexive Kanten anzunehmen. Diese Einschränkung ist notwendig, wenn man sich den Spielgraphen als einen vom Modell aus entlang der Abhängigkeiten der Formel entrollten Graphen vorstellt. Weitere Einschränkungen sind jedoch nicht notwendig.

Die Einschränkung der Logik besteht darin, lediglich *modellabhängige* Operatoren zuzulassen. Darunter sind gerade jene Operatoren zu verstehen, die lediglich im Kontext eines Modelles ausgewertet werden können und auch nur zusammen mit einem solchen eine Semantik besitzen. Eine auf Operatoren, deren Auswer-

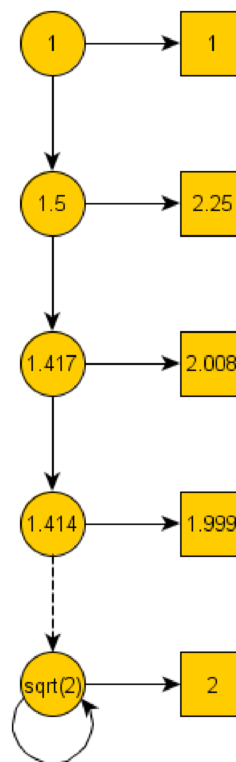


Abbildung 5.4: Der expandierte Graph für Newtons Iterationsverfahren

$$
\begin{aligned}
f(n')(n, \delta\phi) = & \quad (n', \phi) \text{ für } \delta \in \{\Box, \diamond\} \\
f(n')(n, \sigma\phi) = & \left\{ \begin{array}{ll} (n', \phi) & \text{falls } n = n' \\ \bot & \text{sonst} \end{array} \right. \quad \text{für } \sigma \in \{\mu, \nu\} \\
f(n')(n, X) = & \left\{ \begin{array}{ll} (n, bound(X)) & \text{falls } n = n' \\ \bot & \text{sonst} \end{array} \right. \\
f(n')(n, AP) = & \quad \bot
\end{aligned}
$$

Abbildung 5.5: Transformer zur Expansion eines Modelles in einen Spielgraphen

tung vom Modell abhängt, also auch die propositionalen Operatoren der klassischen Logik ausschließt, eingeschränkte Logik werde im Folgenden $M\mu$ bezeichnet, was auf Operatoren des modalen μ-Kalkül hindeutet, deren Auswertung in Abhängigkeit vom Modell geschieht. Operatoren dieser Logik umfassen also insbesondere kleinste und größte Fixpunkte μ, ν und deren zugehörige Fixpunktvariablen (ebenso ist auch eine Alternierung nicht ausgeschlossen), die modalen Operatoren $\Box$ und $\diamond$, sowie atomare Propositionen. Es sind also gerade alle jene Operatoren in dieser reduzierten Logik enthalten, die die in der Beschreibung des Modells als Kontrollflussstruktur immanent enthaltenen prozesshaften bzw. imperativen Aspekte kapseln.

$M\mu$

Ausdrücklich nicht in der Logik verankert sind demnach die Operatoren der propositionalen Logik $\vee$ und $\wedge$. Dies ist dem Umstand zu verdanken, dass diese Operatoren im Spielgraphen eine Abhängigkeit lediglich entlang der Formel, nicht jedoch in Richtung des Modelles verfolgen. Ein Abrollen des Modelles – von dem nur reflexive Kanten an jedem Knoten gefordert werden sollen – scheitert also gerade an der Arität dieser Operatoren. Denn gerade diese fordert i.A. mindestens zwei abhängende Teilformeln. Um diesem Umstand Rechnung zu tragen, müsste der Expansionsprozess also jede dieser Teilformeln an den Knoten berücksichtigend unterscheiden können und somit entsprechend der maximalen Arität der Operatoren $\vee$ und $\wedge$ ebensoviele reflexive unterscheidbare Kanten an jedem Knoten fordern. Ein mögliches Currying[3] zur Anpassung der Arität der propositionalen Operatoren ist im Allgemeinen zwar möglich, soll an dieser Stelle aus Gründen der Übersicht jedoch ausbleiben.

Der Expansionsprozess startet mit der Initialisierung eines Tupels, bestehend aus dem Knoten selbst und einer Formel aus $M\mu$ an allen Knoten des zu expandierenden Modells. Der dieser Expansion zugrunde liegende Transformer ist in Abbildung 5.5 dargestellt. Hierfür wird angenommen, dass die Menge der Fixpunktvariablen und die Menge der atomaren Propositionen einen leeren Schnitt besitzen. Ebenso wird angenommen, dass alle in der Formel vorkommenden Fixpunktvariablen verschieden sind. Die Funktion *bound* bestimmt für eine gegebene Fixpunktvariable diejenige Teilformel, an die sie gebunden ist. So gilt etwa für eine Formel $\diamond(\mu X. \diamond (\nu Y.Y))$, dass $bound(Y) = \nu Y.Y$. Eine mögliche Implementierung dieser Funktion befindet sich im Anhang in Abschnitt A.4.

[3] *Currying* beschreibt, in Anlehnung an Haskell Curry, wenngleich das Verfahren zuvor bereits durch Moses Schönfinkel und Gottlob Frege etabliert wurde und deshalb auch häufig als *Schönfinkeln* bezeichnet wird, die Umwandlung einer Funktion auf mehreren Argumenten in eine Funktion (genauer: eine Closure) auf lediglich einem Argument. Vgl. dazu etwa die sehr schöne und verständliche Übersicht [Sch24].

Abbildung 5.6: Beispiel eines Modelles und einer Formel aus $M\mu$, die mittels POE in eine Spielgraphen expandiert werden sollen.

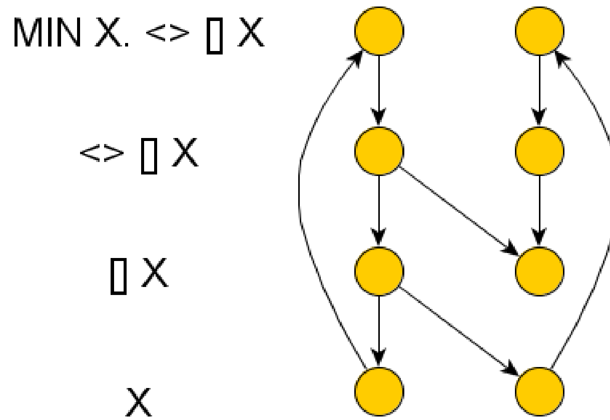


Abbildung 5.7: Der mittels POE expandierte Spielgraph.

Nach der Expansion werden alle Knoten mit der Annotation $\perp$ aus dem Graphen entfernt und es verbleibt der Spielgraph. Die Einteilung der Knoten in disjunktive ($V_\diamond$) und konjunktive ($V_\square$) Knoten wurde aus Gründen der Übersicht an dieser Stelle ausgelassen, ist jedoch durch kleine Änderungen des Transformers aus Abbildung 5.5 ebenfalls leicht zu realisieren bzw. durch die Annotationsinformation bereits implizit in den Knoten kodiert ist.

Ein Beispiel für die Anwendung des Verfahrens zeigen die Abbildungen 5.6 und 5.7. Während in Abbildung 5.6 die Eingabe für den Expansionprozess, bestehend aus dem zu expandierenden Modell und einer Formel aus $M\mu$ zeigt, stellt Abbildung 5.7 den expandierten Spielgraphen bereits nach Entfernung der unerwünschten $\perp$-Senken dar.

Interessant ist in diesem Zusammenhang der Umstand, dass die reduzierte Logik $M\mu$ viel von der Ausdrucksstärke des modalen μ-Kalkül verloren hat und die Formeln meist nur sehr lokale Eigenschaften oder aber nur eine leere bzw. volle Semantik zulassen, obwohl die Logik lediglich um scheinbar „nur“ propositionale Eigenschaften erleichtert wurde. Dies trägt wohl dem Umstand Rechnung, dass die Fixpunkte durch das Wegfallen von $\vee$ und $\wedge$ viel von ihrer Ausdrucksstärke verloren haben. Gerade diese Operatoren bilden analog zur vollständigen Induk-

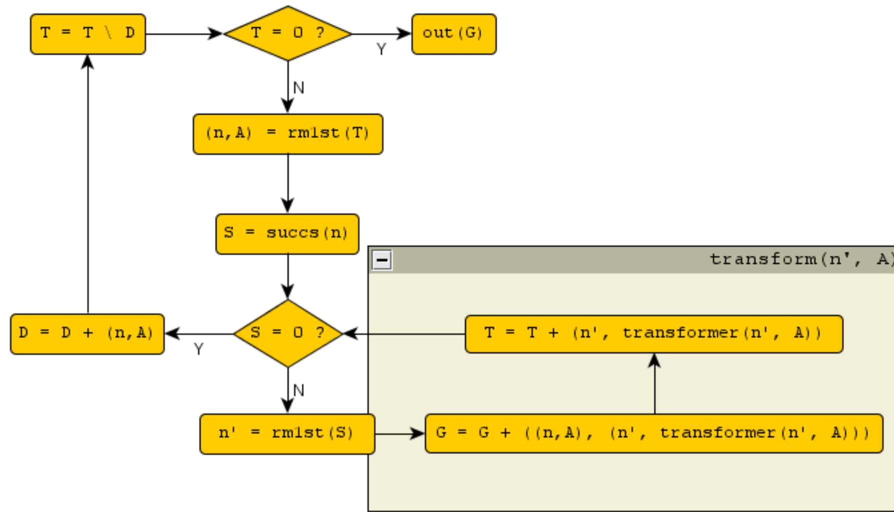


Abbildung 5.8: Kontrollflussgraph für Property Oriented Expansion

tion die Induktionsbasis bzw. analog zu rekursiven oder iterativen Prozessen die Abbruchbedingung, deren Wegfallen zu tautologischen Alles-oder-Nichts- bzw. Ja-oder-Nein-Aussagen führen. Der Logik wurde somit die formale Grundierung der Verallgemeinerungs- und Strukturierungsmöglichkeit ihrer Aussagen genommen.

5.1.4 Beispiel 4 – POE

Das vierte Beispiel schließlich zeigt eine Anwendung von POE auf Kontrollflussgraphen imperativer Programme. Im Sinne einer operationellen Semantik lässt sich die Semantik eines solchen Programmes durch das Abrollen des Kontrollflussgraphen beschreiben. Dieses Abrollen wird jedoch gerade durch POE beschrieben. Die Annotationen an den Knoten übernehmen hierbei die Rolle einer Umgebung, die während des Programmausführung (während des Expansionsprozesses) durch die Aktionen an den Knoten befragt und verändert wird.

Da die Wahl des Kontrollflussgraphen nicht weiter eingeschränkt ist, ist insbesondere auch eine an Kontrollflussgraphen orientierte Beschreibung des POE-Verfahrens selbst zulässig, um das Verfahren aus sich selbst heraus zu gewinnen bzw., um es auf sich selbst anzuwenden. Eine solche imperative Beschreibung des in Abschnitt 5.1 im Pseudocode vorgestellten Verfahrens, zeigt der Kontrollflussgraph in Abbildung 5.8. Dieser besteht aus zweierlei Schleifen, die jeweils zum einen noch nicht bearbeitete Knoten, zum anderen für jeden zu expandierenden Nachfolger eine Transformation durchführen und somit sukzessive den Graphen expandieren.

Dieser Kontrollflussgraph soll nun dazu dienen, die Semantik, die er beschreibt

– nämlich gerade, einen Graphen mittels POE zu expandieren – mittels POE zu bestimmen. Interessant hierbei ist die Tatsache, dass der Kontrollflussgraph das Verfahren beschreibt, mit dem er expandiert werden soll. Ist also erst einmal eine beliebige Implementierung von POE vorhanden, so lässt sich durch die Expansion des Graphen in der Abbildung eine an Kontrollflussgraphen orientierte Realisierung gewinnen.

POE selbst kennt während der Transformation keinerlei globalen Zustand und führt die jeweilige Transformation lokal an einem Knoten unter Berücksichtigung der bisherigen Transformationsinformationen durch. Der globale Zustand lässt sich jedoch durch eine geeignete Umgebung ψ simulieren, die, bestehend aus einem Tupel der in der Umgebung enthaltenden Variablen, von den jeweiligen Knoten nur lokal benutzt, verändert bzw. ausgewertet wird. Für den gesamten Transformationsprozess von POE besteht diese Umgebung aus einem zu transformierenden Graphen G^*, einem resultierenden Graphen G, einem zu transformierenden Knoten n aus einer Menge noch zu transformierender Knoten T und seiner zu transformierenden Nachfolger n' aus einer Nachfolgermenge S, einer Transformationsannotation A, einer Menge bereits transformierter und somit nicht weiter zu betrachtender Knoten D und einem Property-Transformer λ. Diese Umgebung ψ besitzt somit die Gestalt $\psi = (G^*, G, T, D, n, A, n', S, \lambda)$. Zusammen mit dem Kontrollflussgraphen aus Abbildung 5.8 liefert dies den Grundstein für eine Expansion dieses das POE-Verfahren selbst beschreibenden Graphen.

Es ist wichtig hierbei die Annotationen A und ψ wohl zu trennen. ψ ist hierbei die Umgebung, die als Annotation während der Transformation des Kontrollflussgraphen aus Abbildung 5.8 genutzt wird, um diesen mit Hilfe eines noch zu definierenden Transformers in einen Graphen abzurollen, der den Transformationsalgorithmus POE, der durch diesen Graphen beschrieben wird, in schrittweiser Annäherung an den ausgerollten Graphen, der bei Ausführung des Algorithmus entstehen würde, beschreibt. A dagegen ist eine Annotation, die genutzt wird, um mit Hilfe des Transformers λ den Graphen G^* nach und nach in einen Graphen G zu überführen. Es finden beim Abrollen des Kontrollflussgraphen also *gleichzeitig* zweierlei Graphexpansionen statt – zum einen die des Kontrollflussgraphen selbst, zum anderen die durch ihn beschriebene Graphexpansion.

Die Expansion des Kontrollflussgraphen, die durch den in Abbildung 5.9 gezeigten Transformer unterstützt und ermöglicht wird, realisiert im Wesentlichen die an den Knoten durchzuführenden Operationen. Letztendlich wäre es also ebenso möglich, die Knotennamen selbst als Datum (etwa als Funktion oder Codierung einer Funktion) aufzufassen und diese direkt anzuwenden. Dies würde den Transformer vereinfachen, der dann nur noch zwischen verzweigenden Kontrollflussknoten und den den globalen Zustand verändernden Ausführungsknoten unterscheiden müsste. Zu Wahrung einer gewissen Übersichtlichkeit, wird an dieser Stelle jedoch direkt auf eine umgangssprachliche Beschreibung der Knoten und der Realisierung ihrer Funktionalität im Transformer zurückgegriffen.

Die Transformation wird mit einer geeigneten Umgebung ψ im Knoten $\texttt{T} = \texttt{0}$ initialisiert. Dies sind im Wesentlichen die Eingaben für den POE-Algorithmus; also der zu expandierende Graph G^* und der Transformer λ, sowie die Initialisierung der Knoten in G^* mit T. Die Menge der bereits bearbeiteten Knoten D so-

$$
\begin{array}{ll}
f(\mathtt{T = 0})(\psi) = & \psi \\
f(\mathtt{out(G)})(\psi) = & \begin{cases} G & \text{falls } T = \emptyset \\ \bot & \text{sonst} \end{cases} \\
f(\mathtt{(n, A) = rm1st(T)})(\psi) = & \begin{cases} \psi[\mathrm{fst}_1(T) \mapsto n, \\ \quad \mathrm{fst}_2(T) \mapsto A, \\ \quad \mathrm{rest}(T) \mapsto T] & \text{falls } T \neq \emptyset \\ \bot & \text{sonst} \end{cases} \\
f(\mathtt{S = succs(n)})(\psi) = & \psi[\mathrm{succs}(n) \mapsto S] \\
f(\mathtt{S = 0})(\psi) = & \psi \\
f(\mathtt{D = D + (n, A)})(\psi) = & \begin{cases} \psi[D \cup (n, A) \mapsto D] & \text{falls } S = \emptyset \\ \bot & \text{sonst} \end{cases} \\
f(\mathtt{T = T \backslash D})(\psi) = & \psi[T \backslash D \mapsto T] \\
f(\mathtt{n' = rm1st(S)})(\psi) = & \begin{cases} \psi[\mathrm{fst}(S) \mapsto n', \mathrm{rest}(S) \mapsto S] & \text{falls } S \neq \emptyset \\ \bot & \text{sonst} \end{cases} \\
f(\mathtt{transform(n', A)})(\psi) = & \psi[G \cup \{(n, A) \to (n', \lambda(n', A))\} \mapsto G \\
& \quad T \cup (n', \lambda(n', A)) \mapsto T]
\end{array}
$$

Hierbei beschreibt die Funktion fst_i die Projektion auf die i-te Komponente des ersten Elementes einer Liste, die als Argument übergeben wird. $\mathrm{succs}(n)$ liefert die Nachfolgeknoten von n in G^*, $\mathrm{rest}(L)$ liefert die Liste L ohne das erste Element.

Abbildung 5.9: Transformer für den Kontrollflussgraphen in Abbildung 5.8

wie der expandierte Graph G werden leer initialisiert, die übrigen Werte bleiben undefiniert bis zu ihrer Initialisierung während der Expansion, so dass sich die Umgebung wie folgt ergibt $\psi = (G^*, G = \emptyset, T, D = \emptyset, n = \bot, n' = \bot, S = \bot, \lambda)$.

Im Unterschied zu den vorherigen Beispielen, erlaubt die soeben geschilderte Form der Expansion eine Art Debugging des Transformationsprozesses. So werden etwa die Informationen in den beiden Schleifen nicht etwa verworfen, sondern im abgerollten Graphen gesammelt und stehen damit auch nach der Ausführung weiterhin zur Verfügung. Es besteht also die Möglichkeit, den Transformationsprozess – oder eine andere in dieser Form repräsentierte imperative Berechnung – zeitlich in beiden Richtungen verfolgen zu können. Ferner ist es nun leicht, einen Zustand des Expansionsprozesses persistent zu halten und ggf. zu einem späteren Zeitpunkt an dieser Stelle fortzusetzen, da dieser bereits vollständig durch die Umgebung ψ zu diesem Zeitpunkt bestimmt ist. Das Verfahren könnte unter diesen Umständen also iterativ den resultierenden Graphen berechnen und nach und nach eine Lösung bestimmen. Erinnere man sich an das Beispiel der Wurzelbestimmung aus Abschnitt 5.1.2, so könnte das POE-Verfahren nun etwa auf Nutzerinteraktionen reagieren und nur bei Bedarf den Expansionsprozess fortsetzen.

Abschließend soll durch ein kleines Beispiel die Expansion erläutert werden. Einen auf die oben beschriebene Art expandierten Kontrollflussgraph zeigt Abbildung 5.10. Hierbei sind nur diejenigen Annotationen der Umgebung ψ angegeben, die sich zum vorherigen Knoten in mindestens einer ihrer Komponenten geändert haben. Der der Expansion zugrunde liegende Graph besteht aus zwei

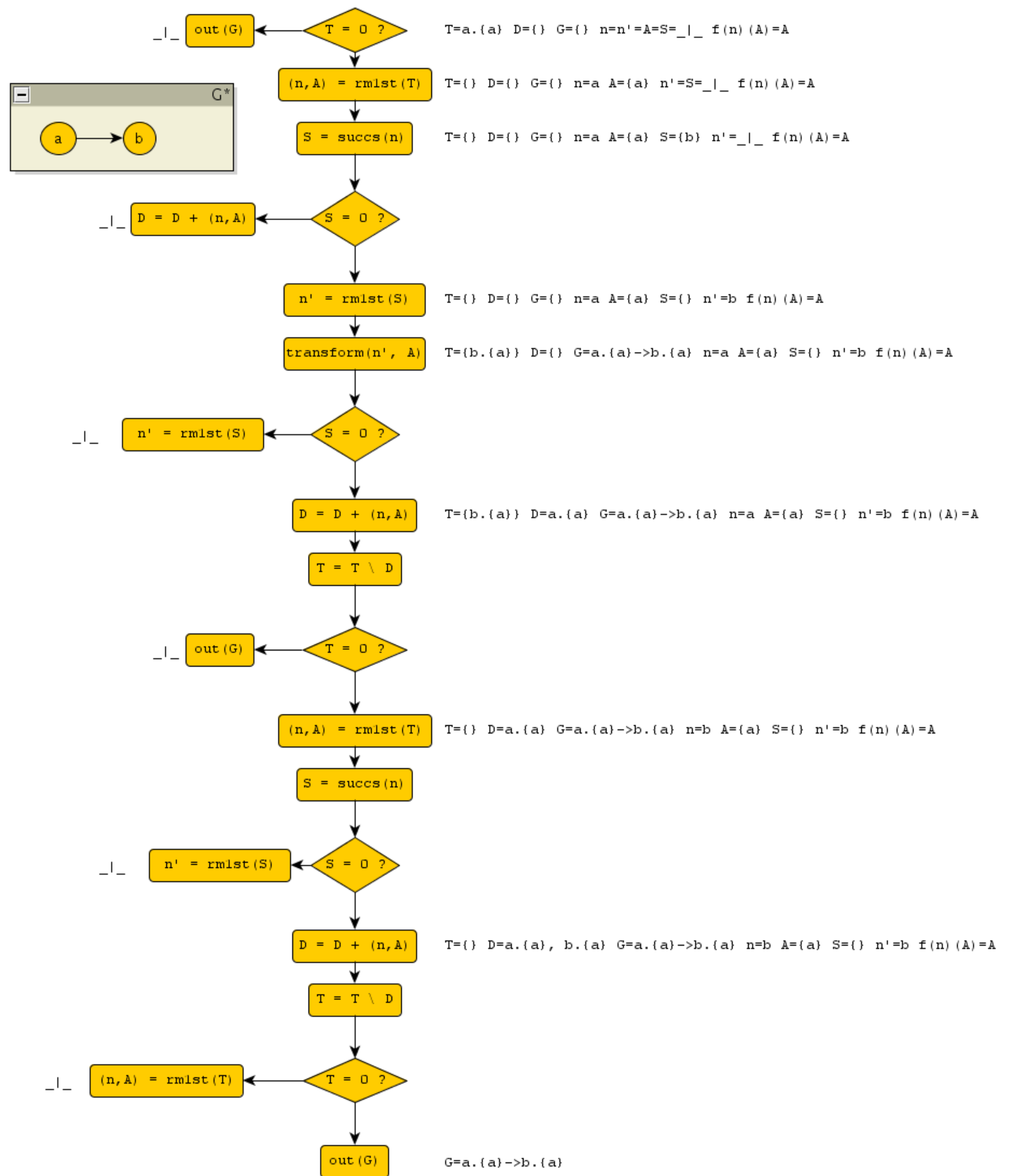


Abbildung 5.10: Expandierter Kontrollflussgraph für Property Oriented Expansion

Knoten a und b, sowie einer Kante von a nach b. Initilialisiert wird die Expansion mit dem Knoten a und einer Menge, die nur den Knoten a enthält. Der Transformer für die Expansion ist die Identität auf der Annotation. Werden wiederum alle Knoten mit der Annotation $\perp$ entfernt, so verbleibt ein Graph, der den Expansionsprozess modelliert und dessen Ergebnis sich im Knoten $out(G)$ befindet. Dies ist gerade ein Graph, der die Annotation in Knoten a in den Nachfolgeknoten b rettet und ansonsten die gleiche Gestalt wie der ursprüngliche Graph besitzt.

Die wohl längste Endlosschleife...

Im vorherigen Abschnitt wurde ein beliebiger Kontrollflussgraph durch einen den POE-Algorithmus beschreibenden Kontrollflussgraphen ersetzt, um mittels dieser Konstruktion POE aus dem POE-Verfahren selbst zu erhalten. Der Algorithmus bestand darin, einen mit einer Umgebung ψ annotierten Knoten $\mathtt{T} = \mathtt{0}$ mittels POE derart abzurollen, dass innerhalb dieser Umgebung ebenfalls ein Abrollprozess simuliert wurde. Doch was passiert, wenn der simulierte Expansionsprozess seinerseits einen Expansionsprozess simuliert? Dies soll nun dadurch untersucht werden, dass die Umgebung dahingehend spezialisiert wird, dass sie nicht einen beliebigen zu expandierenden Graphen in der Komponente G^* innerhalb der Umgebung ψ enthält, sondern eben G selbst. Ebenso wird die Annotation dieser simulierten Graphexpansion in der initialisierten Menge T abgeändert und durch ψ selbst ersetzt. Schließlich wird auch der beliebige Transformer λ durch den speziellen Transformer f aus Abbildung 5.9 ersetzt, so dass die resultierende Umgebung $\psi = (G^*, G, T, D, n, A, n', S, \lambda)$ die Gestalt bekommt $\psi = (G, G, \{(\mathtt{T} = \mathtt{0}, \psi\}, D, \perp, \perp, \perp, \perp, f)$ und der Knoten $\mathtt{T} = \mathtt{0}$ mit eben dieser Umgebung initialisiert wird.

Schnell ist er zu erkennen, dass dieser Expansionsprozess nicht terminiert. Denn wie läuft eine Expansion im vorliegenden Fall eigentlich ab? Im ursprünglichen Graphen wird eine Expansion angestoßen, welche über die Annotation wiederum eine Expansion startet, welche wiederum eine Expansion startet etc. Diese nicht endende Expansion resultiert maßgeblich aus der implizit unendlich rekursiven Initialisierung, die sich selbst in einer Komponente enthält. Dennoch ist in diesem Zusammenhang die Art und Weise des Zustandekommens der Endlosschleife interessant. Hier lässt sich ein Beispiel für eine implizite Beschreibung einer explizit unendlichen hierarchischen Struktur erkennen. Die Unendlichkeit entsteht nicht aus der Zusammensetzung von atomaren Einheiten, wie dies etwa bei den natürlichen Zahlen der Fall ist, oder aber aus einer immer wiederkehrenden sequentiellen Abfolge von Anweisungen, wie dies etwa in imperativen Sprachen durch Anweisung der Art `while`*true* { `skip;` } gewährleistet wird. Hier entsteht die Unendlichkeit in einer hierarchischen Form, indem das Übergeordnete immer wieder in etwas Untergeordnetes extrahiert werden kann und damit ein Analogon zu unendlichen Baumstrukturen liefert.

5.1.5 Einbettung in einen applikativen Kontext

Property Oriented Expansion wurde ursprünglich in einem imperativen Kontext der Datenflussanalyse vorgestellt und angewendet. Hierbei wurde das Verfahren zunächst im Rahmen der Elimination partieller Redundanzen und partiell toter Zuweisungen eingesetzt. Durch geschickte Wahl der Transformer und einer sich an den Expansionsprozess anschließenden Nachbearbeitung wurden hierdurch erstmalig *alle* derartigen Redundanzen eliminiert (siehe hierzu [Ste96]).

Hierbei wurde das vorgestellte Verfahren seinem Wesen nach imperativ eingeführt und auch so behandelt. Um in dieser Arbeit an geeigneter Stelle jedoch weitere Aussagen treffen zu können und eine für diese Art von Aussagen geeignete Grundlage der Formalisierung zu schaffen, soll das Verfahren zunächst in eine applikative – und damit deklarative – Sicht überführt werden, die es uns erlaubt, von zeitlichen und zustandsbasierten Aussagen Abstand zu nehmen.

Bei diesem Ansatz sollen zwei Arten der Applikation unterschieden werden. Diese Kategorisierung beruht zunächst auf der einfachen Beobachtung, dass die der Expansion zugrundeliegenden Graphen ihrer Natur nach sowohl als Tupelmenge, deren Tupel jeweils auf der Grundmenge der Knoten des Graphen aufbauen, wie auch als Funktion aufgefasst werden können. In diesen verschiedenen Sichten, werden die Knoten in der tupelorientierten Anschauung als Menge, in der auf Funktionen beruhenden Sicht als Definitionsbereich aufgefasst. Die Kanten dagegen werden in der erstgenannten Sicht entsprechend als Tupel, bestehend aus zwei Vertretern der Knotenmenge, aufgefasst. Die Orientierung der Tupel impliziert bei dieser Darstellung zugleich die Orientierung der Kante. Die Funktionssicht dagegen beschreibt Kanten als Applikation der Funktion auf Argumente des Definitionsbereiches. Die Orientierung der Kanten ist in dieser Sicht durch die Orientierung des Applikationsprozesses von Funktionen (der im Allgemeinen nicht bijektiv ist) gegeben. Hierbei ist die Signatur einer solchen Funktion $f : N \rightarrow 2^N$ – es werden also Knoten des Graphen als Argumente, und Werte der Funktionsapplikation als Knotenmengen identifiziert.

Das folgende Beispiel soll diese beiden Vorstellungen konkretisieren. Das Beispiel beschreibt einen Graphen, der lediglich einen Knoten mit einer reflexiven Kante besitzt. So ergeben sich in den beiden jeweils vorgestellten Sichten (tupel- und applikationsorientiert), die folgenden Interpretationen für diesen Graphen.

tupelorientiert Die Menge der Knoten besteht lediglich aus dem einzelnen Knoten selbst. Die Kanten dagegen, die als Tupelmenge auf dieser Knotenmenge aufgefasst werden sollen, bestehen aus einem einzigen Tupel, das in beiden Komponenten diesen einzigen Knoten enthält.

$$N = \{v\}$$
$$E = \{(v, v)\}$$

applikationsorientiert In der funktional orientierten Interpretation ist die zugrunde liegende Funktion nur in einem Wert definiert – nämlich gerade in dem Knoten, der in der tupelorientierten Sicht in der Grundmenge angenommen wurde. Die Kanten dieses Graphen dagegen entsprechen nun

gerade einer Anwendung dieser Funktion auf diesen einen Wert, in dem sie definiert ist und es ergibt sich insgesamt die folgende Funktion G.

$$G(x) = \begin{cases} \{v\} & \text{falls } x = v \\ \bot & \text{sonst} \end{cases}$$

Leicht lässt sich sehen, dass dem Wesen nach kein Unterschied zwischen den beiden Darstellungen besteht, sind Funktionen in ihrer deklarativen Sicht doch lediglich kompakte Beschreibungen von Tupelmengen, die Symbole aus dem Definitionsbereich auf Symbole des Wertebereiches abbilden. Wie diese Abbildung genau zu geschehen hat, ist für den mathematischen Funktionsbegriff (im Gegensatz zum Prozedurbegriff der Informatik) jedoch unerheblich. Insbesondere ermöglicht erst diese Trennung des Funktionenbegriffs von der Realisierung der Funktion die Existenz von nicht berechenbaren Funktionen. Andererseits beschreiben diese beiden vorgestellten Darstellungen für Graphen gerade diejenigen Unterschiede, die aus den impliziten Annahmen über das Konzept des Graphen herrühren. So werden Graphen meist zwar als Struktur aufgefasst, indem diese im Sinne von Knoten und Kanten als atomare struktur gebende Einheiten etabliert und auch gedacht, aber als imperative und prozesshafte Struktur im Sinne eines Kontrollflussgraphen interpretiert werden. In diesem Sinne ist Model Checking also gerade die Schnittstelle zwischen funktional orientierten Eigenschaftsbeschreibungen auf imperativ ausgerichteten Modellierungen. Weitere Erkenntnisse lassen sich im Sinne eines „Paradigmenwechsels“[4] hin zu einer Umkehrung dieser Situation in Richtung imperativ beschriebener Eigenschaften auf deklarativen Modellvorstellungen gewinnen. Dies entspräche gerade einer Umkehrung der Modellrelation, die auf den ersten Seiten dieser Arbeit das Model Checking Problem gerade als Frage danach, ob eine Modellrelation $M \models \phi$ erfüllt ist, hin zu der Frage, ob die Modellrelation $\phi \models M$ erfüllbar ist, wobei nun jedoch ϕ die Aufgabe der Modellstruktur und M die Aufgabe der Eigenschaft übernimmt.

Nach dieser Listung verschiedener Anwendungen des POE Verfahrens und einer Einbettung des selbigen in einen applikativen Kontext, sowie einem kleinen Ausblick auf die möglichen Konsequenzen, die aus Sichtwechseländerungen resultieren, welche auf der Verquickung von imperativen und deklarativen Gefügen beruhen, soll das nächste Kapitel schließlich ein weiteres beispielhaftes Anwendungsgebiet von POE etablieren. Hierbei wird POE benutzt, um Spielgraphen abzurollen und um aus diesem Abrollprozess Informationen über das dem Spielgraphen zugrunde liegende Model Checking Problem zu erhalten. Wenngleich das Kapitel eine gesonderte Stellung in dieser Arbeit einzunehmen scheint, ist es dennoch „nur“ eine weitere Anwendung des POE Verfahrens auf eine Graph orientierte Modellstruktur.

[4]Dieser überstrapazierte Begriff soll hier lediglich auf eine starke, wenn auch leichtgewichtige Änderung der Sicht, denn auf eine gar revolutionäre Umkehrung des Bestehenden hindeuten.

5.2 Pläne

Wie die Beispiele des letzten Abschnittes zeigten, ermöglicht POE, einen Graphen nicht nur im Sinne einer Traversierung zu durchlaufen, sondern selbst auf diesen Traversierungsprozess (der im Folgenden, ob des Umstandes des nicht notwendigerweise vollständigen Durchlaufens des Graphen, als *Exploration* bezeichnet wird) einwirken zu können. Somit lassen sich Informationen während der Exploration sowohl akkumulieren, wie auch verändern. Bricht eine klassische Breiten- oder Tiefensuche an einer Stelle ab, wo ein Knoten bereits besucht und entsprechend markiert worden ist, so besitzt ein POE basierter Algorithmus jederzeit die Möglichkeit, abhängig vom bisherigen Verlauf über die weitere Exploration des Graphen zu entscheiden. Somit lässt sich eine klassische Breiten- oder Tiefensuche gerade als Spezialfall von POE auffassen, die den Explorations- und Expansionsprozess abbricht, sobald ein Knoten erneut besucht wird. Ist das wiederholte Besuchen eines Knotens im Falle einer klassischen Traversierung sinnvoll und ein notwendiges Terminierungskriterium für den Algorithmus, so verwehrt gerade dieser Umstand jedoch im Allgemeinen einen Einblick in die sich aus einem differenzierteren Terminierungskriterium ergebenden und auf Kreisen aufbauenden Explorationsstrukturen. Für die klassische Traversierung spielt es keine weitere Rolle, welche Art von Kreis durch das erneute Besuchen geschlossen wird. Durch die Ausgestaltungsmöglichkeit eines differenzierteren Terminierungskriteriums ist es jedoch möglich, in vielfältiger Art und Weise die Graphstruktur berücksichtigend den Explorationsprozess adaptiv anzupassen.

Insbesondere ist es POE basierten Explorationsalgorithmen möglich, ein vollständiges „Abrollen“ des Graphen zu erzielen, indem alle Möglichkeiten, den Graphen zu explorieren – unter Umständen durch mehrmaliges Besuchen eines Knotens – explizit in einen kreisfreien (möglicherweise nicht-endlichen) Graphen überführt werden. Von besonderem Interesse für diese Arbeit sind gerade alle Graphexplorationen (im Folgenden X_G genannt), die aus einer bestimmten Knotenmenge heraus starten. Speziell bei Graphen mit Kreisen ist jedoch zu sehen, dass dieses Explorationsverhalten zu einer unendlichen Menge von Möglichkeiten für diesen Explorationsprozess führt, lässt man das mehrmalige Besuchen eines Knotens zu. Daher soll es im Folgenden darum gehen, sinnvolle Teilmengen dieser Gesamtstruktur X_G auszumachen.

5.2.1 Eine Taxonomie für Pläne

Der Untersuchung und Formalisierung der soeben kurz eingeführten Menge X_G widmet sich der folgende Abschnitt. Um diese unendliche Menge X_G aller Graphexplorationen beherrschen zu können, ist es an dieser Stelle ratsam, sich über die Struktur dieser Menge Gedanken zu machen und eine Taxonomie herauszuarbeiten, die die Menge in jeweils endliche und damit handhabbare Teilstrukturen zerlegt. Hierzu werden diese Explorationen zunächst in solche mit Längenbeschränkungen und solche mit Beschränkungen hinsichtlich struktureller Merkmale eingeteilt. Diese strukturellen Merkmale sind gerade die Explorationen, die Knoten nur einmalig besuchen – und als *kreisfrei* bezeichnet werden – und

solche, die Knoten mehrmals besuchen. Schließlich werden diejenigen Explorationen, die Kreise enthalten, noch genauer unterteilt, indem jene Kreise hervorgehoben werden, die Knoten besuchen, die man ohne diesen Kreis ausgelassen hätte – die im Explorationsprozess also neue Knoten besuchen.

Eine Formalisierung des bereits oben Genannten beschreibt die nun folgende Bestimmung von Teilmengen von X_G, die nicht notwendigerweise disjunkt sein müssen.

X_G^k Alle Explorationen, die höchstens k Knoten besuchen.

$X_G^{\mathbf{1}}$ Genau eine beliebige Exploration.

$X_G^{\sim}$ – X_G **ohne Zyklen** Alle Explorationen, die keine Zyklen enthalten und somit – werden alle erreichbaren Knoten besucht – genau einer klassischen Traversierung mittels einer Breiten- oder Tiefensuche entsprechen.

X_G° – X_G **mit Zyklen** Alle Explorationen die Zyklen enthalten. Diese Menge erlaubt eine weitere Zerlegung in zwei weitere Teilmengen.

- $X_G^{\odot}$ – X_G **mit Mehrwertzyklen** Explorationen mit Zyklen, die einen *Mehrwert* enthalten. Hierunter sind genau alle Zyklen zu verstehen, die Knoten enthalten, die ohne diesen Zyklus nicht in der Exploration vorkommen würden. Durch das Explorieren eines Zyklus werden in diesem Fall also Knoten erreicht, die zuvor noch nicht betrachtet wurden, die für den Explorationsprozess demnach *neu* sind und den Mehrwert dieser Exploration beschreiben. Dieser Menge wird im Verlaufe der Arbeit eine größere Aufmerksamkeit gewidmet.
- $X_G^{\oslash}$ – X_G **mit Zyklen ohne Mehrwert** Dies sind gerade diejenigen Explorationen, die Zyklen enthalten, die ihrerseits Knoten enthalten, die in der Explorationen ebenfalls enthalten wären, auch wenn der Zyklus nicht enthalten wäre. Durch den Zyklus werden nur Knoten etabliert, die der Exploration auch ohne diesen Zyklus früher oder später bekannt werden.

Eine formale Definition dieser Mengen zeigt Abbildung 5.11. Eine Exploration wird in diesem Zusammenhang als eine Liste von besuchten Knoten aufgefasst. Insgesamt ergibt sich bzgl. Mengeninklusion das in Abbildung 5.12 dargestellte Bild.

Ein kleines Beispiel soll die oben erwähnte Zerlegung der Menge X_G mit Leben füllen. Hierzu sei der in Abbildung 5.13 gezeigte Graph gegeben. Der Explorationsprozess möge jeweils im Knoten a starten und über die Knoten b und c – möglicherweise in wiederholter Wiederkehr – bis zur Senke d fortschreiten, wo er schließlich endet. Wie sehen also die verschiedenen möglichen Graphexplorationen aus, die durch die obigen Definitionen etabliert wurden?

X_G Die Menge aller möglichen Explorationen dieses Graphen lässt sich in einem regulären Ausdruck wiedergeben, der die zu besuchenden Knoten beschreibt. Nach dem Start in Knoten a folgt unter Umständen ein Besuch des

$$
\begin{aligned}
&X_G = \{(x_0 \dots x_n) | x_i \in V_G, \forall 0 \leq i < n : (x_i, x_{i+1}) \in E_G\} \\
&X_G^{\mathbf{1}} = \{x | x \in X_G\} \text{ mit } |X_G^{\mathbf{1}}| = 1 \\
&X_G^{k} = \{(x_0 \dots x_n) \in X_G | n < k\} \\
&X_G^{\sim} = \{(x_0 \dots x_n) \in X_G | \forall i, j, i \neq j : x_i \neq x_j\} \\
&X_G^{\circ} = \{(x_0 \dots x_n) \in X_G | \exists i, j, i \neq j : x_i = x_j\} \\
&X_G^{\odot} = \{(x_0 \dots x_n) \in X_G^{\circ} | \forall i, j, i \neq j, x_i = x_j : \exists i < k < j : x_k \notin \{x_0 \dots x_i, x_j \dots x_n\}\} \\
&X_G^{\oslash} = X_G^{\circ} \setminus X_G^{\odot}
\end{aligned}
$$

Abbildung 5.11: Eine Taxonomie von Graphexplorationen

$$
\begin{array}{ccccccc}
 & & X_G^{1} & & & & \\
 & & |\cap & & & & \\
X_G^{k} & \subseteq & X_G & \supseteq & X_G^{\circ} & \supseteq & X_G^{\oslash} \\
 & & |\cup & & |\cup & & \\
 & & X_G^{\sim} & & X_G^{\odot} & &
\end{array}
$$

Abbildung 5.12: Eine Übersicht über die Mengeninklusionsbeziehungen verschiedener Graphtraversierungsmöglichkeiten.

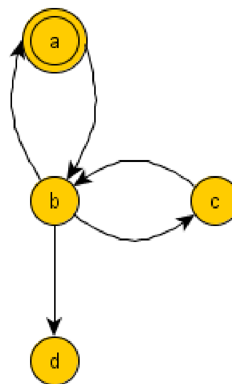


Abbildung 5.13: Beispiel für Explorationsmöglichkeiten auf einem Graphen.

Knotens b. Daran anschließend ist eine beliebige Folge von Besuchen von a oder c möglich, um wieder in den Knoten b zu gelangen. Schließlich bleibt noch die Möglichkeit, in den Knoten d einzumünden, um den Explorationsprozess zu beenden. Die Menge aller solchen *vollständigen* (also in d endenden) Explorationen lässt sich durch den regulären Ausdruck $ab(cb|ab)^*d$ beschreiben. Die Menge X_G lässt sich nun aus allen endlichen Prefixen der durch diesen regulären Ausdruck erzeugbaren Wörter herleiten. Eine nicht ganz formale (aber noch knappe) Beschreibung würde also lauten $X_G = 2^{ab(cb|ab)^*d}$. Bereits dieses kleine Beispiel zeigt eine Vielzahl von möglichen Explorationen des Graphen auf.

$X_G^{\mathbf{1}}$ Eine (nicht eindeutige) einelementige Menge, die eine beliebige Exploration des Graphen beschreibt, wäre etwa $X_G^{\mathbf{1}} = \{abcb\}$.

X_G^3 Die Menge aller Explorationen, die höchstens drei Knoten besuchen, lautet für das gegebene Beispiel $X_G^3 = \{a, ab, aba, abc, abd\}$.

$X_G^{\sim}$ Alle Explorationen, die keine Kreise enthalten, lassen sich in diesem Fall wie folgt beschreiben $X_G^{\sim} = \{a, ab, abc, abd\}$.

X_G° Werden Kreise zugelassen, insbesondere auch beliebige Kreise, so überschreitet die Zahl der möglichen Graphexplorationen spontan die Grenze der Endlichkeit. Die möglichen Graphexplorationen lassen sich auch in diesem Fall wiederum als regulärer Ausdruck $X_G^{\circ} = ab(ab|cb)((ab)^*|(cb)^*)^*(a|c|d)^?$ darstellen. Eine Beschreibung des Ausdrucks könnte wie folgt aussehen: Nach dem Start aus Knoten a muss mindestens ein Kreis über den Knoten b oder c geschlossen werden. Möglicherweise wird daran anschließend ein Knoten a, c oder d besucht.

$X_G^{\odot}$ Die Menge aller Explorationen, die ausschließlich Kreise mit Mehrwert enthalten; die also nur Kreise enthalten, deren Entfernung die Menge der besuchten Knoten verkleinern würde, lässt sich durch den Ausdruck

$$X_G^{\odot} = \{abcbd, aba, abcba\}$$

beschreiben und enthält drei Elemente. Im ersten Fall wird durch das zweite Aufsuchen von b ein Kreis geschlossen, der jedoch einen Knoten c enthält, der außerhalb dieses Kreises nicht vorkommt. Genauso verhält es sich im zweiten Fall, wobei hier der Knoten a einen Kreis schließt, der einen nicht außerhalb dieses Kreises vorhandenen Knoten b besitzt und deshalb das Kriterium eines *Mehrwertzykluses* rechtfertigt. Im letzten Fall werden zwei Kreise geschlossen (einmal vom Knoten a und einmal vom Knoten b), die jedoch in beiden Fällen wiederum Knoten aufsuchen, die außerhalb dieser Zyklen nicht besucht werden (eben den Knoten c).

Diese Menge ist endlich, sofern es der zu expandierende Graph ist. Dies ist leicht einzusehen, fordert diese Menge doch gerade in den Kreisen immer wieder das

$$
\begin{aligned}
f_{X_G^{\mathbf{1}}}(n)(A) &= \begin{cases} A+1 & \text{falls rand()} > 0.5 \\ \bot & \text{sonst} \end{cases} \\
f_{X_G^{k}}(n)(A) &= \begin{cases} A+1 & \text{falls } A \leq k \\ \bot & \text{sonst} \end{cases} \\
f_{X_G^{\sim}}(n)(A) &= \begin{cases} A \cup \{n\} & \text{falls } n \notin A \\ \bot & \text{sonst} \end{cases} \\
f_{X_G^{\odot} \cup X_G^{\sim}}(n)(A) &= \begin{cases} \{n\} \cup \{\hat{x} | x \in A\} & \text{falls } n \notin A \text{ und } \hat{n} \notin A \\ \{n\} \cup A \backslash \{\hat{n}\} & \text{falls } \hat{n} \in A \\ \bot & \text{sonst } (n \in A) \end{cases}
\end{aligned}
$$

rand() ermittelt hierbei eine Zufallszahl zwischen 0 und 1.

Abbildung 5.14: Property Transformer für die Plantaxonomie.

Besuchen neuer, noch bisher unbesuchter Knoten. Besitzt die zu expandierende Graphstruktur jedoch nur endlich viele Knoten, so können auch nur endlich viele neue Knoten besucht werden.

$X_G^{\oslash}$ In dieser Menge sind genau alle Explorationen enthalten, die Kreise besitzen, die keinerlei Mehrwert besitzen. Im Fall des konkret gegebenen Beispiels ist diese Menge wiederum durch einen regulären Ausdruck wie folgt beschrieben $X_G^{\oslash} = ab(ab)^+d^?$. Das Aufsuchen von Knoten c ist im Speziellen nicht zulässig, da ansonsten ein Mehrwertzyklus entstehen würde, der in dieser Menge nicht enthalten sein darf.

Gerade das Beispiel der letzten Menge hat jedoch gezeigt, dass die Beschreibung der Teilmengen von X_G nicht ausreicht, um die Menge X_G selbst erschöpfend zu beschreiben, weshalb diese Mengen auch nur als Teilmengen und nicht als Partitionen bezeichnet wurden. Vielmehr ergeben erst vielfältige Kombinationsmöglichkeiten der Elemente der jeweiligen Mengen $X_G^*, * \in \{\mathbf{1}, k, \sim, \odot, \oslash\}$ die Gesamtmenge X_G. So ist für das Beispiel aus Abbildung 5.13 etwa die Exploration *ababcb* weder in der Menge $X_G^{\odot}$ noch in der Menge $X_G^{\oslash}$ enthalten, da diese sowohl den Mehrwertzyklus *bcb*, als auch einen Zyklus ohne Mehrwert *aba* enthält.

Im Folgenden werden einige dieser soeben vorgestellten Teilmengen durch eine geeignete Wahl von Transformern und mittels POE explizit ermittelt. Kandidaten hierfür sind gerade die endlichen Mengen, kann ansonsten doch eine Terminierung des POE basierten Algorithmuses nicht garantiert werden. Die Menge X_G dagegen ist leicht durch den Graphen selbst, zumindest implizit, darstellbar – nämlich gerade durch den Graphen selbst, der alle Möglichkeiten ihn zu explorieren durch seine Knoten- und Kantenmenge beschreibt.

5.2.2 Graphexploration mittels POE

Die im letzten Abschnitt vorgestellten endlichen Teilmengen der potenziell unendlich großen Grundmenge X_G lassen sich, wie bereits in der Einleitung zu diesem Kapitel angedeutet, mittels POE ermitteln. In diesem Abschnitt sollen

zu diesem Zwecke die geeigneten Transformer vorgestellt werden, die diese Aufgabe leisten sollen. Diese sind bereits gesammelt in Abbildung 5.14 dargestellt und sollen im Folgenden noch weiter erläutert werden.

Der Transformer zur Ermittlung einer beliebigen endlichen Graphexploration $f_{X_G^{\mathtt{1}}}$ nutzt eine Zufallszahl, für eine Abbruchbedingung. Solange eine zufälliger Sollwert noch nicht erreicht ist, wird die Transformation fortgesetzt. Bei genauerer Betrachtung ist zunächst nicht ersichtlich, ob dieser Transformer überhaupt terminiert, offenbart er doch die Möglichkeit, dass das Zufallsexperiment immer wieder das „falsche“ Ergebnis liefert und die Transformation damit beliebig fortgesetzt werden kann. Andererseits ist auch zu sehen, dass die Wahrscheinlichkeit für viele solche, die Terminierung gefährdende Zufallsereignisse, mit dem Anstieg ihrer Anzahl immer geringer wird. Ferner ist bei einer nicht erfolgenden Terminierung, also unendlich vielen solchen terminierungsgefährdenden Ereignissen mit einer Null-Wahrscheinlichkeit einer Nicht-Terminierung zu rechnen. Eine formale Ausführung dieser Argumentation findet sich im Anhang im Abschnitt B.2.

$f_{X_G^{\mathtt{1}}}$

Der Transformer zur Ermittlung einer beliebigen Exploration mit einer Obergrenze zu explorierender Knoten $f_{X_G^k}$ benutzt eine Obergrenze k und einen Laufindex über die Transformationsannotation. Solange die Annotation den gegebenen Maximalwert noch nicht erreicht hat, wird sie erhöht. Ist der Maximalwert irgendwann erreicht, so terminiert die Exploration.

$f_{X_G^k}$

Der Transformer zur Bestimmung kreisfreier Explorationen von Knoten $f_{X_G^{\sim}}$ merkt sich in der Transformationsannotation die bereits besuchten Knoten in Form einer Menge. Wird ein Knoten, der in der Menge bereits besuchter Knoten schon vorhanden ist erneut aufgesucht, so terminiert die Exploration in diesem Fall. Hierdurch ist sichergestellt, dass nur jene Knoten aufgesucht werden, die noch nicht in der Menge bereits besuchter Knoten (in der Expansionsannotation) vorhanden sind. Dadurch wird das Schließen eines Kreises verhindert und es werden jederzeit nur Knoten aufgesucht, die bisher noch nicht aufgesucht wurden.

$f_{X_G^{\sim}}$

Die letzte und auch komplexeste Transformation zur Bestimmung endlicher Explorationen wie auch Explorationen mit Mehrwertzyklen $f_{X_G^{\odot} \cup X_G^{\sim}}$ unterscheidet sowohl die Menge der schon besuchten Knoten, wie auch einen Status eines Knotens, der Buch darüber hält, ob sich seit dem letzten Besuch an der Menge der besuchten Knoten etwas geändert hat. Zu diesem Zweck wird, wie bereits auch beim Transformer $f_{X_G^{\sim}}$ eine Menge besuchter Knoten gehalten. Innerhalb dieser Menge kann ein Knoten den Status *gesund* und *krank* erhalten. Gesunde Knoten in der Menge sind mit einem Dach, kranke Knoten ohne eine Dach gekennzeichnet, so dass in der Menge $\{\hat{a}, b\}$ der Knoten a gesund, der Knoten b dagegen krank ist.

$f_{X_G^{\odot} \cup X_G^{\sim}}$

Nun sind während der Exploration drei Fälle denkbar. Ist der aktuelle Knoten weder gesund, noch krank in der Menge der bereits besuchten Knoten enthalten, so wird er als kranker Knoten aufgenommen und alle anderen Knoten der Menge wieder „geheilt“ und entsprechend auf gesund gesetzt. Ansonsten ist der Knoten bereits in der Menge der besuchten Knoten enthalten. Dort kann er entweder als gesunder Knoten oder aber als kranker Knoten vorhanden sein.

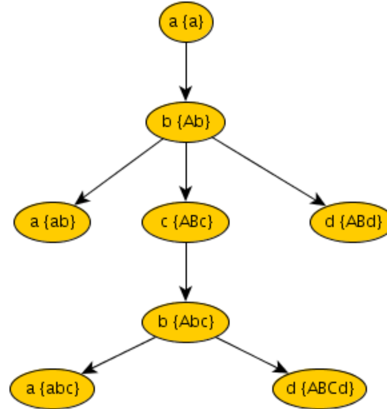


Abbildung 5.15: Beispielexploration mit Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$.

Ein gesunder Knoten wird auf krank gesetzt, ohne die anderen Knoten dadurch zu beeinflussen. Wird dagegen ein bereits besuchter Knoten in einem kranken Zustand erneut aufgefunden, so terminiert die Exploration an dieser Stelle.

Der Gesundheitszustand eines Knotens lässt sich in diesem Sinne auch dahingehend interpretieren, ob seit der letzten Exploration dieses Knotens ein neuer, noch nicht zuvor gesehenen Knoten exploriert wurde, oder nicht. Knoten sind gesund, solange sich seit dem letzten Aufsuchen des Knotens neue Knoten in der Explorationsmenge eingefunden haben. Er kränkelt jedoch, sobald ein Knoten erneut aufgesucht wird, ohne dass sich etwas an der Explorationsmenge geändert hat – bis auf den Gesundheitszustand der Knoten in ihr. Schließlich wird die Exploration beendet, wenn ein Knoten erneut aufgesucht wird, ohne dass sich seit dem letzten Aufsuchen, an der Explorationsmenge etwas geändert hat. Die Zyklen werden in dieser Vorstellung also gerade immer dann geschlossen, wenn ein Knoten erneut aufgesucht wird. Der Zyklus ist in diesem Sinne akzeptabel, wenn sich seit dem letzten Aufsuchen des Knotens neue Knoten in der Exploration eingefunden haben. Ansonsten wird die Exploration abgebrochen.

Eine Beispielexploration für diesen Transformer ist in Abbildung 5.15 zu sehen. Der Transformer expandiert den schon aus Abbildung 5.13 von Seite 68 bekannten Graphen mit Hilfe eben dieses Transformers. Gesunde (im Transformer mit einem Dach versehene) Knoten sind in dieser Darstellung mit großen Buchstaben, kranke (im Transformer ohne Dach versehene) Knoten dagegen mit kleinen Buchstaben ausgezeichnet.

5.2.3 Eigenschaften von POE und seinen Transformern

Nachdem diverse Transformer zur Bestimmung von beliebigen Graphexplorationen im letzten Abschnitt vorgestellt wurden, soll es nun darum gehen, eine wichtige Eigenschaft des Transformers $f_{X_G^{\odot} \cup X_G^{\sim}}$ nachzuweisen. Nämlich gerade,

dass die durch diesen expandierten Graphen keine Kreise enthalten und sich demnach für weitere Aufzählungen besonders eignen. Für dieses Resultat sind jedoch noch einige Zwischenschritte notwendig, die ebenfalls vorgestellt werden. Zunächst wird der Begriff der Monotonie auf Graphen übertragen und argumentiert über die Mächtigkeit der Annotationen an den Knoten während der Expansion. Anschließend beschreibt der Abschluss dieser neuen Variante der Monotonie, dass sie sich auf Pfadstrukturen übertragen lässt. Schließlich wird der Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ auf eben diese neuen Begrifflichkeiten angewendet, um die gewünschte Eigenschaft der Kreisfreiheit in den Graphen nachzuweisen.

Zunächst soll der Begriff der *strukturellen Monotonie* definiert werden, der beschreibt, dass sich Annotationen (in diesem Falle Mengen) während eines Expansionsprozesses in ihrer Mächtigkeit monoton verhalten. Für Graphen, die keine Mengen als Annotationsinformation tragen, sondern etwa funktionale Strukturen oder etwa Zahlen, findet diese Definition keine Anwendung. Dies wird in der folgenden Definition formalisiert.

Definition 4 (strukturelle Monotonie) *Einen expandierten Graphen bezeichnet man als* strukturell monoton *(oder kurz* monoton*), wenn für jeweils zwei beliebige Knoten* (n, A) *und* (n', A') *gilt*

$$(n, A) \to (n', A') \Rightarrow |A| \leq |A'|$$

Dieser Begriff der Monotonie lässt sich, definiert in einer lokalen Sicht, auch auf Pfade auf dieser Graphstruktur generalisieren. Dies zeigt das folgende einfache Lemma.

Lemma 1 (Transitiver Abschluss der strukturellen Monotonie) *Sei ein Pfad in einem* monotonen *Graphen G gegeben, der OBdA die folgende Gestalt besitzt.*

$$\ldots \to (n_i, A_i) \to \ldots \to (n_{i+j}, A_{i+j}) \to \ldots$$

Dann gilt

$$|A_i| \leq |A_{i+j}|$$

Beweis *Der Beweis wird mittels Induktion über j geführt. Für den Induktionsanfang sei also* $j = 0$*. Dann folgt jedoch unmittelbar* $|A_{i+0}| \leq |A_{i+0}| \Leftrightarrow |A_i| \leq |A_{i+0}|$*. Sei die Eigenschaft nun also gezeigt für* $j-1$ *und es gelte* $|A_i| \leq |A_{i+j-1}|$*. Wähle nun die Kante* $(n_{i+j-1}, A_{i+j-1}) \to (n_{i+j}, A_{i+j})$*. Da G monoton ist, folgt* $|A_{i+j-1}| \leq |A_{i+j}|$*. Zusammen mit der Induktionsvoraussetzung ergibt sich somit insgesamt* $|A_i| \leq |A_{i+j-1}| \leq |A_{i+j}| \Rightarrow |A_i| \leq |A_{i+j}| \square$

Ist ein Graph monoton und besitzt er einen Kreis, so gilt für diesen Kreis, dass alle Annotationen auf diesem Kreise dieselbe Mächtigkeit besitzen. Dies ist leicht

einzusehen, darf doch die „erste“ solche Annotation in einem Kreis nicht weniger mächtig sein als die „letzte“. Andererseits darf auch die „letzte“ Annotation auf einem solchen Kreis nicht weniger mächtiger sein als die „erste“. Folglich müssen beide Annotationen gleich mächtig sein. Da die Wahl des „ersten“ und „letzten“ Knotens jedoch beliebig war, lässt sich eine solche Argumentation für alle Knoten generalisieren und das Argument folgt für alle Knoten auf dem Kreis. Eine formale Variante dieses Gedankenganges soll nun folgen.

Satz 3 (Mengenkonstanz in Kreisen) *Sei ein Kreis in einem* monotonen *Graphen G gegeben, der oBdA die folgende Gestalt besitzt.*

$$(n_0, A_0) \to (n_1, A_1) \to \ldots \to (n_k, A_k) \to (n_0, A_0)$$

Dann gilt für je zwei beliebige Mengen A_i und A_j

$$|A_i| = |A_j|$$

Beweis durch Widerspruch *Angenommen es gäbe einen Index i mit $(n_i, A_i) \to (n_{i+1}, A_{i+1})$ und $|A_i| \neq |A_{i+1}|$. Da G monoton ist, folgt sowohl $|A_i| < |A_{i+1}|$ als auch $|A_k| \leq |A_0|$. Mittels Lemma 1 folgt jedoch sowohl $|A_0| \leq |A_i|$ als auch $|A_{i+1}| \leq |A_k|$. Insgesamt ergibt sich dann jedoch $|A_k| \leq |A_0| \leq |A_i| < |A_{i+1}| \leq |A_k| \Rightarrow |A_k| < |A_k|$. Ein Widerspruch. Also kann es einen solchen Index i nicht geben* □

Folgend soll nun gezeigt werden, dass ein durch Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ expandierter Graph keine Kreise enthält und sich somit im Sinne eines aufzählenden Durchlaufes eignet. Anschaulich wird dies einfach durch einen Widerspruch, der darauf beruht, dass die annotierten Mengen in einem Kreis nicht kleiner werden. Dann folgt jedoch, dass diese Mengen immer gleich mächtig sein müssen. Wenn die Mengen jedoch immer gleich mächtig bleiben (innerhalb eines solchen *angenommenen* Kreises), muss immer wieder ein Knoten $\hat{n}$ aus der annotierten Menge entfernt werden. Irgendwann wären also alle Knoten $\hat{n}$ aus den annotierten Mengen entfernt. Dann kann jedoch nicht mehr die Bedingung für den Transformer erfüllt sein, der die Mengenmächtigkeit konstant hält. Insbesondere kann es solche Kreise dann also gar nicht geben.

Der eigentliche Beweis gestaltet sich jedoch komplizierter und stellenweise technischer und soll nun vorgestellt werden. Dafür wird zunächst nachgewiesen, dass ein mit dem Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ expandierter Graph monoton ist.

Lemma 2 *Ein durch den Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ expandierter Graph G ist monoton im Sinne von Definition 4.*

Beweis *Wähle zwei beliebige Knoten (n, A) und (n', A'). Gibt es keine Kanten zwischen den beiden Knoten, so ist nichts weiter zu zeigen. Sei also eine Kanten zwischen den beiden Knoten gegeben, die oBdA die Gestalt habe $(n, A) \to (n', A')$. Da der Graph durch Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ expandiert*

wurde, hat die Kante ferner die Gestalt $(n, A) \rightarrow (n', f_{X_G^{\odot} \cup X_G^{\sim}}(n)(A))$. *Entsprechend der Fallunterscheidung des Transformers hat die Kante die Gestalt* $(n, A) \rightarrow (n', \{n\} \cup \{\hat{x} | x \in A\})$ *oder* $(n, A) \rightarrow (n', \{n\} \cup A \backslash \{\hat{n}\})$. *In beiden Fällen wird die Menge jedoch nicht verkleinert und es gilt insgesamt* $|A| \leq |A'|$ □

Schließlich verbleibt es dem noch zu zeigenden und nunmehr folgenden Satz, dass wesentliche Ergebnis dieses Abschnittes zu zeigen. Nämlich, dass die Expansion mit dem Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ in einen kreisfreien Graphen resultiert.

Satz 4 (Kreisfreiheit) *Ein durch den Transformer* $f_{X_G^{\odot} \cup X_G^{\sim}}$ *expandierter Graph* G *ist ein DAG (directed acyclic graph).*

Beweis durch Widerspruch *Angenommen* G *wäre kein DAG. Dann enthält* G *(mindestens) einen Kreis. OBdA habe dieser die Gestalt* $(n_0, A_0) \rightarrow (n_1, A_1) \rightarrow \ldots \rightarrow (n_k, A_k) \rightarrow (n_0, A_0)$. *Aus Lemma 2 und dem Satz über Mengenkonstanz in Kreisen (Satz 3) folgt dann* $|A_0| = |A_1| = \ldots = |A_k|$. *Die Mengen bleiben im Expansionsprozess durch Transformer* $f_{X_G^{\odot} \cup X_G^{\sim}}$ *jedoch nur konstant in ihrer Mächtigkeit, wenn bei jeder Anwendung des Transformers gilt* $f_{X_G^{\odot} \cup X_G^{\sim}}(n)(A) = \{n\} \cup A \backslash \{\hat{n}\}$. *Es muss also nur dieser Fall betrachtet werden.*

Mittels Induktion über i *wird nun gezeigt, dass gilt* $\forall 0 \leq i \leq k : \hat{n_0} \notin A_i$. *Der Induktionsanfang: Es gibt eine Kante in* G *der Gestalt* $(n_k, A_k) \rightarrow (n_0, A_0)$. *Insbesondere gilt also* $A_0 = f_{X_G^{\odot} \cup X_G^{\sim}}(n_0)(A_k)$. *Wegen der Mengenkonstanz muss jedoch gelten* $f_{X_G^{\odot} \cup X_G^{\sim}}(n_0)(A_k) = \{n_0\} \cup A_k \backslash \{\hat{n_0}\} = A_0 \Rightarrow \hat{n_0} \notin A_0$. *Der Induktionsschluss: Es gelte* $\hat{n_0} \notin A_{i-1}$. *Betrachte man nun die Kante* $(n_{i-1}, A_{i-1}) \rightarrow (n_i, A_i)$, *so folgt wiederum aus der Mengenkonstanz im Kreis* $A_i = \{n_i\} \cup A_{i-1} \backslash \{\hat{n}_{i-1}\}$. *Zusammen mit der Induktionsvoraussetzung folgt damit jedoch insgesamt* $\hat{n}_0 \notin A_i$.

$\hat{n}_0$ *ist also in keiner der Mengen* A_i *im Kreis enthalten. Andererseits existiert jedoch eine Kante* $(n_0, A_0) \rightarrow (n_1, A_1)$. *Für* A_1 *gilt jedoch* $A_1 = \{n_0\} \cup A_0 \backslash \{\hat{n}_0\}$. *Dies kann jedoch nur gelten, wenn der Transformer mit der Eigenschaft* $\hat{n}_0 \in A_0$ *expandiert wurde. Ein Widerspruch. Also muss* G *ein DAG sein* □

5.3 Beispielanwendung – Model Checking auf $\exists\mu$

Eine Möglichkeit, die Informationen, die während einer Graphexpansion entstehen, nutzbar zu machen, ist die Expansion eines eine Eigenschaft aus $\exists\mu$ respektierenden Spielgraphen. Hierbei wird die in den vorherigen Abschnitten vorgestellte Taxonomie dazu genutzt, verschiedene Explorationen eines Modelles auf die Expansionen eines Spielgraphen abzubilden. Somit ergeben sich aus der Expansion des Spielgraphen Rückschlüsse auf die Exploration des Modelles. Folglich lassen sich etwa die durch den Transformer $f_{X_G^{\sim}}$ bei der Expansion eines Spielgraphen gesammelten Information nutzen, um Rückschlüsse auf Explorationen des dem Spielgraphen zugrunde liegenden Modelles zu ziehen, die

eine dem Spielgraphen ebenfalls zugrunde liegenden Eigenschaft erfüllen. Die Expansion eines Spielgraphen ist demnach also durch genau *zwei* Eigenschaften charakterisiert. Zum einen wird durch eine Liveness-Eigenschaft aus $\exists\mu$ eine implizite Beschreibung der Lösungsmenge in Form eines Spielgraphen erzeugt. Zum anderen wird durch POE diese Lösungsmenge – die möglicherweise unendlich ist – klassifiziert und auf eine endliche Teilmenge reduziert, die aufgezählt werden kann. Man kann sich diesen Prozess also als Kombination zweier hintereinander geschalteter Filter vorstellen, von denen der erste gerade alle gültigen Explorationen des Modelles liefert und der zweite diese Menge einschränkt auf jene Explorationen, die eine durch den Property Transformer beschriebenen Eigenschaft besitzen. Die zweite Filterung durch POE ist nur notwendig, da die erste Filterung durch die Synthetisierung von Modell und Eigenschaft eine nicht notwendigerweise endliche Lösungsmenge produziert.

Es sind also zwei hintereinander ausgeführte Model Checking Prozesse am Werk, die sowohl auf unterschiedlichen Modellstrukturen, als auch auf unterschiedlichen zu untersuchenden Eigenschaften operieren. Der in diesem Sinne erste Model Checking Prozess synthetisiert die zu untersuchende Eigenschaft und das zu untersuchende Modell in eine gemeinsame Struktur eines Spielgraphen. In dieser Darstellung sind bereits alle Informationen über die Lösung des Problems implizit kodiert. Darauf aufbauend expliziert POE diese Lösungsmenge in eine endliche Darstellung mittels eines zweiten Model Checking Prozesses, der das Modell des Spielgraphen wiederum in eine implizite Lösung eines expandierten Spielgraphen synthetisiert, die nun jedoch durch einfache Graphtraversierungen (in diesem Sinne einem *Ablesen* der Lösungen vergleichbar) in Lösungen für das ursprüngliche Problem übertragen werden können. Die Spielgraphexpansion hat in diesem Fall also im Wesentlichen die gleiche Aufgabe, wie die Spielgrapherzeugung. Wird bei der Spielgrapherzeugung eine (temporale) Eigenschaft in eine graphartige Modellstruktur vermengt, so wird bei der Expansion eine (durch den Transformer beschriebene) Eigenschaft mit der Modellstruktur des Spielgraphen vereinigt. Insbesondere sind diese beiden Prozesse auch beiderseits keine Model Checking Prozesse im *klassischen* Sinne, dass sie eine Ja/Nein-Antwort liefern, sondern erzeugen eine implizite Beschreibung der Lösungsmenge in Form einer Graphstruktur, die, im Falle des Spielgraphen, durch spielbasierte Model Checking Algorithmen, oder aber, im Falle des expandierten Spielgraphen, durch klassische Graphtraversierungen, expliziert werden.

Weiterhin bleibt jedoch zu berücksichtigen, wie sich die Expansion eines Spielgraphen auf die Lösungsmenge (und deren Beschreibung) auswirkt. So ist zum Beispiel eine Exploration eines neuen Spielgraphknotens nicht notwendigerweise auch die Exploration eines neuen Modellknotens, der diesem Spielgraphen zugrunde liegt. Jede Exploration entlang der Formelstruktur (also entlang des strukturellen Aspekts des Problems) folgt Spielgraphknoten, die sich lediglich in ihrer Teilformel- nicht jedoch in ihrer Modellknotenkomponente unterscheiden. Gerade solche Folgen von Spielgraphknoten liefern also keinerlei Information bzgl. der Lösung für das Modell – wenngleich jedoch Informationen bzgl. Erfüllung der Teilformeln. Somit ergibt sich die Frage, in welcher Richtung (also in Richtung der Modellstruktur, in Richtung der Formelstruktur oder aber in beiderlei Richtungen) die Interpretation des Ergebnisses zu geschehen hat. Da das klassische Model Checking Ergebnis Aussagen über das Modell tätigt –

wenngleich sich unter Berücksichtung der Formelstruktur auch neuartige Interpretationen ergäben – soll die Projektion auch an dieser Stelle auf die Modellstruktur vollzogen werden.

Hierbei ergeben sich zwei Möglichkeiten der Realisierung: Zum einen kann diese Projektion auf das Model während der Graphexpansion durch den Transformer, zum anderen in der anschließenden Graphtraversierung des expandierten Spielgraphen durchgeführt werden. Beide Ansätze bieten ihre Vor- und Nachteile. Wird die Projektion bereits während der Expansion durchgeführt, so verbleibt die Arbeit (und damit auch die Rechenzeit) für diese Projektion dem Transformer. Andererseits wird die spätere Traversierung des expandierten Spielgraphen in den einzelnen Knoten erleichtert. Genau andersherum verhält es sich, wenn die Projektion erst in der Graphtraversierung durchgeführt wird. Letztendlich muss für jeden der expandierten Spielgraphknoten die Annotationsinformation bzgl. dieser Projektion durchgeführt werden und keine der beiden Varianten hätte diesbgzl. einen Vorteil gegenüber der anderen.

Abbildung 5.16 zeigt beispielhaft eine Übersicht über das bisherige Vorgehen. Hier sind drei Abbildungen zu sehen, die das Model Checking Problem in seiner klassischen Form, wie auch die daraus extrahierten neuartigen Graphstrukturen zeigen. Es ist ein Graph, der lediglich aus einem Knoten mit einer reflexiven Kante besteht, auf eine Eigenschaft (in diesem Fall $\mu X. \diamond X \vee p$, die einen Pfad zu einem Knoten mit der Eigenschaft p fordert) hin zu überprüfen. Daraus wird der zweite Graph, der Spielgraph erzeugt. Aus Gründen der Übersicht sind die Spielgraphknoten von a bis e durchbuchstabiert. So steht der Knoten d etwa für den Spielgraphknoten, der aus der Synthese des einzigen Modellknotens mit der Teilformel $\diamond X$ entstanden ist. Dieser Spielgraph wird nun mit dem Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ in den dritten in der Abbildung gezeigten Graphen expandiert. Die Knoten Bezeichnungen entsprechen den schon aus Abbildung 5.15 von Seite 72 bekannten Konventionen.

Zunächst ist zu erkennen, dass zwei Knoten (nämlich `c.ABc` und `c.ABcDE`) als Blätter im Baum auftauchen. Diese beschreiben die im Knoten c – und damit insbesondere auch im einzigen Knoten des ursprünglichen Model Checking Problems – gültige Proposition p. Verfolgt man nun im expandierten Spielgraphen die Knoten hin zu diesen Blättern, so ergeben sich zwei mögliche Sequenzen `a.a, b.Ab, c.ABc` und `a.a, b.Ab, d.ABd, e.ABDe, a.aBDe, b.abDe, c.ABcDE`. Projiziert man diese nun auf ihre Spielgraphknotenkomponente, so ergeben sich die beiden Sequenzen `a, b, c` und `a, b, d, e, a, b, c`. Die Formel lässt sich also durch diese beiden möglichen Wege im Spielgraphen belegen. Der erste Pfad erfüllt die Formel entlang der propositionalen Eigenschaften der Formel, eben weil die Formel im Modellknoten bereits erfüllt ist. Der zweite Pfad im Spielgraphen argumentiert über einen Pfad im Modell, der entlang der reflexiven Kante genommen werden kann. Betrachten wir die Knoten im Pfad etwas genauer und schauen uns ihre Komponenten an, so ergeben sich die beiden folgenden alternativen Darstellungen

- $\overbrace{\mu X. \diamond X \vee p.p}^{a}, \overbrace{\diamond X \vee p.p}^{b}, \overbrace{p.p}^{c}$

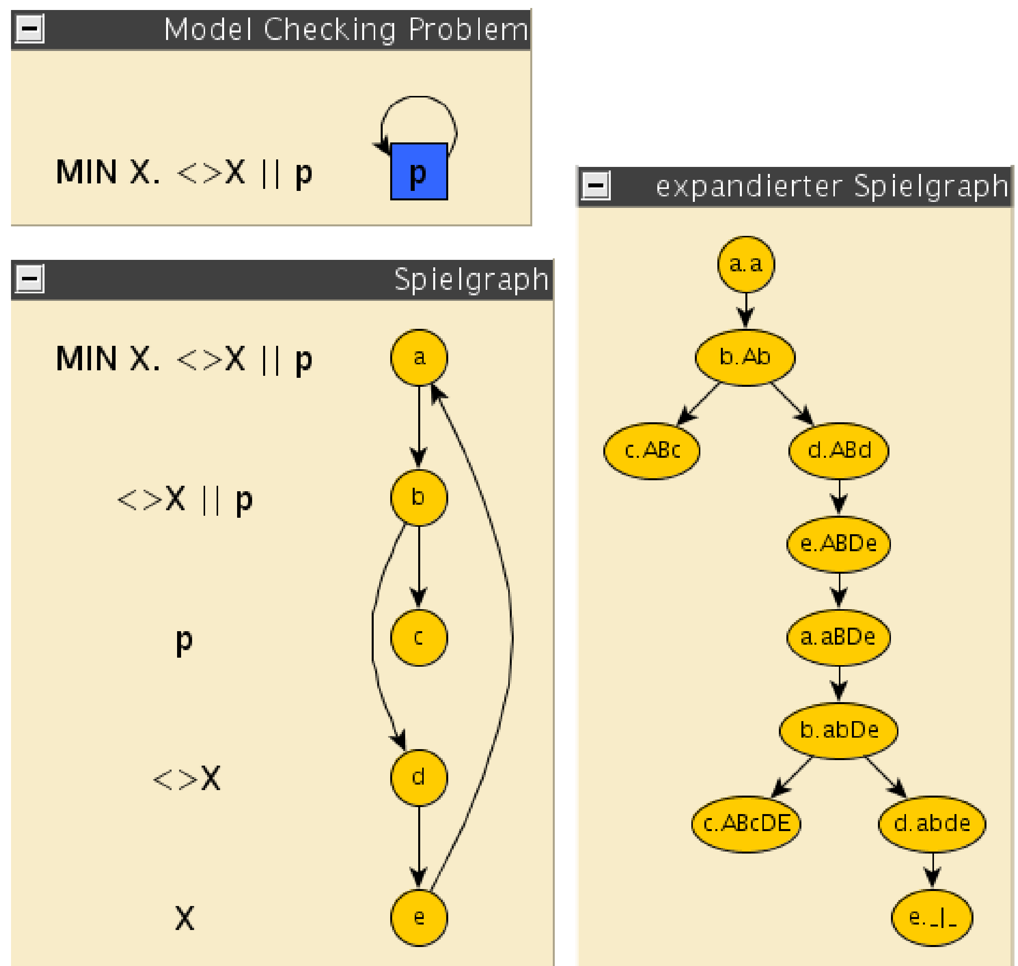


Abbildung 5.16: Beispielexpansion eines Spielgraphen

- $\overbrace{\mu X. \diamond X \vee p.p}^{a}, \overbrace{\diamond X \vee p.p}^{b}, \overbrace{\diamond X.p}^{d}, \overbrace{X.p}^{e}, \overbrace{\mu X. \diamond X \vee p.p}^{a}, \overbrace{\diamond X \vee p.p}^{b}, \overbrace{p.p}^{c}$

für die oben beschriebenen Pfade. Um nun letztendlich an Beleginformationen für das Ursprungsmodell zu gelangen, müssen auch diese Pfade wiederum auf ihre Modellknotenkomponente projiziert werden. Ein naiver Ansatz würde einfach und direkt auf die Modellknoten projizieren. Dabei ergäben sich jedoch nur die wenig vielversprechenden Pfade `p, p, p` und `p, p, p, p, p, p, p`. Diese Projektion lässt unberücksichtigt, ob der Pfad entlang einer propositionalen Komponenten, oder aber entlang einer Modell relevanten Komponente etabliert worden ist. Es ist also nicht zu erkennen, ob das mehrmalige Aufsuchen des Knotens `p` durch ein Explorieren der Teilformeln und deren Erfüllbarkeit, durch ein Explorieren von Nachfolgern, oder aber gar durch eine Kombination aus beidem entstanden ist. Die Projektion auf eine Modellknotenkomponente – und damit insbesondere auch auf eine nicht propositionale Komponente – darf also nur erfolgen, wenn der Beleg auch entlang einer solchen Komponente geführt wird. Wenn also die Exploration entlang eines $\diamond$- oder $\Box$-Knotens (im Falle von $\exists\mu$ also nur entlang eines $\diamond$-Knotens) erfolgt. Diese Projektion liefert schließlich die beiden erwünschten Pfade `p` und `p, p`. Der erstere Pfade ist gültig, weil der Knoten selbst bereits die gewünschte Eigenschaft p, die über einen Pfad zu erreichen ist, selbst besitzt. Der zweite Pfad ist gültig, weil dieser über den Weg über eine Kante einen Pfad zu einem Knoten mit der gewünschten Eigenschaft p erzeugt. Insbesondere ist ein drittes Aufsuchen des Knotens zwar eine ebenso gültige Lösung des ursprünglichen Model Checking Problems, doch keine gültige Lösung im durch POE reduzierten Lösungsraum. Dieser lässt nämlich nur gerade kreisfreie Pfade oder aber Pfade mit Mehrwertzyklen zu. Diese Einschränkung des Lösungsraumes ist durch den Property Transformer bestimmt und kann durch diesen beeinflusst werden, so dass ein Austauschen des Property Transformers zu andersartigen Pfadstrukturen führen kann, die dem Anwendungsfall angemessen sind.

5.4 Beispielanwendung – Toolkombinationen

Eine weitere Beispielanwendung für die Aufzählung von Pfaden beschreibt die Kombination verschiedener Werkzeuge (im Folgenden *Tools* genannt), deren Abhängigkeit in einer Graphstruktur spezifiziert ist (auch Kompatibilitätsgraph genannt). Abbildung 5.17 zeigt beispielhaft die Abhängigkeit vier verschiedener Tools a,b,c und d. Hierbei beschreiben die Kanten eine Reihenfolge, in der diese verwendet werden können. Nach Ausführung von a können b und c in beliebiger Häufigkeit und Reihenfolge ausgeführt werden – eventuell aufbauend auf dem Ergebnis der Ausführung von a. Schließlich muss der Prozess durch ein Tool d beschlossen und damit beendet werden.

Expandiert man diesen Graphen mit dem Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$ ergibt sich das in Abbildung 5.18 gezeigte Bild. Annotationen $\hat{a}$ sind in der Abbildung großbuchstabig als A markiert. Annotationen a sind unverändert als kleinbuchstabiges a zu erkennen. Knoten, die in einen unerwünschten Zustand expandieren – deren Transformationsergebnis also $\perp$ liefert – sind mit _|_ annotiert und die

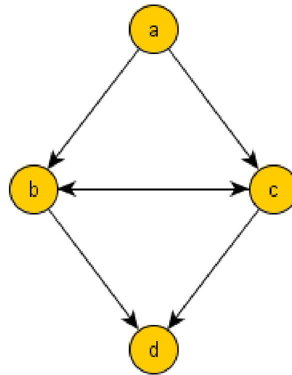


Abbildung 5.17: Spezifikation von Toolabhängigkeiten durch einen Kompatibilitätsgraphen

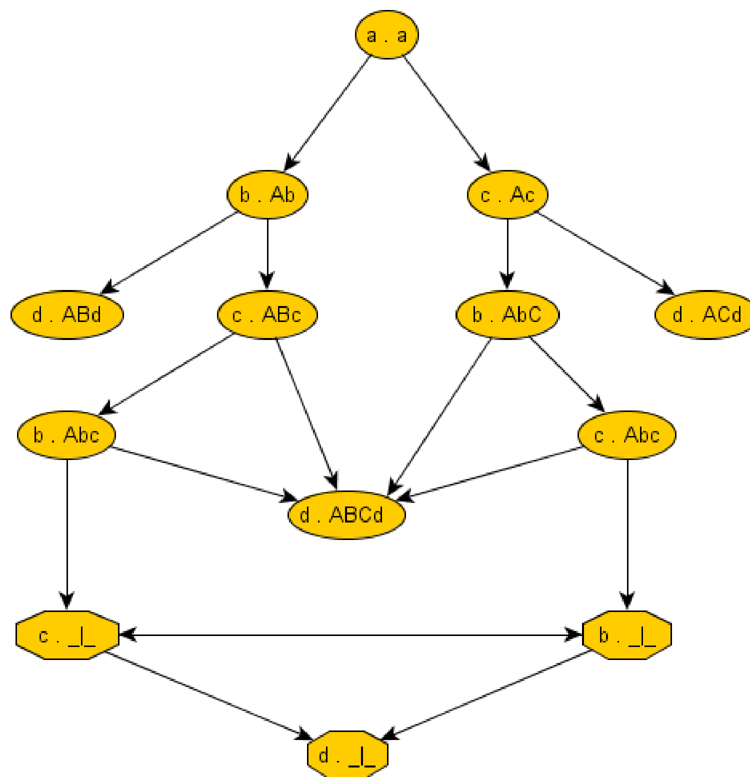


Abbildung 5.18: Expandierter Toolabhängigkeitsgraph.

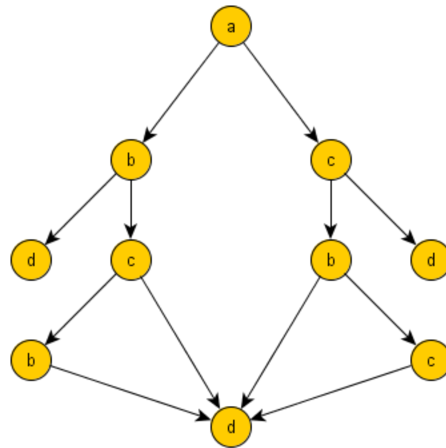


Abbildung 5.19: Expandierter Toolabhängigkeitsgraph nach Elimination redundanter Knoten.

Knoten zur Hervorhebung mit einem Achteck umrahmt. Diese Knoten sind im Expansionsprozess unerwünschte (z.B. zu häufige) Ausführungen eines Tools und können daher entfernt werden.

Entfernt man alle jene Knoten, die eine Annotation $\perp$ besitzen, so ergibt sich das in Abbildung 5.19 gezeigte Bild. Insbesondere wurden in diesem Bild auch die Annotationen der Knoten entfernt, um die Analogie zum Ausgangsgraph aus Abbildung 5.17 zu verdeutlichen.

Doch wie lässt sich dieser neu gewonnene Graph nun deuten? Zunächst lassen sich alle möglichen Wege vom Start der Toolausführung – vom Knoten a – bis zum Ende der Toolausführung in einer Senke – in diesem Fall ausschließlich Knoten d – verfolgen. Damit ergeben sich die folgenden möglichen Sequenzen.

- abd
- $abcd$
- $abcbd$
- $acbcd$
- $acbd$
- acd

Diese Sequenzen beschreiben also alle jene Ausführungen, die unter der Spezifikation aus Abbildung 5.17 gültig sind. In diesem Fall wird ein Tool auch erst ein zweites mal aufgerufen, wenn in der Zwischenzeit bereits ein anderes Tool aktiv war. Dies ist zwar nicht in jedem Fall erwünscht – so kann etwa

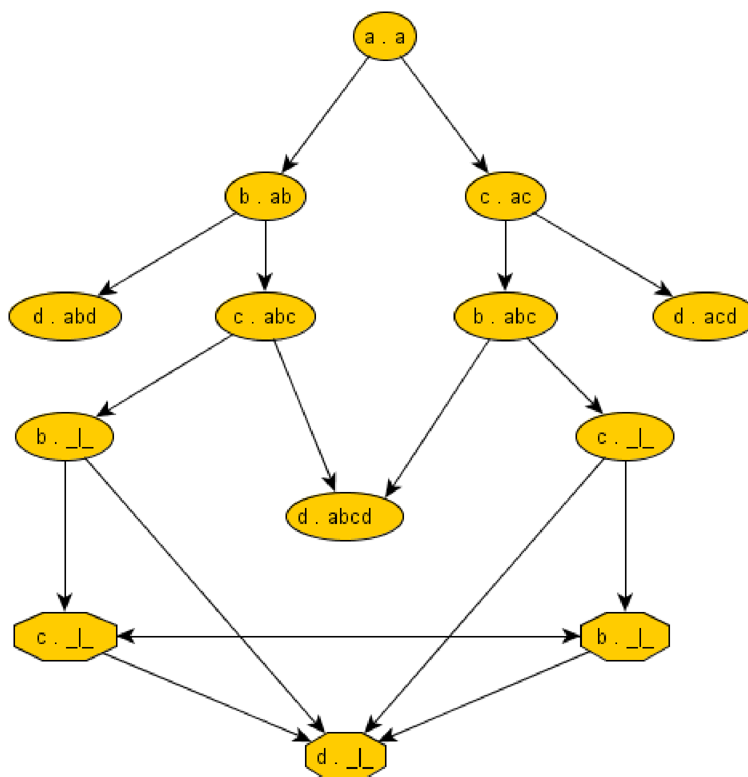


Abbildung 5.20: Expandierter Toolabhängigkeitsgraph für einmalige Ausführung von Tools.

ein Tool Zwischenergebnisse produzieren, die es bei der zweiten Ausführung als zusätzliche Eingaben mit verwertet, wie es etwa bei der Indizierung und Verzeichniserstellung von LaTeXder Fall ist – beschreibt aber doch in den meisten Fällen das Verhalten eines deterministischen ausschließlich von seinen Eingaben abhängenden Tools. Insbesondere wäre die Sequenz *abcbcd* zwar eine der durch den Kompatibilitätsgraphen beschriebenen Spezifikation genügende Sequenz, doch ist nach Abarbeiten der Teilfolge *abcb* ein Ergebnis entstanden, das keine Ausführung von c rechtfertigt, da in dem entstandenen Kreis *cbc* lediglich das Tool b ausgeführt worden ist, dessen Ausführung jedoch bereits erfolgt ist und dem Kreis *cbc* die Legitimation für eine Ausführung entzieht.

Ein weiteres Beispiel für die Ausführungsabhängigkeiten von Tools ist in den Abbildungen 5.20 und 5.21 zu sehen. Diese entsprechen den vorherigen Graphen mit der Ausnahme, dass in diesem Fall eine Toolausführung nicht mehrfach durchgeführt werden, sondern jedes Tool nur einmalig an der Ausführung beteiligt sein darf. Expandiert wurden diese Graphen durch den Transformer $f_{X_{\tilde{G}}}$.

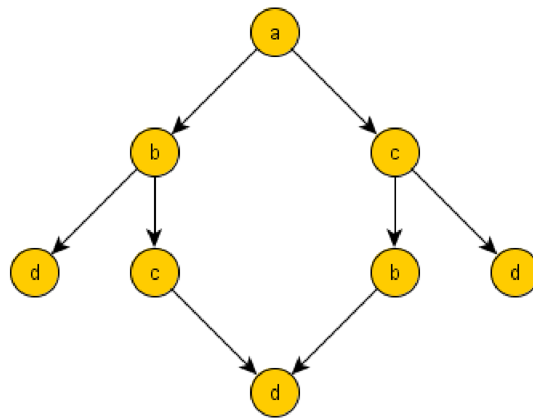


Abbildung 5.21: Expandierter Toolabhängigkeitsgraph nach Elimination redundanter Knoten für einmalige Ausführung von Tools.

Kapitel 6

Implementierung

Nachdem die vorgestellten Konzepte und Lösungen in ihrer Breite und Tiefe untersucht wurden, beschreibt dieses Kapitel Ansätze, diese Verfahren in geeigneter Weise zu implementieren. Hierbei stand zunächst die Frage nach einem geeigneten Paradigma im Vordergrund. In Erinnerung an Kapitel 3 über Sichten und Konzepte im Allgemeinen und Kapitel 2 über Model Checking im Speziellen, beschreibt das Konzept des Model Checking ja gerade eine Vereinheitlichung, oder auch in Bezugsetzung von deklarativen und imperativen Konzepten. Ebenso beschreibt der Übergang von einer klassischen Model Checking Sicht hin zu einer spielbasierten Sicht auf eine ebenso eindrucksvolle Art einen Paradigmenwechsel. Demnach ist die Frage nach der geeigneten Wahl eines Paradigmas also nicht sinnvoll eindeutig zu beantworten und die Wahl fiel auf zwei unabhängige Implementierungen, die den jeweiligen Sichten Rechnung tragen sollen. In den beiden folgenden Abschnitte wird also sowohl ein imperativer wie auch ein deklarativer Ansatz der Implementierung vorgestellt.

Ebenso wird in einem Abschnitt erläutert, wie die Transformer aus den vorherigen Kapiteln benutzt werden können, um aus einer Spielgraphexpansion schließlich Belege für das zugrunde liegende Modell zu erhalten.

Das Kapitel schließt mit einem kurzen Abschnitt über die Integration der Implementierung in ein graphisches Modellierungsframework. Hierdurch ist die Expansion mittels einer graphischen Benutzerführung möglich. Ebenso wurde die Realisierung der Bestimmung der Belege mittels Spielgraphexpansion in einen bestehenden Model Checker integriert und dieser damit erweitert.

6.1 POE

Zentral bei der Bestimmung von möglichen Graphexplorationen war das POE-Verfahren. Dieses wurde benutzt, um auf einer Spielgraphrepräsentation des Model Checking Problems aus den Möglichkeiten des $\diamond$-Spielers, ein Spiel zu gewinnen, Rückschlüsse auf die Möglichkeiten, eine temporale Formel aus $\exists\mu$ zu belegen, zu ziehen. Zu diesem Zwecke wurde der Spielgraph – eben mit Hilfe von

POE – ausgerollt und eine bestimmte Klasse von Graphexplorationen dadurch bestimmt.

Die Implementierung von POE erfolgte in einer imperativen, wie auch in einer funktionalen Sicht – insbesondere war es möglich, die imperative Implementierung aus einer beliebigen anderen POE-Implementierung automatisch zu gewinnen, wie das Beispiel aus Abschnitt 5.1.4 zeigte.

6.1.1 Imperativ

Die imperative Implementierung von POE und den Transformern respektiert die graphartige und damit streng verhaltensorientierte Sicht dieser Lösung, indem diese Implementierung gerade eine graphartige Realisierung vorsieht. Hierbei wird der Quellcode als Kontrollflussgraph modelliert. Sowohl POE lässt sich auf diese Weise als Graph interpretieren – wie dies insbesondere auch die Pseudocodebeschreibung aus Abschnitt 5.1 suggeriert – als auch die für den Transformationsprozess notwendigen Transformer. Somit liefert Abschnitt 5.1.4 bereits eine vollständige imperative Beschreibung des Verfahrens, sobald eine beliebige Implementierung von POE zur Verfügung steht. Abbildung 6.1 zeigt eine imperative Beschreibung, die der Beschreibung in Abschnitt 5.1.4 bis auf einige Intialisierungmechanismen gleicht. Gerade diese Initialisierung findet jedoch implizit in der Initiliasierung des POE-Verfahrens selbst statt und kann demnach bei einer derartigen Vorgehensweise entfallen.

Der Algorithmus selbst wurde als Worklistalgorithmus realisiert, der auf zwei Listen arbeitet. Eine *Todo-Liste* T enthält gerade die Knoten, die im Transformationsprozess noch behandelt werden müssen. Eine *Done-Liste* D enthält dagegen gerade diejenigen Knoten, die bereits bearbeitet worden sind. Der Algorithmus entfernt nun nach und nach Elemente aus der Todo-Liste und transformiert diese unter Berücksichtigung des Transformers, fügt die Transformationsergebnisse wiederum in die Todo-Liste ein und aktualisiert die Done-Liste mit dem aktuellen Knoten. Hierbei wird vor einem neuerlichen Betrachten der Todo-Liste jeweils darauf geachtet, diese zuvor um die Elemente der Done-Liste zu bereinigen, so dass ein bereits abgearbeitetes Element nicht erneut bearbeitet wird.

Abbildung 6.1 zeigt den zugehörigen Kontrollflussgraphen, der den Expansionsprozess auf den beiden Listen modelliert. Die Todo-Liste ist mit T, die Done-Liste mit D bezeichnet. Ferner beschreibt die Funktion *removeFst* eine Funktion, die eine Liste als Argument aufnimmt, das erste Element der Liste entfernt und dieses zurück gibt. Die Funktion *succs* bestimmt die Menge der Nachfolger vom Knoten n im zu expandierenden Graphen. Die Funktion *transform* schließlich beschreibt den Transformer, der einen Knoten n' des Modells, sowie eine Annotation A als Eingabe erhält und als Resultat das Transformationsergebnis liefert. Die Operation $T{:}a$ fügt ein Element a an eine Liste T an, während die Operation $T \setminus D$ alle Elemente aus T entfernt, die in der Liste D enthalten sind.

Die obere der beiden Schleifen iteriert über die Elemente der Todo-Liste und transformiert diese nach und nach. Die untere der beiden Schleifen dagegen

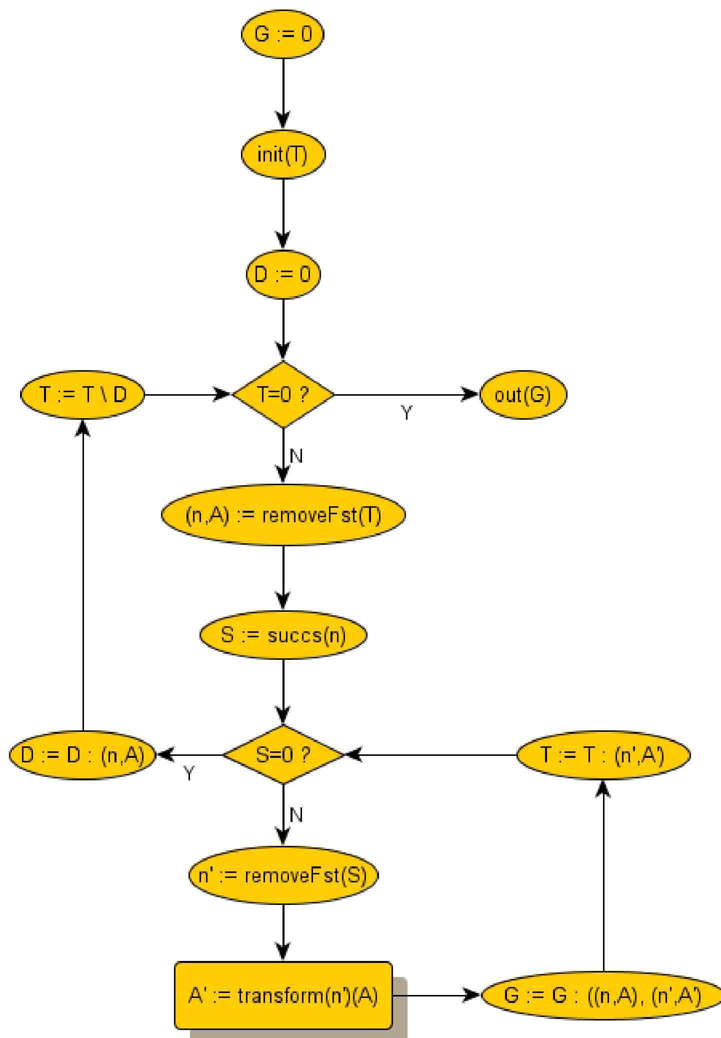


Abbildung 6.1: Imperative Implementierung von POE

iteriert über die Nachfolger eines einzelnen Knotens und transformiert diese der Reihe nach.

Da die imperative Variante mit einer beliebigen Implementierung simuliert werden kann, gehen wir an dieser Stelle nicht weiter auf diese Implementierung ein, sondern konzentrieren uns auf eine deklarative Implementierung, deren Realisierung die eben beschriebene Variante nach sich zieht. Ebenso könnte man sich auf diese Weise POE als einen Interpreter für haltende Programme vorstellen. Der als Kontrollflussgraph vorliegende Programmcode wird vom POE-Verfahren interpretiert und in eine endliche der Ausführung dieses Graphen entsprechende Form expandiert.

Interessant in diesem Zusammenhang ist der Umstand, dass diese imperative Implementierung ihrem Wesen nach nicht zwangsläufig imperativ ist. Es wurde lediglich ein Kontrollflussgraph angegeben, der wiederum seinem Wesen nach imperativ funktioniert. Die eigentliche Realisierung der Semantik dieses Kontrollflussgraphen wird jedoch durch POE selbst bestimmt. Da die Sicht des Programmes dem Programmierer selbst jedoch imperativ erscheint, soll dieses Verfahren an dieser Stelle auch als imperativ gelten. Denn auch ein in Prolog spezifiziertes Programm mag dem Programmierer als logisches Programm erscheinen, wenngleich die Implementierung der Programmiersprache selbst imperativ vorgehen mag. Es wird also die Sicht, wie sie dem Programmierer erscheint, als Paradigma zu Grunde gelegt, wenngleich sich hinter dieser Sicht mehre andere Paradigmen verbergen mögen, die dem Programmierer verborgen bleiben. Führe man sich das Kapitel 3 über verschiedene Sichten und Aspekte und eine mögliche Hierarchie auf denselben wieder in Erinnerung, so ist gerade dieses Beispiel der verschiedenen Ebenen einer Implementierung geradezu prototypisch für diese hierarchische Sicht. Denn welche verschiedenen Paradigmen kommen hier eigentlich zum Tragen? Zum einen gibt es eine imperative Sicht, die durch den Kontrollflussgraphen etabliert wird. Zum anderen wird die Funktionalität dieses Kontrollflussgraphen durch einen POE basierten Algorithmus realisiert. Dieser POE basierte Algorithmus wiederum mag etwa in einer funktionalen Sprache implementiert sein (was er in der Implementierung des Autors tatsächlich ist – vergleiche dazu auch den nächsten Abschnitt 6.1.2). Diese funktionale Implementierung mag wiederum in imperativen Programmcode übersetzt werden, der letztendlich etwa von einer virtuellen Maschine ausgeführt wird, welche wiederum auf der realen Hardware in imperativer Form läuft.

Leicht ist hier zu erkennen, dass schwer, wenn nicht unmöglich, auszumachen ist, welchem Paradigma eine konkrete Implementierung letztendlich folgt. Wie auch bei den verschiedenen Sichten, etwa des Model Checkings, kommt es hierbei auf einen eng umgrenzten Rahmen an, in den die Problemstellung eingebettet ist. Model Checking ist per se weder spiel- noch mengentheoretisch fundiert. Vielmehr kommt es auf den konkreten Anwendungsfall an, welche Sichtweise im Moment gerade dominiert. Ebenso bestimmt erst der Kontext, in dem eine konkrete Implementierung Beachtung findet, ihr Paradigma.

6.1.2 Deklarativ/Applikativ

Der nun folgende Abschnitt beschreibt die Realisierung des POE-Verfahrens in Form eines funktionalen Programmes. Als Implementierungssprache wurde hierzu *Scheme* gewählt. Scheme ist eine Mitte der 1970er Jahre am MIT[1] von Guy Lewis Steele Jr. und Gerald Jay Sussman entwickelter LISP[2]-Dialekt (siehe [SGLS75]), der eine eng an den von Alonzo Church vorgestellten Lambda-Kalkül (vgl. [Chu65]) angelehnte Realisierung darstellt. Die Möglichkeit, Closures (oder gar Continuations) als First-Class-Objects zu behandeln, ermöglichen in einer generischen Form geradezu beliebige Annotationen während des Transformationsprozesses, ohne dies über Quotierungs-, Maskierungs- oder Kodierungsmechanismen realisieren zu müssen. Ebenso repräsentiert Scheme den in dieser Arbeit schon an so vielen Stellen beschriebenen Wechsel der Sichten wie wohl kaum eine andere Sprache (wie jedoch viele andere Lisp-Dialekte[3]), indem sie Struktur und Verhalten in ihrer Syntax vereint. Alles ist eine Liste, das Programm selbst, wie auch die Daten, mit denen es umgeht. Es gibt also keine weitere (syntaktische) Unterscheidung, ob etwas nun ein Verhalten beschreibendes Stück Programmcode oder aber ein Datum darstellt.

Es sei noch angemerkt, dass Scheme keine rein funktionale Sprache ist, sondern ebenso einen Mechanismus zur Zustandsbehandlung besitzt. Es gibt jedoch eine *intuitive* (im Gegenteil zu auf *Konvention* beruhende) strikte Trennung (bei der Wahl der Notation wird ein Ausrufezeichen an eine den Zustand verändernde Funktion annotiert) zwischen imperativen und funktionalen Konzepten. Die vorliegende Implementierung kommt jedoch vollständig ohne Zustand aus und ist demnach eine rein funktionale Implementierung[4].

Bei der Implementierung standen zwei Kriterien im Vordergrund. Zum einen sollte die Implementierung leicht verständlich und nachvollziehbar sein, um durch einfache Anpassungen verschiedene Szenarien des Verfahrens auszutesten. Zum anderen sollte die Implementierung genügend effizient sein, um auch größere Fallbeispiele untersuchen zu können, die bereits aus sehr kleinen Modellen und nur leicht geschachtelten Eigenschaften entstehen. Aus diesem Grund greift die Implementierung auf eine generische – und demnach austauschbare – Schnittstelle zur Graphmanipulationen zurück.

Eine funktionale Realisierung des Algorithmus zeigt Abbildung 6.2. Die Funktion POE greift hierbei auf einen global verfügbaren zu expandierenden Graphen G zurück. Der expandierte Graph, wie auch der zu expandierende Graph werden hierbei als Struktur im Sinne einer aus *Atomen* (vgl. dazu auch Abschnitt 3) aufgebauten dem Wesen nach gegenständlichen Einheit aufgefasst. Eine Erweiterung dieses Graphen entspricht in diesem Sinne einer Änderung dieser Struktur. Knoten und Kanten werden hinzugefügt, indem Elemente dieser Struktur

[1] Massachusetts Institute of Technology

[2] LISP steht für List-Processing.

[3] Insbesondere ist interessanterweise gerade der Ursprung von Scheme, nämlich Common-Lisp und ANSI Lisp, eine eher imperative Sprache mit mächtigen Iterationskonzepten, wohingegen Scheme selbst die Iterationsmöglichkeiten fast ausschließlich aus der Rekursion bezieht.

[4] Nach dieser rein funktionalen Realisierung, wurden zwar aus Gründen der Performanz Änderungen durchgeführt, die explizit einen Zustand zuließen, doch soll an dieser Stelle lediglich dem Umstand Rechnung getragen werden, dass dieser nicht per se notwendig war.

$$
\begin{array}{lcl}
\mathrm{POE}(G,f,\emptyset,D) & = & \emptyset \\
\mathrm{POE}(G,f,(n,A):T,D) & = & \mathrm{POE}(G,f,(T\cup t_n^A)\backslash(D\cup(n,A)),D\cup(n,A)) \\
 & & \cup\, T_n^A
\end{array}
$$

wobei gilt
$t_n^A = \{(n', f(n)(A)) | n' \in \mathrm{succs}_G(n)\}$
$T_n^A = \{(n,A) \rightarrow x | x \in t_n^A\}$

Abbildung 6.2: Implementierung von POE mittels Graphen als Strukturen

$$
\begin{array}{l}
\mathrm{POE}(G,f,\emptyset,D) = \bot \\
\mathrm{POE}(G,f,(n,A):T,D) = \\
\quad \lambda(n^*,A^*). \\
\qquad \begin{array}{ll}
t_n^A & \text{falls } n* = n \text{ und } A* = A \\
\mathrm{POE}(f,T\cup t_n^A\backslash D, D\cup(n,A))(n*,A*) & \text{sonst}
\end{array}
\end{array}
$$

wobei gilt $t_n^A = \{(n', f(n)(A)) | n' \in G(n)\}$

Abbildung 6.3: Implementierung von POE mittels Graphen als Funktionen

derart verändert werden, dass die neue Struktur diese Änderung repräsentiert. Die verwendete Struktur besteht im Sinne einer einfacheren Notation in diesem Falle aus einer Tupelliste, wobei ein Tupel jeweils ein Paar aus Quell- und Zielknoten repräsentiert. Zu Beginn startet die Graphexpansion mit den initialen Knoten in der Liste zu erledigender Knoten T. Ist diese Menge leer, so besteht die Graphexpansion gerade aus diesem leeren Graphen. Genau dies beschreibt der erste Fall der Funktion. Ist die Menge noch zu expandierender Knoten nicht leer, so enthält sie mindestens einen noch zu expandierenden Knoten n mit einer an ihm geltenden Annotation A. Zunächst bestimmt die Menge T_n^A gerade jene Kanten von diesem Knoten n zu seinen Nachfolgern n' mit den neuen Annotationen $f(n')(A)$ für jeden Nachfolger n'. Ist der aktuelle Knoten verarbeitet, so wird er zum expandierten Graphen hinzugefügt, der gerade entsteht, wenn man den restlichen Graphen expandiert. Zu diesem Zweck, wird der bearbeitete Knoten in der Liste der bearbeiteten Knoten D gemerkt und die Liste der noch zu erledigenden Knoten um die neuen expandierten Nachfolger von n erweitert. Die Rekursion endet, wenn alle Knoten des Graphen erschöpfend expandiert wurden oder aber ein Knoten mit der gleichen Expansionsannotation ein zweites Mal erreicht wird – gerade solche Knoten werden nämlich gar nicht mehr in die Liste der noch zu bearbeitenden Knoten aufgenommen.

Eine alternative Realisierung, die nicht auf Tupellisten, sondern Funktionen beruht, zeigt Abbildung 6.3. Auch diese alternative Realisierung soll erneut verdeutlichen, wie wenig *eine* Sicht definitorischen Charakter für ein zu untersuchendes Gegenstandsfeld zu beanspruchen vermag, zeigt doch diese Variante, dass der beschriebene Expansionsprozess nicht mehr nur eine im Sinne eines Verhaltens beschriebene Veränderung einer gegenständlichen Größe ist (wie im oben beschriebenen auf Tupellisten basierenden Fall), als vielmehr eine einen anderen Prozess verändernde Aktivität. Wird der Graph gerade als Funktion

beschrieben und in diesem Sinne als Verhalten interpretiert, so bewirkt POE in dieser Sichtweise eine Verhaltensänderung in der Art, dass der ursprüngliche Graph (das ursprüngliche Verhalten) in einer Art und Weise geändert wird, die wiederum durch eine dritte verhaltensmodellierende Größe (nämlich gerade den Property Transformer) beschrieben wird. Gerade dieser Aspekt zeichnet jedoch auch den Hinweis darauf ab, dass der strukturorientierten Sicht eine implizite Verhaltensorientierung immanent zu sein scheint, die es erschwert, eine gerade nicht auf Verhalten basierte Sicht in der Beschreibung des zu expandierenden Graphen (in welche Begriffswelt nun auch immer dessen Bedeutung eingebettet sein mag) in diesem Kontext einzunehmen.

Die konkrete Realisierung der Funktion POE sieht in diesem Falle wie folgt aus. Ist die Liste (auch diese strukturelle Form ließe sich durch wenig Aufwand in eine funktionale Darstellung bringen) der noch zu bearbeitenden Knoten leer, so entspricht diese gerade der für keinen Wert definierten Funktion $\perp$. Ansonsten gibt es wie im oben geschilderten Fall einen noch zu expandierenden Knoten n mit einer entsprechenden Annotation A, die, wie bereits oben geschildert, verarbeitet werden. Anders ist lediglich die Verknüpfung der Struktur, die im vorherigen Fall als Vereinigung mit einer Menge realisiert wurde, in diesem Fall jedoch einer bedingten Funktionsapplikation entspricht, wie auch die Bestimmung der Nachfolgerknoten, die im vorherigen Fall eine den Graphen als Argument verarbeitenden Funktion `succs` benötigte, in diesem Fall jedoch einer Funktionsapplikation auf den zu expandieren Graphen G selbst entspricht.

Aus Gründen der Einfachheit wurde in der konkreten Implementierung mittels Scheme von diesen beiden Varianten (derer es mit Sicherheit noch mehrere gibt) dahingehend abstrahiert, dass der Graph selbst und die Zugriffe auf diesen in generischer Form gekapselt wurden. Eine vollständige Implementierung dieser Funktionen befindet sich im Anhang A.1.

Die Implementierung wurde durch die in Abschnitt 5.1 etablierten Beispiele getestet.

Laufzeitbetrachtungen

Laufzeitbetrachtungen gestalten sich in funktionalen Sprachen ebenso schwierig, wie in mathematischen Formeln. Diese beschreiben nämlich nicht gerade, die Lösung des Problems in Form von Schritten, als vielmehr die Reduktion des Hauptproblems auf Teilprobleme im Sinne eines induktiven Vorgehens. In einem bekannten Epigramm fasst A. J. Perlis (vgl. [Per82]) diese Situation wie folgt zusammen.

> A LISP programmer knows the value of everything, but the cost of nothing.

Ein zynisches Gegenargument auf dieses Zitat war häufig *A C programmer knows the cost of everything, but the value of nothing.* Doch welche Laufzeitbetrachtungen sind in diesem Sinne letztendlich trotz solch pessimistischer Einstellungen möglich? Als Maß für die Anzahl der Schritte in imperativen Sprachen kann

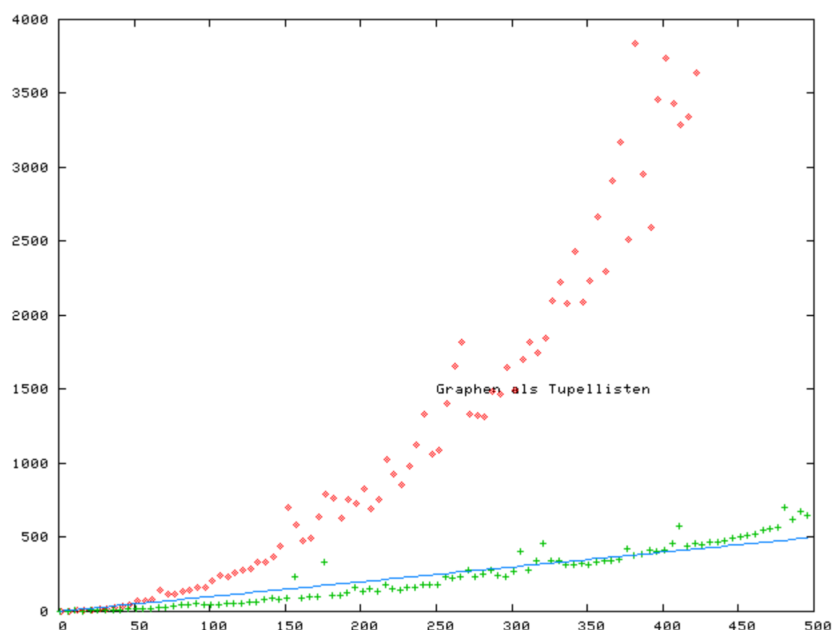


Abbildung 6.4: Laufzeitgegenüberstellung von Graphrepräsentationen.

die Anzahl der Funktionsapplikationen in funktionalen Sprachen als adäquates Gegenstück, als Grundlage von Laufzeitanalysen aufgefasst werden. In Anbetracht des POE-Verfahrens hängt die Laufzeit also wesentlich sowohl von der verwendeten Graphstruktur, wie auch von den Transformern ab.

Letztendlich stellt sich sich bei einer Realisierung der Graphstruktur mittels Tupellisten eine ernüchternde lineare average-case-Laufzeit (in der Anzahl der Knoten) für das Finden von Nachfolgern ein. Auch das Aktualisieren der Kanten verbucht mit einer linearen Laufzeit wiederum zu große Kosten. Bei Verwendung von Hash-Tabellen können diese Zugriffs- und Aktualisierungskosten jedoch bei sinnvoller Wahl einer geeigneten Hashfunktion auf eine amortisierte konstante average-case-Laufzeit gesenkt werden. Eine Gegenüberstellung der Laufzeiten von einer Implementierung, die auf Tupellisten basiert und jeweils einer Implementierung, die auf Hash-Tabellen basiert, zeigt Abbildung 6.4. Zugrunde gelegt wurde hierfür ein einfacher Graph mit lediglich einem Knoten mit einer reflexiven Kante, sowie einem Transformer, der während der Transformation eine Ganzzahl bis zu einer Obergrenze hoch zählt und dadurch beliebig große lineare Graphen erzeugt, deren Knotenanzahl von der Obergrenze abhängt. Die Anzahl der Knoten im expandierten Graphen wird in dieser Abbildung der Laufzeit in Millisekunden gegenübergestellt. Die Punkte in Diamantenform repräsentieren hierbei die Implementierung mittels Tupellisten, während die Pluszeichen auf die auf Hash-Tabellen basierende Realisierung verweisen. Die eingezogene Gerade beschreibt eine lineare Funktion und soll lediglich der Orientierung dienen.

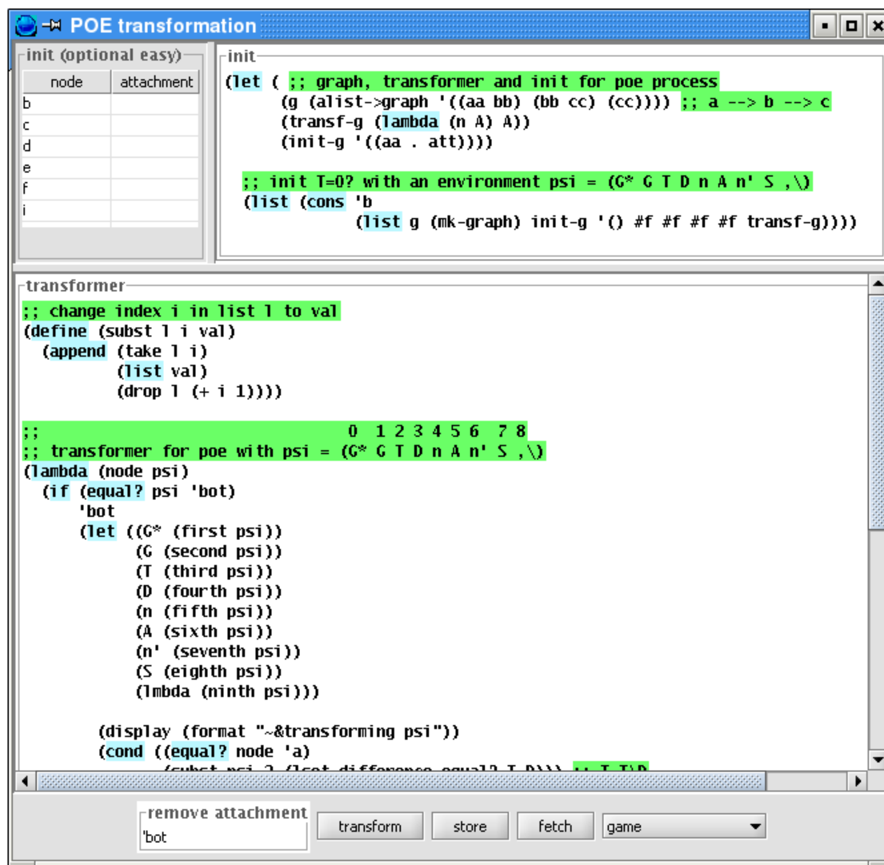


Abbildung 6.5: Plugin zur Expansion von SIB-Graphen

6.2 Integration in das Java ABC

Neben einer API zur Integration in beliebige Scheme basierte Drittanwendungen, entstand prototypisch ebenfalls eine Java basiert Variante. Diese war notwendig, um eine Integration mit dem Java ABC[jab] zu realisieren. Beim Java ABC handelt es sich um eine Modell gestützte und Java basierte graphische Entwicklungsumgebung, in der durch Komponenten (genannt SIB – kurz für *Service Independant Building-Blocks*) realisierte Basisfunktionalitäten in beliebigen Abstraktionsschichten zu größeren Anwendungen aggregiert werden. Die jeweilige intentionelle Semantik der modellierten Graphen (auch genannt SIB-Graphen) wird durch Plugins realisiert. Eine weitere Semantik für derartige Graphen wurde in Form eines POE-Plugins realisiert, die es ermöglicht, derartige Komponentengraphen unter Angabe eines Transformers zu expandieren.

SIB

Hierzu können die Intialbelegungen der SIBs entweder in einer tabellarischen Form oder aber durch direkte Ausführung von Scheme Code, der die Intialbelegung vornimmt, angegeben werden. Das Script für die Initialisierung muss als Rückgabewert eine Liste aus Tupeln, die jeweils aus Knoten und Initialwert bestehen, realisieren, die an den entsprechenden Knoten annotiert werden sollen.

Der Transformer selbst wird in Form eines kleinen Scheme-Scriptes abgelegt, das für jeden zu expandieren Knoten ausgeführt wird. Dies muss in Form einer Funktion auf zwei Argumenten realisiert werden, in der das erste Argument dem aktuellen zu expandierenden Knoten und das zweite Argument der bisherigen Expansionsannotation entspricht. Abbildung 6.5 zeigt einen Screenshot des Plugins.

Zusätzlich wurde ein Syntax-Highlighter realisiert, der bei der Eingabe eine visuelle Unterstützung für spezielle Schlüsselwörter bereitstellt.

Sowohl Annotionsinitialisierung, als auch der Transformer werden persistent mit dem Modell abgelegt und können beim erneuten Laden eines Modelles aus diesem extrahiert werden.

Bei der Expansion des Graphen wird der Transformer für die jeweiligen Knoten und Annotionsinformationen aufgerufen und erzeugt schließlich einen neuen SIB-Graphen, der dem ursprünglichen SIB-Graphen in seiner expandierten Version entspricht. Die Annotationsinformationen stehen in Form von nichtpersistenten Metainformationen (so genannten User-Objekten) an den Knoten dieses neuen expandierten SIB-Graphen zur Verfügung und können von anderen Plugins zur Weiterverarbeitung genutzt werden.

Realisiert wurde diese Integration in Java durch einen Java basierten Scheme-Interpreter namens SISC (etwa [Mil02]).

Ein ebenfalls in das Java ABC bereits integrierter Model Checker mit Namen GEAR[BR] wurde um die Möglichkeit erweitert, erfüllende Pfade auf dem Modell für Formeln aus $\exists\mu$ anzeigen zu lassen. Zu diesem Zweck kann eine Formel wie gewöhnlich in den Model Checker eingegeben werden. Die Überprüfung und Anzeige der möglichen Belege selbst wird dann jedoch durch das neu realisierte Plugin übernommen. Dafür kann das in Abbildung 6.6 gezeigte Fenster genutzt werden, um mögliche Belege dieser Formel im überprüften SIB-Graphen anzeigen zu lassen. Die einzelnen Folgen von Knoten, die die Formel belegen, werden in einer Tabelle angezeigt und können durch einen Klick darauf im SIB-Graphen selektiert werden. Diese Pfadansicht kann durch Angabe eines regulären Ausdruckes im Eingabefeld über der Tabelle eingeschränkt werden, um den Blick auf Pfade zu lenken, die für den aktuellen Anwendungsfall relevant erscheinen.

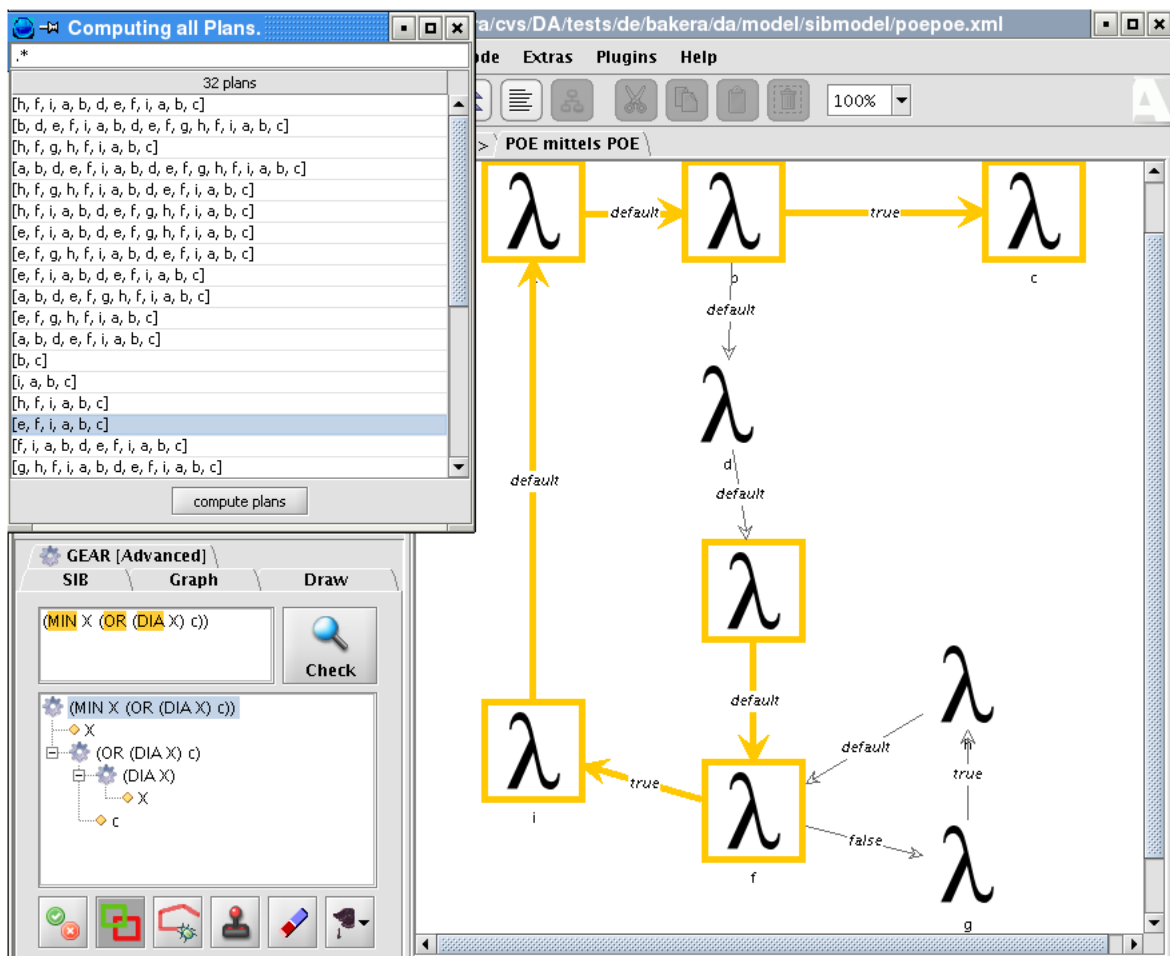


Abbildung 6.6: Plugin zur Erkundung verschiedener Pläne

Kapitel 7

Ausblick

Der folgenden Abschnitt liefert einen kurzen Überblick über mögliche Ansätze für über diese Arbeit hinausgehende Studien, sowie über neue Gedanken, die sich aus dem Umgang mit der Thematik ergeben haben.

Zunächst fällt auf, dass das vorgestellte Verfahren nur für eine eingeschränkte Logik (eben $\exists\mu$ und $\forall\nu$) konzipiert wurde. Weitergehende Überlegungen sollten daher eine Generalisierung auf den uneingeschränkten μ-Kalkül berücksichtigen.

Da das vom Anwender spezifizierte intendierte Verhalten nicht das spezifizierte Verhalten per se ist, sondern vielmehr aus einem Abstraktions- und Übersetzungsprozess in eine formale Sprache hervorgegangen ist, liegt hier eine Quelle möglicher Fehler, die die Etablierung des Model Checking-Verfahrens in praktischen realen Umgebungen und unter realistischen – und damit komplexen – Anwendungsbedingungen erschweren. Ziel weiterer Forschungen kann somit sein, die zur Zeit etablierten Logiken auf ihre Anwendbar- und Benutzbarkeit hin zu untersuchen bzw. die bestehende Lücke zwischen Spezifikations- und Implementierungssprachen zu verkleinern und schließlich zu beseitigen.

Das POE-Verfahren wurde speziell für ein Verhalten beschreibende Kontrollflussgraphen eingeführt und auch in dieser Arbeit weitgehend unter diesem Gesichtspunkt behandelt. Dennoch wurde schnell klar, dass sich Graphen, die andere Phänomene beschreiben, nur schlecht oder aber gar nicht mittels POE expandieren lassen. Sicherlich ließe sich das POE-Verfahren jedoch dahingehend verallgemeinern, um andere Graphen damit behandeln zu können. Beschrieben die Graphen etwa ein Schaltnetzwerk, so wird das Ergebnis eines Knotens in diesem Fall von all seinen Eingängen, also von all seinen Vorgängern bestimmt. POE expandiert jedoch in Richtung der Nachfolger, und diese zudem einzeln. Eine Generalisierung bestünde also darin, sowohl über die Nachfolger als auch über die Vorgänger abrollen zu können und zudem die Möglichkeit zu besitzen, nicht nur existenziell, sondern auch all-quantifiziert die Annotationen über Nachfolger bzw. Vorgänger bestimmen zu können.

Eine weitere Verallgemeinerung möge darin bestehen, das POE-Verfahren auf hierarchische Graphen auszuweiten und damit eine weitere Kategorie impliziter

Strukturbeschreibungen abzudecken. Hiermit sind dann jedoch explizit unendliche Strukturen möglich, die es im Besonderen zu beachten gilt. In diesem Fall ist die Terminierung nicht ausschließlich durch den Transformer bedingt, sondern hinge in diesem Fall auch maßgeblich von der Graphstruktur ab.

Die durch das POE-Verfahren und auch schon die durch die Erzeugung der Spielgraphen generierten Graphen wachsen in Bezug auf Modellgröße und Formeltiefe schnell immens. Hier gibt es sicher noch Möglichkeiten, diese Graphen entweder in ihrer Größe selbst zu reduzieren (insbesondere die expandierten Spielgraphen können eventuell direkt auch auf dem Modell selbst, ohne den Umweg über den Spielgraphen, expandiert werden) oder aber die Graphen implizit zu beschreiben. Letztere Variante ist sowohl für den Spielgraphen wie auch für den Expansionsprozess darauf bedenkenswert, so dass auch spielbasierte Model Checking Algorithmen von einer solchen symbolischen Darstellung der Spielgraphen profitieren können.

POE beschreibt in Bezug auf Spielgraphen eine Interpretation der selbigen als Kontrollflussstrukturen. Diese Interpretation durch POE kann jedoch umgangen werden, wenn die Spielgraphen selbst als Kontrollstrukturen direkt – ohne den Umweg über POE – interpretiert werden. Da POE in dem hier betrachteten Fall mehrere Ausführungssequenzen einer solcher Kontrollflussstruktur *gleichzeitig* betrachtet, müsste eine direkte Interpretation des Spielgraphen also eine nebenläufige Ausführung sein, die verzweigende Knoten als alternativ aber *gleichzeitig* auszuführende Kontrollflüsse auffasst. Ebenso müssten die Spielregeln – in diesem Falle die Gewinnbedingungen – einflussnehmend und spielentscheidend in die Logik dieser Kontrollflüsse kodiert werden.

Die bisherigen Model Checking Ergebnisse zielen auf Ja/Nein- oder aber Ja/Nein-Weil-Aussagen bezüglich des betrachteten *Modelles* ab. Nicht weiter betrachtet wird jedoch die Formelstruktur bei der Argumentation. Hier ist weiterhin zu untersuchen, welche begründenden oder widerlegenden Informationen sich aus derartigen Informationen für das Ergebnis des Model Checking Prozesses ziehen lassen. Das spielbasierte Model Checking erlaubt es, nicht nur Aussagen über die Modellstruktur als vielmehr auch über die Formelstruktur zu tätigen. So lässt sich etwa darüber, warum eine Eigenschaft in einer Struktur erfüllt ist oder nicht, nicht nur über die Struktur selbst argumentieren, als vielmehr auch über Aussagen, die in Form von Teilformeln in der Eigenschaft existieren. Ist eine Aussage bzgl. eines Modelles also nicht erfüllt, so kann dies zum einen am Modell liegen, oder aber andererseits auch an der Eigenschaft. Bisherige Ansätze sehen in Form von Fehlerpfaden immer die Modellierung als Argumentationsgrundlage für die Nicht-Erfüllbarkeit, nicht jedoch die Eigenschaft selbst. Erfüllbarkeit kann somit also entweder durch eine Änderung der Modellstruktur, durch eine Änderung der Eigenschaft, oder aber durch eine Mischung aus beidem erzielt werden. Nicht ganz klar scheint im Moment, in welcher Richtung die Ergebnisse eine intentionelle Anleitung zur Anpassung der Eigenschaft an die Modellierung, der Modellierung an die Eigenschaft, oder aber an eine Anpassung von Modellierung und Eigenschaft an die eigenen Vorstellung liefern kann.

Kapitel 8

Fazit

Die Arbeit hat nach einer kurzen Einführung in das Themengebiet Model Checking und einer Anwendung auf Spielgraphen, insbesondere Wert auf den Wechsel der Sichten gelegt, der bei diesem Vorgang geschehen ist. Nicht nur, weil sich dieser Wechsel in diesem Falle schon als fruchtbarer Kandidat erwiesen hat, sondern auch, weil ein weiterer Wechsel zum eigentlichen Ergebnis dieser Arbeit führen konnte. POE war hierbei in diesem Sinne qualitativ eine gleichwertige notwendige Änderung des Standpunktes bzgl. Spielgraphen, wie es die Spielgraphen ihrerseits für das klassische Model Checking Problem waren. Die im klassischen Modell des Model Checking noch implizit kodierten Informationen in den temporal-logischen Eigenschaften wurden durch den Spielgraphen expliziert. Die noch verbleibende implizite Kodierung der Lösungsmenge wurde nun anschließend durch POE weitergehend expliziert und ermöglichte eine adaptive Regulierung dieser erzeugten und im Sinne des Anwendungsfalles wünschenswerten Lösungsmenge.

Die Einteilung von stark impliziten hin zu immer expliziteren Beschreibungen soll noch einmal an der folgenden kleinen Taxonomie verdeutlicht werden.

- explizit
- semi-explizit
- semi-implizit
- implizit

Hierbei beschreiben die einzelnen Stufen die schrittweise Annäherung von einer Problembeschreibung über eine Lösungsbeschreibung bis zur faktischen Lösung selbst. Diese Einteilung ist nicht auf Model Checking allein beschränkt, sondern erlaubt für vielfältige Probleme und deren Lösungen eine Einteilung in eine dieser vier Kategorien.

Rein implizite Beschreibungen von Problemen begegnen uns häufig in rein mathematischen Kontexten. Hier wird die Struktur rein implizit beschrieben und

keinerlei Aussagen über die Erzeugung getroffen – manchmal noch nicht einmal über deren Existenz. So sagt die klassische Primzahldefinition nichts über deren Existenz oder deren Bestimmung aus, sondern beschreibt diese nur in Bezug auf ihre Teiler. Im Bereich des Model Checking ist der klassische Vertreter in dieser Kategorie wieder zu finden, der die zu untersuchende Eigenschaft (und bei hierarchischen Modellen eventuell auch die Modelle selbst) nur in impliziter Beschreibung für erlaubte Pfadstrukturen auf dem Modell enthält. Diese impliziten Beschreibungen sind insofern sinnvoll und wichtig, als dass sie es ermöglichen, im Sinne einer Spezifikation betrachtet zu werden und vielfältige Realisierungen dieser Spezifikation zu ermöglichen.

Semi-implizite Beschreibungen erlauben nun die Darstellung eines algorithmischen Lösungsmusters, das eine Berechnung oder Annäherung an die rein implizite Grundlage ermöglicht, nicht jedoch offensichtliche Aussagen zur Terminierung von Verfahren oder der Endlichkeit der Lösung tätigt. So ist der Spielgraph in diesem Sinne eine Explizierung, die jedoch nicht notwendigerweise endlich sein muss, wenn es die zu grunde liegenden Modellstruktur etwa nicht ist. Ebenso ermöglicht POE in diesem Zusammenhang eine Explizierung der Pfadstrukturen, nicht jedoch notwendigerweise in jedem Fall – eben dann nicht, wenn die verwendeten Transformer nicht terminieren.

Semi-explizite Beschreibungen garantieren nun, im Gegensatz zu den semi-impliziten Beschreibungen, die Terminierung der Verfahren bzw. die Endlichkeit der Strukturen (etwa der Spielgraphen oder aber der durch POE expandierten Graphen). Da eine Terminierung für diese Kategorie notwendig ist, disqualifizieren sich alle Turing-vollständigen Sprachen als allgemeingültige Kandidaten der Problemlösung dieser Kategorie. Ohne gleich in eine rein explizite und damit triviale Lösungsbeschreibung zu fallen, existieren erstaunlich wenige Kandidaten im Bereich der Programmiersprachen, die derartige Probleme zu lösen vermögen. Eine Möglichkeit ist etwa die von Hofstadter in [Hof79] vorgestellte Programmiersprache BLOOP[1], die lediglich beschränkte Schleifen zulässt. Ein weiterer Kandidat wäre ein POE-Verfahren, das sich auf Transformer mit endlichem Bild beschränkt. Ebenso in diese Kategorie fallen jedoch auch die vorgestellten POE-Graphen zu Bestimmung von Plänen. Diese erzeugen im Sinne einer Aufzählung von Pfadstrukturen nicht-zyklische implizite Beschreibungen in Form eines DAGs. Dieser lässt sich jedoch jederzeit explizieren, um schließlich eine rein explizite Lösungsmenge zu erhalten.

Schließlich beschreibt die Kategorie expliziter Problembeschreibungen, eine endliche Darstellung der Lösungsmenge selbst. Eine Lösung des Problems der Primzahlen zwischen fünf und zehn bestünde gerade in der Lösungsmenge $\{5, 7\}$. Wenngleich es scheinbar wenige Ansätze zu Problemlösungsverfahren in dieser Kategorie zu geben scheint, so findet sie dennoch ihre Anwendung etwa in Bereichen der Künstlichen Intelligenz, der unscharfen Logiken oder überall da, wo Probleme nicht immer zwangsläufig berechenbar sind, sich aber aus einem Fundus gesammelter Informationen Lösungen generieren lassen – etwa durch Immitationslernen eines Agenten während eines Turingtests.

Den Weg von impliziten Problembeschreibungen hin zu immer expliziteren Lösun-

[1]BLOOP steht für *bounded loop.*

gen ist ein der Informatik immanentes Vorgehen der Problemreduktion. Kleine (endliche) Teile eines unüberschaubaren Gesamtproblems werden in lokalen Strukturen behandelt, um somit Lösungen für das Gesamte zu erzeugen. Die Lösungen für solche Explizierungsprozesse werden hierbei seit jeher wiederum in impliziten Problem-Lösungs-Sprachen (Programmiersprachen) behandelt, die ihre impliziten Eingaben explizieren.

Im Kontext des Model Checkings ist dieser Explizierungsprozess in mehreren Stufen erfolgt. Das Ursprungsproblem geht von einer expliziten Graphrepräsentation aus (die wiederum ihrerseits einer impliziten Kontrollstruktur zugrunde liegen mag), die auf Eigenschaften hin untersucht werden soll, die wiederum implizit durch Ausdrücke einer Logik beschrieben werden. Diese Eigenschaft wird (zusammen mit der Modellstruktur) weiter in einen Spielgraphen expliziert. In diesem wird der implizite Charakter des Ursprungsproblems in die Regeln verlagert, wo er sich in Form von unendlichen Spielen niederschlägt. Hier greift nun das POE Verfahren an und expliziert aus diesen Spielen, alle interessanten (im Sinne der Definition aus Abschnitt 5) möglichen endlichen Spiele in einen DAG, die es ermöglichen, einen Sieger zu ermitteln. Dieser DAG wiederum wird auf die Modellkomponenten des Ursprungsproblems projiziert und ermöglicht letztendlich die Ermittlung einer expliziten Lösung im ursprünglichen Problemraum.

Auf diesem Weg ist gut zu erkennen, wie mit zunehmender Explizierung der Lösung die Größe der Darstellungen (von Modell und Formel über Spielgraph und expandiertem POE-DAG bis hin zu den aufgezählten Pfaden) anwächst, wenngleich das Problem in seiner Wurzel durch die Reduktion der Logik der Eigenschaften bereits verkleinert wurde. Dies ist jedoch auch nicht weiter verwunderlich, ist das Problem doch bereits für alle endlichen Pfade auf endlichen Modellstrukturen exponentiell in der Größe der Modelle. Vielmehr lag der Kern der Arbeit jedoch auch nicht auf Effizienzbetrachtungen, denn auf einer Fusion und damit Änderung verschiedener scheinbar gegensätzlicher Standpunkte in ein neuartiges Verfahren, das es nunmehr ermöglicht, den Lösungsraum für das Model Checking Problem exemplarisch zu erkunden, oder aber auch durch die Wahl eigener Transformer und eigener Logiken adaptiv zu generieren.

Anhang A

Implementierungsdetails

A.1 POE

Abbildung A.1 zeigt eine Implementierung des POE-Verfahrens in Scheme. Hierbei wurde, bis auf bei den hervorgehobenen Funktionen *empty-graph*, *succs* und *add-edges* lediglich der Sprachstandard, sowie einige Implementierungsvorschläge in Form von SRFIs (Scheme Request For Implementation, vgl. auch [srf]) verwendet. Die generische Graphschnittstelle zur Manipulation von Graphen erfordert lediglich die Implementierung der drei oben bereits genannten Funktionen zur Bestimmung der Nachfolger eines Knotens (*succs*), dem Hinzufügen einer Menge von Kanten bzw. Knoten in der Form einer Liste bestehend aus dem Quellknoten an der ersten und den Zielknoten an den übrigen Stellen (*add-edges*), sowie die Möglichkeit, auf einen leeren Graphen ((*empty-graph*)) zuzugreifen.

Die Menge der zu expandierenden Knoten wird in Form einer Liste aus Paaren übergeben, wobei jedes Paar jeweils aus einem Knoten des zu expandierenden Graphen, sowie einer an diesem Knoten initial geltenden Annotation besteht. Der Aufruf zur Bestimmung von Wurzeln mit Hilfe des Newtonverfahrens aus

```
(define (POE:trans transformer graph todos dones)
  (define (T ts ds)
    (if (null? ts)
        empty-graph
        (let* ((n (caar ts))
               (A (cdar ts))
               (transformed
                (map (lambda (succ)
                       (cons succ (transformer succ A)))
                     (succs n graph)))
               (new-edges (cons (car ts) transformed)))

          (add-edges new-edges
                     (T (lset-difference equal?
                                         (append (cdr ts)
                                                 transformed)
                                         (cons (car ts) ds))
                        (cons (car ts) ds))))))
  (T todos dones))


(define (poe graph transformer init)
  (POE:trans transformer
             graph
             init
             '()))
```

Abbildung A.1: POE als Scheme-Programm

Abschnitt 5.1.2 würde in diesem Falle also wie folgt aussehen.

```
(poe (add-edges '(b)
                (add-edges '(a a b) empty-graph))

     (lambda (n A)
       (cond ((equal? n 'b) (* A A))
             ((equal? n 'a)
              (let ((fault (abs (- 2 (* A A)))))
                (if (< fault 0.01)
                    A
                    (/ (+ (/ 2 A) A)
                       2))))
             (else 'BOT)))

     '((a . 1.0)))
```

Das erste Argument kodiert den Graphen der Eingabe – Knoten a übernimmt hierbei die Rolle des Iterationsschrittes, während im Knoten b die Probe in Form eines Quadrierungsschrittes durchgeführt wird. Die anonyme λ-Funktion beschreibt den Property Transformer, der abhängig vom aktuellen Knoten entweder die vorhandene Annotation quadriert oder aber eine gewichtete Mittelwertbildung vornimmt. Da der Graph für diesen Fall unendlich groß ist, weil das Verfahren die Wurzel nur beliebig annähern, nicht jedoch genau bestimmen kann, erfolgt ein Abbruch bei einer genügend kleinen Differenz zu einem Sollwert (in diesem Fall 0.01). Initialisiert wird das Verfahren mit einer 1 am Knoten a. Als Ergebnis liefert die Funktion ein Ergebnis entsprechend der zugrunde liegenden Graphrepräsentation, das dem expandierten Graphen entspricht.

Abbildung A.2 zeigt noch eine end-rekursive Fassung der Funktion. Diese bedient sich einer Hilfsfunktion `T-it`, die das Ergebnis des Expansionsprozesses im letzten Argument aufsammelt und schließlich zurück gibt, wenn die Liste der noch zu expandierender Knoten `ts` leer ist.

A.2 Transformer

Die im Abschnitt 5.1 über die POE Transformation vorgestellten Transformer wurden jeweils als Scheme Funktionen, die auf zwei Argumenten arbeiten, realisiert. Das erste Argument beschreibt den Knoten, den es zu expandieren gilt, das zweite Argument enthält schließlich die bisherige Transformationsinformation. Hierbei ist es wesentlich für die Transformer, mit welchen Initialisierungsinformationen die POE Expansion gestartet wurde.

Um die Darstellung übersichtlich zu halten und auf die wesentliche Funktionalität zu konzentrieren, wurden die Transformer nicht robust implementiert, sondern gehen von einer korrekten Benutzung aus.

Der erste Transformer `transformer-k` beschreibt die Implementierung der Funk- `transformer-k`

```
(define (poe:trans transformer graph todos dones)
  (define (T-it ts ds poe-g)
    (if (null? ts)
        poe-g
        (let* ((n (caar ts))
               (A (cdar ts))
               (transformed
                (map (λ (succ) (cons succ (transformer succ A)))
                     (succs n graph)))
               (new-edges (cons (car ts) transformed)))

          (T-it (lset-difference equal?
                                 (append (cdr ts) transformed)
                                 (cons (car ts) ds))
                (cons (car ts) ds)
                (add-edges new-edges poe-g)))))
  (T-it todos dones empty-graph))
```

Abbildung A.2: POE als end-rekursives Scheme-Programm

tion $f_{X_G^k}$. Dieser geht von einer Initialisierung mit einer Annotation aus, die sowohl den aktuellen, als auch die maximale Anzahl der zu explorierenden Knoten in einem Tupel (n, k) widerspiegelt. Dies wird durch ein Paar `(n . k)` realisiert, wobei `n` jeweils erhöht wird, wenn `k` noch nicht erreicht wurde.

```
(define (transformer-k node A)
  (if (eq? A 'BOT)
      'BOT
      (let ((n (car A))
            (k (cdr A)))
        (if (< n k) (cons (+ n 1) k)
            'BOT))))
```

`transformer-1`

Der nächste Transformer `transformer-1` realisiert die Funktion $f_{X_G^1}$ und geht von einer Initialisierung mit einer Zahl (reell, ganz-rational oder natürlich) aus. Diese Zahl wird in jedem Schritt erhöht, wenn das Zufallsereignis einen Fortgang widerspiegelt, und beendet im anderen Fall. Die Funktion `(random-real)` liefert hierbei eine Zufallszahl $0 < x < 1$ und ist in SRFI 27 (vgl. [srf]) spezifiziert.

```
(define (transformer-1 node A)
  (cond ((eq? A 'BOT) 'BOT)
        ((> (random-real) 0.5) (+ A 1))
        (else 'BOT)))
```

`transformer-~`

Der Transformer zur Ermittlung nicht-zyklischer Expansionen `transformer-~` entspricht einer Realisierung der Funktion $f_{X_G^{\sim}}$. Er geht von einer Initialisierung

mit einer Liste, die den Knoten selbst enthält, aus, welche sich in Scheme durch ein Paar `(n . (n))` für einen Knoten `n` realisieren lässt.

```
(define (transformer-~ node A)
  (cond ((eq? A 'BOT) 'BOT)
        ((member node A) 'BOT)
        (else (cons node A))))
```

`transformer-union-new`

Der schließlich letzte Transformer `transformer-union-new` realisiert die Funktion $f_{X_G^{\odot} \cup X_G^{\sim}}$. Die Annotation besteht hierbei aus einer Liste von besuchten Knoten, von denen einige möglicherweise bereits mehrfach aufgesucht wurden. Gesunde Knoten werden hierbei in einem Tupel besteht aus Knoten und einem ^ dargestellt. Die Funktion definiert zunächst eine Hilfsfunktion `mark^`, die ein übergebenes Argumente, das nicht mit ^ markiert ist, mit einem ^ markiert und ansonsten eine bereits vorhandene Markierung belässt. Anschließend wird eine der Funktion $f_{X_G^{\odot} \cup X_G^{\sim}}$ entsprechende Fallunterscheidung durchgeführt und die Elemente in der Annotation entsprechend ihrem Gesundheitszustand geändert bzw. neu aufgenommen.

Die Funktion `cut` dient der Spezialisierung eines oder mehrerer Parameter und ist in SRFI 26 (vgl. wiederum [srf]) spezifiziert.

```
(define (transformer-union-new node nodes)
  (define (mark^ x)
    (if (and (pair? x) (equal? (cdr x) '^))
        x
        (cons x '^)))

  (if (eq? nodes 'BOT)
      'BOT
      (let ((node^ (cons node '^)))
        (cond ((not (or (member node^ nodes)
                        (member node  nodes)))
               (cons node
                     (map (cut mark^ <>) nodes)))

              ((member node^ nodes)
               (cons node (delete node^ nodes)))

              (else 'BOT)))))
```

A.3 Teilformelbestimmung

Im Folgenden wird eine Funktion zur Bestimmung der Menge aller Teilformeln einer gegebenen Formel f vorgestellt. *atom*? beschreibt hierbei ein Prädikat, das zu *true* auswertet, genau dann, wenn das gegebene Argument einen atomaren nicht weiter zerlegbaren Ausdruck – also entweder eine atomare Proposition

oder eine Fixpunktvariable – darstellt. Dann ergibt sich die Menge $SF(\phi)$ – die Menge aller Teilformeln von ϕ – wie folgt.

$$SF(\phi) = \begin{cases} SF(\phi_1) \cup SF(\phi_2) \cup \{\phi\} & \text{falls} \quad \phi = \phi_1 \vee \phi_2 \text{ oder } \phi = \phi_1 \wedge \phi_2 \\ \{\phi\} & \text{falls} \quad atom?(\phi) \\ SF(\phi') \cup \{\phi\} & \text{falls} \quad \phi = \diamond\phi' \text{ oder } \phi = \Box\phi' \\ SF(\phi') \cup \{\phi\} & \text{falls} \quad \phi = \mu X.\phi' \text{ oder } \phi = \nu X.\phi' \end{cases}$$

Eine Implementierung einer solchen Funktion zeigt die folgende Realisierung in Scheme. Hierbei wird die Formel in einer Prefixdarstellung angenommen. Somit würde etwa die Formel $\mu X. \diamond X \vee true$ in dieser Notation in `(MIN X (OR true (DIA X)))` überführt.

```
(define (all-sub-exps φ)

  (define (atom? φ)
    (and (not (pair? φ))
         (not (null? φ))))

  (define (sub-exps Φ)
    (cond ((null? Φ)
           '())

          ((atom? (car Φ))
           (lset-adjoin equal?
                        (sub-exps (cdr Φ))
                        (car Φ)))

          (else
           (lset-adjoin equal?
                        (sub-exps (append (cdar Φ)
                                          (cdr Φ)))
                        (car Φ)))))

  (sub-exps (list φ)))
```

A.4 Bindungsbestimmung für Fixpunktvariablen

Der folgende Abschnitt beschreibt die Implementierung einer Funktion zur Bestimmung einer Teilformel, an die eine gegebene Fixpunktvariable X in einer Formel ϕ gebunden ist.

$$bound(X, \phi) = \begin{cases} false & \text{falls } X \in AP \vee X \in FIX \\ \phi & \text{falls } \phi = \sigma X.\phi' \text{ mit } \sigma \in \{\mu, \nu\} \\ \hat{\exists} a \in args(\phi) : bound(X, a) & \text{sonst} \end{cases}$$

Hierbei liefert $args$ gerade die Argumente der Funktion ϕ. Der Quantor $\hat{\exists}$ wertet nicht in jedem Fall zu einem boolschen Wert $true$ oder $false$ aus, sondern liefert eine gültige – in diesem Zusammenhang nicht zu $false$ auswertende – Belegung der Variablen, über die die Quantifizierung durchgeführt wird, sofern eine solche existiert. Gibt es keine solche Belegung, so wertet $\hat{\exists}$ ebenfalls zu $false$ aus. So gilt etwa $(\hat{\exists}x \in \{0,1\} : 0) = 0$, andererseits gilt jedoch $(\hat{\exists}x \in \{0,1\} : 2) = false$.

Eine Implementierung der obigen Funktion in Scheme ist im Folgenden zu sehen.

```
(define (bound fix φ)
  (cond ((not (pair? φ))
         #f)

        ((and (member (first φ) '(max min))
              (equal? fix (second φ)))
         φ)

        (else (any (cut bound fix <>)
                   (cdr φ)))))
```

Hierbei wird die Formel ϕ wie auch schon in Abschnitt A.3 in einer Prefixnotation-Notation angenommen. `any` übernimmt hierbei die Rolle von $\hat{\exists}$ und `cut` spezialisiert ein Argument der übergebenen Funktion.

Anhang B

Beweise

Die in einigen der vorherigen Abschnitte nur argumentativ unterstützten, jedoch nicht formal geführten oder zu technischen Beweise, werden an dieser Stelle nun ausführlich formuliert.

B.1 Wohldefiniertheit von POE

Die in Abschnitt 5.1.5 vorgestellte Realisierung von POE mittels einer funktionalen Implementierung terminiert nicht in jedem Fall. Ob das Verfahren terminiert, hängt zum einem von der zu expandierenden Modellstruktur, und zum anderen vom verwendeten Transformer ab. Die Modellstruktur hat nur in sofern einen Einfluss, wenn sie nicht endlich ist und somit über unendliche lange Ketten in dieser Struktur eine Terminierung verhindert. Etwas subtiler gestaltet sich das Terminierungskriterium bzgl. des Transformers. Hierbei ist leicht einzusehen, dass eine Terminierung stark vom Wertebereich des Transformers abhängt. Besitzt der zu expandierende Graph etwa nur einen Knoten mit einer reflexiven Kante und soll die Expansion mit dem Transformer $f(n)(A) = A + 1$ und einer Initialisierung des Knotens mit einer beliebigen Ganzzahl durchgeführt werden, so erzeugt der Transformer immer wieder neue Knoten, die es ihrerseits zu expandieren gilt, ohne eine Terminierung zu garantieren.

Nun soll gezeigt werden, dass für endliche Wertebereiche des Transformers das POE Verfahren stets terminiert. Wenngleich diese Einschränkung unter Umständen zu stark erscheint, so lässt sie sich der Beweis dennoch einfach und direkt aus der Funktionsdefinition ableiten. Hierbei beschreibt $ran(f)$ den Bildbereich unter dem Definitionsbereich von f, also das Bild der Funktion f. Dagegen beschreibt $ran(G)(f)$ das Kreuzprodukt aus Modell und dem Bild von f, es gilt also $ran(G)(f) = \{(n, A) | n \in G_V, A \in ran(f)\}$.

Zur Erinnerung folgt an dieser Stelle erneut die Definition von POE aus Abbildung 6.2 von Seite 90.

$$\begin{aligned}\mathrm{POE}(G,f,\emptyset,D) &= \emptyset\\ \mathrm{POE}(G,f,(n,A):T,D) &= \mathrm{POE}(G,f,(T\cup t_n^A)\backslash(D\cup(n,A)),D\cup(n,A))\\ &\quad\cup T_n^A\end{aligned}$$

Hierbei gilt $t_n^A = \{(n', f(n)(A))|n' \in \mathrm{succs}_G(n)\}$ und $T_n^A = \{(n,A) \to x|x \in t_n^A\}$.

Satz 5 *Wenn $|ran(G)(f)| < \infty$ und $T \subseteq ran(G)(f) \supseteq D$, dann gilt*

$$\mathrm{POE}(G,f,T,D) \neq \bot.$$

Beweis *Falls $T = \emptyset$, gilt $\mathrm{POE}(G,f,\emptyset,D) = \emptyset \neq \bot$. Der andere Fall wird nun mittels Induktion über $|ran(G)(f)\backslash D|$ gezeigt. Für den Induktionsanfang gelte also $|ran(G)(f)\backslash D| = 0$. Also gilt $ran(G)(f) \subseteq D$. Zusammen mit $D \subseteq ran(G)(f)$ folgt $D = ran(G)(f)$. Dann folgt jedoch*

$$\begin{aligned}\mathrm{POE}(G,f,(n,A):T,D) =\\ \mathrm{POE}(G,f,(T\cup t_n^A)\backslash(D\cup(n,A)),D\cup(n,A))\cup T_n^A =\\ \mathrm{POE}(G,f,(T\cup t_n^A)\backslash ran(G)(f),ran(G)(f))\cup T_n^A =\\ \mathrm{POE}(G,f,\emptyset,ran(G)(f))\cup T_n^A =\\ T_n^A\end{aligned}$$

Der vorletzte Schritt folgt aus $T \subseteq ran(G)(f)$ und $t_n^A \subseteq ran(G)(f)$. Für den Induktionsschritt sei nun also $|ran(G)(f)\backslash D| = n$. Wiederum folgt

$$\begin{aligned}\mathrm{POE}(G,f,(n,A):T,D) =\\ \mathrm{POE}(G,f,T\cup t_n^A\backslash(D\cup(n,A)),D\cup(n,A))\cup T_n^A\end{aligned}$$

Nun gibt es zwei Möglichkeiten. Entweder ist der zu expandierende Knoten (n,A) bereits in D enthalten oder nicht.

$(n,A) \notin D$ *– Dann folgt die Behauptung mittelbar, denn dann gilt $(n,A)\cup D \supset D$ und insbesondere auch $|(n,A)\cup D| > |D|$ und damit auch $|ran(G)(f)\backslash D| > |ran(G)(f)\backslash D\cup(n,A)|$ und der Induktionsschluss lässt sich vollziehen.*

$(n,A) \in D$ *– Dann folgt*

$$\mathrm{POE}(G,f,(T\cup t_n^A)\backslash D,D)\cup T_n^A$$

Falls gilt $T\cup t_n^A\backslash D = \emptyset$, folgt direkt $\mathrm{POE}(G,f,\emptyset,D)\cup T_n^A = T_n^A \neq \bot$. Ansonsten folgt für geeignete Wahl von n', A' und T'

$$\begin{aligned}\mathrm{POE}(G,f,(n',A'):T',D)\cup T_n^A =\\ \mathrm{POE}(G,f,(T'\cup t_{n'}^{A'})\backslash(D\cup(n',A')),D\cup(n',A'))\cup T_n^A\end{aligned}$$

Insbesondere gilt nun jedoch auch $(T'\cup t_{n'}^{A'})\backslash(D\cup(n',A'))\cap(D\cup(n',A')) = \emptyset$ und die Argumentation lässt sich erneut, dann jedoch wie unter $(n,A) \notin D$ vollziehen □

B.2 Terminierungsnachweis von $f_{X_G^1}$

Der nachfolgende Beweis beschreibt den bereits informal in Abschnitt 5.2.2 vorgestellten Nachweis über die Terminierung des Transformers $f_{X_G^1}$, der eine beliebige Exploration beschreibt.

Lemma 3 *Ein POE-Algorithmus, der einen beliebigen Graphen mit dem Transformer*

$$f_{X_G^1}(n)(A) = \begin{cases} A+1 & \textit{falls } \mathrm{rand}() > 0.5 \\ \perp & \textit{sonst} \end{cases}$$

expandiert, terminiert.

Beweis *Angenommen, der Transformer* $f_{X_G^1}$ *setze die Transformation jeweils mit einer Wahrscheinlichkeit von* $\frac{1}{k}$ *für eine beliebige aber feste positive ganzzahlige Größe k fort. Dann beträgt insbesondere die Wahrscheinlichkeit für eine zweimalige Ausführung des Transformers* $\frac{1}{k^2}$ *und die Wahrscheinlichkeit für eine n-malige Ausführung* $\frac{1}{k^n}$. *Schließlich beträgt jedoch die Wahrscheinlichkeit für eine Nicht-Terminierung* $\lim_{n\to\infty} \frac{1}{k^n} = 0$. *Ergo muss die Transformation terminieren* □

Abbildungsverzeichnis

2.1 Beispiel einer Kripke Struktur. 13

2.2 Ein Modell eines Kaffee- und Tee-Automaten. 14

2.3 Beispiel eines Kripke Transitions Systems. 16

2.4 Die Semantik einer Linearzeitlogik 17

2.5 Ein Anschauung für Modalitäten. 18

2.6 Semantik von HML 19

2.7 Anschauung für die Operatoren $\Box$ und $\Diamond$ 20

2.8 Semantik von CTL 21

2.9 Eine anschauliche Vorstellung der CTL-Operatoren. 22

2.10 Semantik des modalen μ-Kalküls 24

2.11 Ein Vergleich der Ausdrucksstärken verschiedener Temporallogiken. 25

2.12 Operatoren von $\exists\mu$. 27

4.1 Ein kleines Beispiel für einen Spielgraphen. 40

5.1 Eingabenvalidierung einer Funktion auf 0 und 1. 52

5.2 Validierung von Eingaben für eine boolsche *und*-Funktion. 53

5.3 Der zu expandierende Graph für Newtons Iterationsverfahren . . 55

5.4 Der expandierte Graph für Newtons Iterationsverfahren 56

5.5 Transformer zur Expansion eines Modelles in einen Spielgraphen 57

5.6 Beispiel eines Modelles und einer Formel aus $M\mu$, die mittels POE in eine Spielgraphen expandiert werden sollen. 58

5.7 Der mittels POE expandierte Spielgraph. 58

5.8 Kontrollflussgraph für Property Oriented Expansion 59

5.9 Transformer für den Kontrollflussgraphen in Abbildung 5.8 . . . 61

5.10 Expandierter Kontrollflussgraph für Property Oriented Expansion 62

5.11 Eine Taxonomie von Graphexplorationen 68

5.12 Eine Übersicht über die Mengeninklusionsbeziehungen verschiedener Graphtraversierungsmöglichkeiten. 68

5.13 Beispiel für Explorationsmöglichkeiten auf einem Graphen. . . . 68

5.14 Property Transformer für die Plantaxonomie. 70

5.15 Beispielexploration mit Transformer $f_{X_G^{\odot} \cup X_G^{\sim}}$. 72

5.16 Beispielexpansion eines Spielgraphen 78

5.17 Spezifikation von Toolabhängigkeiten durch einen Kompatibilitätsgraphen 80

5.18 Expandierter Toolabhängigkeitsgraph. 80

5.19 Expandierter Toolabhängigkeitsgraph nach Elimination redundanter Knoten. 81

5.20 Expandierter Toolabhängigkeitsgraph für einmalige Ausführung von Tools. 82

5.21 Expandierter Toolabhängigkeitsgraph nach Elimination redundanter Knoten für einmalige Ausführung von Tools. 83

6.1 Imperative Implementierung von POE 87

6.2 Implementierung von POE mittels Graphen als Strukturen . . . 90

6.3 Implementierung von POE mittels Graphen als Funktionen . . . 90

6.4 Laufzeitgegenüberstellung von Graphrepräsentationen. 92

6.5 Plugin zur Expansion von SIB-Graphen 93

6.6 Plugin zur Erkundung verschiedener Pläne 95

A.1 POE als Scheme-Programm 104

A.2 POE als end-rekursives Scheme-Programm 106

Literaturverzeichnis

[Alu99] Rajeev Alur. Timed automata. In *CAV*, pages 8–22, 1999.

[BR] Marco Bakera and Clemens Renner. GEAR – Game Based Easy and Reverse Model Checking. http://jabc.cs.uni-dortmund.de/modelchecking/.

[CD89] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 428–437, London, UK, 1989. Springer-Verlag.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT Press Cambridge, Mass, 1999.

[Chu65] Alonzo Church. The calculi of lambda-conversion. *Annals of mathematics studies ; 6 ; 6*, pages II, 82 S., 1965.

[EH86] E. Allen Emerson and Joseph Y. Halpern. 'sometimes' and 'not never' revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[EJS93] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model-checking for fragments of μ-calculus. In *CAV*, pages 385–396, 1993.

[Fey65] Curry; Haskell; Feys. Combinatory logic, 1965.

[Gra03] Ronald L. Graham. *Concrete mathematics : [a foundation for computer science].* Boston [u.a.] : Addison-Wesley, 2. ed., 13. print. edition, 2003.

[Hen94] Thomas A. Henzinger. Temporal proof methodologies for timed transition systems. In *Information and Computation*, volume 112, pages 273–337, 1994.

[Hof79] Douglas R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid.* Basic Books, Inc., New York, NY, USA, 1979.

[jab] Java ABC Framework. http://www.jabc.de.

[Kle52] S.C. Kleene. *Introduction to metamathematics*. Van Nostrand New York, 1952.

[Lam80] Leslie Lamport. 'sometime' is sometimes 'not never' - on the temporal logic of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185, New York, NY, USA, 1980. ACM Press.

[Mil02] S.G. Miller. SISC: A Complete Scheme Interpreter in Java. Technical report, Technical Report, January 2002.

[MOSS99] M. Müller-Olm, D. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In G. File A. Cortesi, editor, *Proc. of Static Analysis Symposium (SAS'99), Venice, Italy*, volume 1694 of *Lecture Notes in Computer Science (LNCS)*, pages 330–354, Heidelberg, Germany, September 1999. Springer-Verlag.

[Per82] Alan J. Perlis. Special feature: Epigrams on programming. *SIGPLAN Not.*, 17(9):7–13, 1982.

[Rob74] Abraham Robinson. *Non-standard analysis*. Studies in logic and the foundations of mathematics. Amsterdam [u.a.]: North-Holland [u.a.], rev. ed. = [2. ed.] edition, 1974.

[SB92] B. Steffen and O. Burkart. Model checking for context-free processes. In *CONCUR'92, Stony Brook (NY)*, volume 630 of *Lecture Notes in Computer Science (LNCS)*, pages 123–137. Springer-Verlag, 1992.

[Sch24] M.S. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92(3):305–316, 1924.

[SGLS75] Gerald Jay Sussman and Jr. Guy Lewis Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, December 1975.

[srf] Scheme Requests for Implementation. http://srfi.schemers.org/.

[Ste91] Bernhard Steffen. Data flow analysis as model checking. In A.R. Meyer T. Ito, editor, *Theoretical Aspects of Computer Science (TACS'91), Sendai (Japan)*, volume 526 of *Lecture Notes in Computer Science (LNCS)*, pages 346–364, Heidelberg, Germany, September 1991. Springer-Verlag.

[Ste96] Bernhard Steffen. Property oriented expansion. In *Proc Int. Static Analysis Symposium (SAS'96), Aachen (Germany)*, volume 1145 of *Lecture Notes in Computer Science (LNCS)*, pages 22–41, Heidelberg, Germany, September 1996. Springer-Verlag.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.