Fehlerdiagnose beim Model-Checking durch animierte Strategie-Synthese

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von
Haiseung Yoo

Dortmund
2007

Inhaltsverzeichnis

1 Einleitung **1**

2 Temporale Logiken **7**

2.1 Notationen 8

2.2 CTL^* und ihre Teillogiken 8

2.3 Der modale μ-Kalkül 16

3 Model-Checking **25**

3.1 Fixpunktsatz 26

3.2 Model-Checking für den modalen μ-Kalkül 31

3.3 Gleichungssysteme 42

3.4 Abhängigkeitsgraphen 47

3.5 Tableausystem 53

4 Spielbasiertes Model-Checking **57**

4.1 Begriffe und Definitionen über Spiele 57

4.2 Strategie-Synthese 64

4.3 Korrektheit 83

4.4 Komplexität 89

5 Optimierungen und Verbesserungen **103**

5.1 Optimierung für Alternierungstiefe 2 103

5.2 Verbesserungen für höhere Alternierungstiefe 107

5.3 Korrektheit 113
5.4 Diskussion 119

6 Implementierung und Benutzung des Tools 123
6.1 Eingabeformel 124
6.2 Eingabe des Modells 128
6.3 Aufbau der Spielgraphen 129
6.4 Spielsteurung 137

7 Zusammenfassung und Ausblick 141

Literaturverzeichnis 143

Stichwortverzeichnis 148

Kapitel 1

Einleitung

Die Informationstechnik hat in den letzten Jahren eine äußerst rasante Entwicklung vollzogen. Unsere Gesellschaft wandelt sich dabei von einer Industriegesellschaft zu einer Informationsgesellschaft. Viel Hard- und Software wird entwickelt und in verschiedenen Bereichen eingesetzt. Da sie auch in kritischen Gebieten etwa für den Betrieb von Kernkraftwerken oder für die Kontrolle von Luftfahrzeugsverkehr eingesetzt werden können, ist es sehr wichtig, die Sicherheit zu gewährleisten. Eine traditionelle Methode dafür ist, dass man die Fälle, die denkbar auftreten können, berücksichtigt und die Vorgänge simuliert. Mit dem Ergebnis wird es dann festgestellt, ob es einem Fehler unterliegt. Ein Vorteil solcher „Simulation und Testen“ Verfahren ist, dass die entwickelte Hard- oder Software direkt eingesetzt wird, weshalb man normalerweise keine weitere Hard- oder Software braucht.

„Simulation und Testen“ hat aber viele Nachteile. Die Anzahl der Fälle, die man untersuchen will, kann enorm groß sein, so dass man nicht alle Fälle behandeln kann. Auch wenn man es kann, dauert die Testphase lang, und die Kosten werden entsprechend hoch. Meistens versucht man deshalb die Fälle auszusuchen, die in Wirklichkeit vorkommen können. Man kann hierbei ein oder mehrere Fälle übersehen, die in einer normaler Situation nicht auftreten, aber trotzdem nicht ausgeschlossen werden dürfen. Solche menschliche Denkfehler können sogar einen erheblichen Schaden verursachen.

Model-Checking ist hingegen ein Verfahren zur rechnergestützten, vollautomatischen Verifikation von Hard- sowie Softwaresystemen. Das zu prüfende Hard- oder Softwaresystem wird mit einem endlichen Automaten bzw. einem Transitionssystem modelliert und die gewünschten Eigenschaften werden durch eine temporallogische

Formel beschrieben. Dann wird nachgeprüft, ob die Formel im Modell erfüllt ist. Die Interpretation eines Hard- oder Softwaresystems als ein Modell sowie die Formulierung der Eigenschaften in einer temporalen Logik ist zwar eine zusätzliche Arbeit, jedoch kann das Verfahren dann vollautomatisch durchgeführt werden, so dass Zeit und Kosten insgesamt erheblich gespart werden.

In der Entwurfsphase eines technischer Systems wie Hardwaredesigns ist es sehr wichtig, so frühzeitig wie möglich die Fehler zu entdecken, damit die Entwicklungszeit kurz wird und dadurch die Kosten reduziert werden. Model-Checking ist dafür hervorragend geeignet, da es völlig automatisch ohne Interaktion der Benutzer durchgeführt werden können. Wird ein Fehler entdeckt, dann wird ein Gegenbeispiel generiert, das zur Analyse des Fehlers verwendet werden kann.

Noch ein großer Vorteil von Model-Checking ist, dass alle mögliche Fälle vollständig berücksichtigt werden. Die menschlichen Denkfehler werden somit vermieden. U.a. wegen solcher Vorteile wird das Model-Checking in vielen Bereichen eingesetzt, z.B. beim Entwurf von digitalen Schaltwerken sowie von Kommunikationsprotokollen.

Zu den typischen temporalen Logiken gehören u.a. CTL(Computational Tree Logic), LTL(Linear Temporal Logic), CTL^* und der modale μ-Kalkül. Der modale μ-Kalkül ist eine Fixpunktlogik, die wegen ihrer hohen Ausdrucksmächtigkeit immer mehr Interesse erweckt. Viele temporale Logiken, wie z.B. CTL, LTL und CTL^* lassen sich in Formeln des modalen μ-Kalküls übersetzen. In dieser Arbeit wird der modale μ-Kalkül als Mittel zur Beschreibung der gewünschten Eigenschaften eingesetzt.

Inzwischen sind viele Model-Checking-Algorithmen entworfen werden. Der Hauptanlass der meisten Entwürfe war zum einen die Suche eines Algorithmus mit einer möglichst guten Laufzeitkomplexität und zum anderen die Suche einer Lösung des sogenannten Zustandsexplosionsproblems.

Alle bisher bekannte Model-Checking-Algorithmen für den modalen μ-Kalkül besitzen eine exponentielle Laufzeitkomplexität bzgl. der Alternierungstiefe der Fixpunktoperatoren, mit denen eine Aussage über größte sowie kleinste Fixpunkte formuliert werden. Die Alternierungstiefe der Eingabeformel ist zwar in der Regel nicht so groß, dass sie kleiner als 3 ist, aber kann das Modell enorm groß sein. In der Regel unterscheidet man deshalb zwei Anwendungsgebiete des Model-Checkings, je nachdem, wie groß das Modell sein kann.

Für den Fall, dass das Modell über Hunderttausende von Knoten enthält, können

symbolische Model-Checking-Algorithmen Abhilfe leisten, in denen *BDDs* (Binary Decision Diagramms) verwendet wird. Man stellt Graphenstrukturen, die während der Berechnung eines Model-Checking-Algorithmus in Bezug auf boolesche Funktionen erzeugt werden, in einer kompakten (speichersparenden) Form BDD dar. Die Abarbeitung dieser Algorithmen erfolgt im Allgemeinen *bottom-up*, d.h. eine Formel wird erst bewertet, nachdem alle Teilformeln bewertet werden.

Wenn die Erfüllbarkeit einer Formel nicht für alle Knoten sondern lediglich für einen Knoten im Modell geprüft werden soll, kann ein lokaler Model-Checking-Algorithmus eingesetzt werden, in dem die Bewertung der Knoten nach Bedarf erweitert wird. In einem günstigen Fall stoppt ein lokaler Model-Checking-Algorithmus, bevor alle Knoten bzgl. die Eingabeformel bewertet werden, und liefert das Ergebnis.

Ist das Modell relative klein, so dass er beispielsweise weniger als Zehntausende Knoten enthält, dann braucht man nicht unbedingt symbolische oder lokale Model-Checking-Algorithmen einzusetzen. In dem Fall ist es von großer Bedeutung, eine ausführliche und leicht nachvollziehbare Fehlerdiagnose zu haben, damit man möglichst schnell die Ursache des entstehenden Fehler lokalisieren kann. Einige Model-Checkers liefern zwar ein Gegenbeispiel als Information über den Fehlern. Jedoch sind solche Informationen meistens nicht vollständig und schwer zugänglich für den Anwender. Die vollständigen Fehlersituationen graphisch darzustellen ist in vielen Fällen nicht möglich, da eine Beweisstruktur eines unendliche Baumes wegen Nichtdeterminismus gewisser temporaler Operatoren dabei entsteht.

Einen interessanten Aspekt bietet eine spieltheoretische Annährung an das Model-Checking-Problem. Das Model-Checking-Problem wird dabei als ein Spiel mit zwei Spielern interpretiert, so dass ein Spieler die Erfüllbarkeit der Eingabeformel im Modell zu beweisen versucht, während der andere Spieler das Gegenteil versucht. Die beiden Spieler bestimmen abwechselnd die nächsten Knoten und je nachdem, ob die (Teil-)Formel von einem Knoten aus erfüllt wird, steht der Gewinner auf dem Knoten fest.

In einem Spiel-Algorithmus wird nicht nur die Erfüllbarkeit der Formel prüft, sondern auch die Begründung des Resultates in den Knoten speichert. Man erhält zwar wie bei Model-Checking eine Beweisstruktur eines unendlichen Baumes. Jedoch sind die Fehlersituationen rekonstruierbar ohne erneute Ausführung des Erfüllbarkeitstests, da die Informationen über die Begründung des Ergebnisses mitberechnet und

gespeichert werden. Bei der Fehlerdiagnose wird dann jeweils nur ein Pfad betrachtet, der interaktiv mit einem Tool konstruiert wird, das die Fehlersituation aufdeckt.

In dieser Arbeit wird ein neuer Algorithmus für den modalen μ-Kalkül entwickelt. Besonderheit unseres Spiel-Algorithmus ist, dass er auf einem Model-Checking-Algorithmus basiert, in dem größte sowie kleinste Fixpunkte iterativ berechnet werden. Bisher wurde vermutet, dass die Laufzeitkomplexität eines derartigen Spiel-Algorithmus viel schlechter als von Model-Checking-Algorithmus ist, da die Speicherverwaltung aufwendiger wird.

Beim Durchlauf der Iteration im Model-Checking-Algorithmus wird die Erfüllbarkeit von Teilformeln für die Knoten wiederholt geprüft und als Bewertung der Knoten temporär gespeichert. Sobald die Rahmenbedingung der Bewertung geändert wird, werden die Informationen über die Bewertung gelöscht und neu berechnet. Problem bereitend ist, dass die Informationen über die Begründung der Erfüllbarkeit durch Zurücksetzung der Bewertung sowie Veränderung der Rahmenbedingung verlorengehen. Bisher wurde deshalb an die Spiel-Algorithmen mit einem anderen Ansatz herangegangen.

Wir zeigen in dieser Arbeit, dass man die Informationen über die Bewertung und ihre Begründung selektiv beim Durchlauf der Iteration beibehalten kann, wodurch das Spiel-Problem gelöst wird. Es ist erstaunlich, dass die Laufzeitkomplexität unseres Spiel-Algorithmus nicht schlechter als von Model-Checking-Algorithmus wird. Dies wurde durch sorgfältige Speicherverwaltung erreicht.

Eine weitere Schwerpunkt dieser Arbeit liegt darin, dass die Fehlerdiagnose anschaulich dargestellt wird. Neben Implementierung unseres Spiel-Algorithmus wird ein Tool entwickelt, mit dem sich ein oder mehrere Pfade interaktiv konstruieren lassen, die die Fehlersituation graphisch zeigen.

Die Vorgehensweise dieser Arbeit ist Folgende:

Wir führen zunächst grundlegende mathematische Begriffe sowie Definitionen ein. Dann werden einige temporale Logiken wie CTL, LTLund CTL^*vorgestellt. Danach wird der Begriff von Modell, Syntax sowie Semantik der modalen μ-Kalkül definiert.

Im dritten Kapitel werden die Fixpunktsätze von Knaster-Tarski und Kleene erläutert. Danach wird das Model-Checking-Problem erklärt. Dann wird ein globaler Model-Checking-Algorithmus betrachtet. Wir werden relativ ausführlich die Abfolge der Berechnung dieses Algorithmus illustrieren, so dass wir einen spiel-

theoretischen Ansatz bereits von dieser Berechnungsabfolge ansatzweise erklären können.

Im vierten Kapitel werden die Begriffe und Definitionen über Spiele eingeführt und dann zwei Spiel-Algorithmen vorgestellt. Beim ersten Algorithmus wird sich darauf konzentriert, zu zeigen, dass das Spiel-Problem mit dem Algorithmus gelöst werden kann, der auf einem Model-Checking-Algorithmus basiert. Durch sorgfältige Speicherverwaltung sowie Ausnutzung der Monotonieeigenschaft bei der Fixpunktberechnung schaffen wir zu entscheiden, welche Zwischenergebnisse bei der weiteren Berechnungen beibehalten werden dürfen bzw. weggeworfen werden müssen. Danach werden die Korrektheit und die Laufzeitkomplexität unseres neuen Algorithmus gezeigt.

In fünften Kapitel werden Verbesserungen sowie Optimierungen vorgestellt. Wir betrachten dabei spezielle Fälle mit der Alternierungstiefe 2 und höher. Wir werden die Korrektheit der Verbesserungen zeigen. Unser Algorithmus wird dann mit dem neuesten Ergebnis der Forschung verglichen.

Im sechsten Kapitel wird die Implementierung sowie Benutzung des Spiel-Algorithmen erklärt. Wir werden hierfür die Datenstruktur und den Hauptteil des Programmes zeigen und die Schnittstelle für die Anwender ausführlich erläutern. Unsere Spiel-Algorithmen dienen dazu, die Fehlern beim Systementwurf leicht zu finden. Eine gute Bedienbarkeit ist genauso wichtig wie die Performance des Programmes.

Zum Schluss wird ein Überblick sowie Ausblick über den Spiel-Algorithmus zur Fehlerdiagnose beim Model-Checking gegeben.

Kapitel 2

Temporale Logiken

Die Systeme, die in dieser Arbeit von Interesse sind, sind reaktive Systeme [MaPn95], die kontinuierlich mit der Umgebung interagieren. Sie werden bei unterschiedlichen Anwendungen eingesetzt, wie z.B. eingebettete Systeme und Kommunikationsprotokolle. Viele dieser Anwendungen sind sicherheitskritisch, weil Fehlfunktionen einen erhebliche Schaden verursachen können. Deshalb ist man sehr daran interessiert, die Entwicklung reaktiver Systeme durch Methoden der formalen Spezifikation und Verifikation zu unterstützen. Um Eigenschaften von reaktiven Systemen zu formulieren, die das zeitliche Verhalten des Systems betreffen, sind temporale Logiken gut geeignet, da sie Operatoren wie „immer“ oder „schließlich“ besitzen. Die Sicherheitseigenschaft (gewisse Ereignisse dürfen nie eintreten) oder die Lebendigkeitseigenschaft (irgendwann muss ein bestimmtes Ereignis eintreten) kann somit problemlos ausgedrückt werden. Durch verschachtelte Verwendung temporaler Operatoren können auch komplexere Bedingungen spezifiziert werden, wie „jedesmal, wenn ein Ereignis A erfolgt, tritt irgendwann ein Ereignis B ein“.

In diesem Kapitel werden zunächst allgemeine Notationen festgelegt und dann einige typische temporale Logiken CTL(Computation Tree Logic) [CES86] sowie LTLTemporal Logic) [Pn77] eingeführt. Danach wird der modale μ-Kalkül vorgestellt, dessen Ausdrucksmächtigkeit stärker als CTLund LTList. Wir benutzen die Formeln dieser Logik zur Formulierung der gewünschten Eigenschaften. Neben Modellierung reaktiver Systeme wird die Syntax sowie Semantik des modalen μ-Kalküls definiert. Schließlich werden einige bekannte Sätze über diese Logik erläutert, die wir beim Entwurf unseres Algorithmus anwenden werden.

2.1 Notationen

Es werden die mengentheoretisch üblichen Schreibweisen verwendet. Für die Teilmengenrelation schreiben wir $\subseteq$, für die echte Teilmengenrelation $\subset$ und für die Differenzmenge zweier Mengen A und B schreiben wir $A - B$. Für die Vereinigungsmenge $A \cup B$ zweier Mengen A und B benutzen wir auch die Bezeichnung $A + B$. Falls B eine einelementige Menge (z.B. $B = \{x\}$) ist, schreiben wir auch $A + x$ bzw. $A - x$ für die Vereinigungsmenge $A + \{x\}$ bzw. für die Differenzmenge $A - \{x\}$. Die Menge 2^A ist die Potenzmenge von A. Normalerweise werden die großen Buchstaben $A, B, C, \ldots$ für Mengen und die kleinen Buchstaben $a, b, c, \ldots$ für Elemente verwendet. Die Bezeichnung $A \leq b$ wird für eine Vergleichrelation $\leq$ benutzt, wenn $a \leq b$ für alle $a \in A$ gilt.

Abkürzend werden auch die üblichen Symbole $\exists$ (es existiert ein $\cdots$), $\forall$ (für alle $\cdots$), $\Rightarrow$ (folgt $\cdots$), $\Leftrightarrow$ (genau dann, wenn $\cdots$) benutzt.

2.2 CTL^* und ihre Teillogiken

In diesem Abschnitt werden temporale Logiken eingeführt, mit denen man zeitliche Abfolgen von Ereignissen in einem System spezifizieren kann. Es wird zunächst die temporale Logik CTL^* vorgestellt, indem ihre Syntax und Semantik definiert wird. Anschließend werden einige wichtige Teillogiken von CTL^* definiert.

Die temporale Logik CTL^* enthält neben den booleschen Operatoren $\wedge$ (Konjunktion), $\vee$ (Disjunktion) und $\neg$ (Negation) die temporalen Operatoren ($\mathbf{X}, \mathbf{F}, \mathbf{U}$, usw.) sowie die Pfadquantoren **E** und **A**. Im Folgenden werden einige Operatoren von CTL^* umgangssprachlich erklärt:

- Mit **X** („next time“) wird verlangt, dass eine Eigenschaft im nächsten Zustand erfüllt wird.
- Der Operator **F** („finally“) wird verwendet, um zu beschreiben, dass eine Eigenschaft irgendwann in einem Zustand auf dem Pfad erfüllt wird.
- Der Operator **G** („globally“ oder „generally“) wird für die Aussage verwendet, dass eine Eigenschaft in jedem Zustand auf dem Pfad erfüllt wird.
- Für den Operator **U** („until“) werden zwei Eigenschaften kombiniert benutzt, um zu beschreiben, dass die zweite Eigenschaft irgendwann in einem Zustand

auf dem Pfad erfüllt wird und die erste Eigenschaft in jedem Zustand erfüllt wird, solange die zweite Eigenschaft noch nicht erfüllt ist.

- Der Operator **R** („release“) ist der logisch duale Operator von **U**. Er wird für die Aussage benutzt, dass die zweite Eigenschaft solange erfüllt wird, bis die beiden Eigenschaften erfüllt werden. Es wird aber nicht verlangt, dass die erste Eigenschaft irgendwann erfüllt werden muss. In dem Fall soll die zweite Eigenschaft immer gelten.
- Mit dem Pfadquantor **E** wird die Aussage beschrieben, dass es einen Pfad gibt, für den eine Eigenschaft erfüllt wird.
- Der Pfadquantor **A** spezifiziert die Aussage, dass eine Eigenschaft für alle Pfade erfüllt wird.

Durch Verwendung der temporalen Operatoren können die zeitlichen Abfolgen der bestimmten Ereignisse in einem System leicht ausgedrückt werden. Wir definieren zuächst die Syntax der CTL^*-Formeln.

Definition 2.2.1 (*Syntax von CTL^**)
Sei $AP = \{p, q, \cdots\}$ die Menge der atomaren Propositionen. Die Menge der Zustandsformeln ZF und die Menge der Pfadformeln PF von CTL^* sind folgendermaßen definiert:

- *Ist $p \in AP$, dann ist p eine Zustandsformel.*
- *Sind f und q Zustandsformeln, dann sind $\neg f$, $f \vee g$, und $f \wedge g$ Zustandsformeln.*
- *Ist f eine Pfadformel, dann sind $\mathbf{E}f$ und $\mathbf{A}f$ Zustandsformeln.*
- *Ist f eine Zustandsformel, dann ist f auch eine Pfadformel.*
- *Sind f und g Pfadformeln, dann sind $\neg f, f \vee g, f \wedge g, \mathbf{X}f, \mathbf{F}f, \mathbf{G}f, f\,\mathbf{U}\,g$, und $f\mathbf{R}g$ Pfadformeln.*

Die CTL^*-Formeln beschreiben Eigenschaften der Zustände eines Systems bzw. Abläufe bestimmter Zustandsübergänge in einem System. Zur formalen Verifikation der CTL^*-Formeln wird zunächst eine Abstraktion von einem System zu einer Graphenstruktur benötigt, in der die Zustände und Zustandsübergänge abgebildet werden. Welche Graphenstruktur dafür ausgewählt wird, hängt von dem betrachtenden System ab. Im Folgenden wird die Kripke-Struktur vorgestellt, die oft zur Modellierung von reaktiven Systemen eingesetzt wird.

Definition 2.2.2 (*Kripke-Struktur*)
Eine Kripke-Struktur ist ein Tripel $K = (S, R, I)$*, wobei*

- S *ist eine Menge von Zuständen,*
- $R \subseteq S \times S$ *ist eine Übergangsrelation,*
- $I : S \to 2^{AP}$ *ist eine Markierungsfunktion, die jedem Zustand eine Menge von atomaren Propositionen zuordnet.*

Die Größe einer Kripke-Struktur $K = (S, R, I)$ wird mit $|M|$ bezeichnet, wobei $|M| = |S| + |R|$, d.h. die Größe von M ist die Summe der Anzahl der Zustände und der Anzahl der Zustandsübergänge.

Definition 2.2.3 (*Pfad in Kripke-Struktur*)
Sei $K = (S, R, I)$ *eine Kripke-Struktur. Dann gilt:*

- *Ein Pfad in* K *ist eine Folge von Zuständen* $\pi = s_0, s_1, \ldots$ *mit* $(s_i, s_{i+1}) \in R$ *für alle* $i \geq 0$*.*
- *Wenn* $\pi = s_0, s_1, \ldots$ *ein Pfad in* K *ist, dann sei* π^i *das Suffix von* π*, das mit* s_i *beginnt, also* $s_i, s_{i+1}, \ldots$*, und* π_i *das* i*-te Element im Pfad, also* s_i*. Mit* $|\pi|$ *wird die Länge eines Pfades* π *bezeichnet.*

Bei der Definition der Semantik von CTL^* sind die Fälle zu unterscheiden, ob die Formel f eine Zustandsformel oder eine Pfadformel ist. Die Bedeutung von f wird somit bzgl. Zuständen bzw. Pfaden definiert, d.h. f beschreibt eine Menge von Zuständen einer gegebenen Kripke-Struktur bzw. eine Menge von Pfaden in dieser. Im Folgenden wird dies in einer Relationenschreibweise ausgedrückt. $K, s \models f$ bedeutet, dass die durch die Formel f beschriebenen Eigenschaften im Zustand $s \in S$ der Kripke-Struktur $K = (S, R, I)$ gilt.

Wenn f eine Zustandsformel ist, so bedeutet $K, s \models f$, dass f im Zustand s der Kripke-Struktur K gilt. Im Fall einer Pfadformel f bedeutet $K, \pi \models f$, dass f entlang dem Pfad π in der Kripke-Struktur K gilt. Anstatt $K, s \models f$ und $K, \pi \models f$ wird auch oft $s \models_K f$ bzw. $\pi \models_K f$ geschrieben. Wenn K aus dem Zusammenhang ersichtlich ist, wird es weggelassen.

Definition 2.2.4 (*Semantik von CTL^**)
Seien f_1 und f_2 Zustandsformeln und g_1 und g_2 Pfadformeln. Dann ist $\models$ wie folgt induktiv definiert.

- $s \models p \Leftrightarrow p \in I(s)$
- $s \models \neg f_1 \Leftrightarrow s \not\models f_1$
- $s \models f_1 \vee f_2 \Leftrightarrow s \models f_1$ *oder* $s \models f_2$
- $s \models f_1 \wedge f_2 \Leftrightarrow s \models f_1$ *und* $s \models f_2$
- $s \models \mathbf{E}g_1 \Leftrightarrow$ *es gibt einen Pfad* π*, so dass* $\pi \models g_1$
- $s \models \mathbf{A}g_1 \Leftrightarrow$ *für alle Pfade* π*, die mit dem Zustand* s *beginnen, gilt* $\pi \models g_1$
- $\pi \models f_1 \Leftrightarrow s \models f_1$ *mit* $s = \pi^0$
- $\pi \models \neg g_1 \Leftrightarrow \pi \not\models g_1$
- $\pi \models g_1 \vee g_2 \Leftrightarrow \pi \models g_1$ *oder* $\pi \models g_2$
- $\pi \models g_1 \wedge g_2 \Leftrightarrow \pi \models g_1$ *und* $\pi \models g_2$
- $\pi \models \mathbf{X}g_1 \Leftrightarrow \pi^1 \models g_1$
- $\pi \models \mathbf{F}g_1 \Leftrightarrow$ *es gibt ein* $k \geq 0$*, so dass* $\pi^k \models g_1$
- $\pi \models \mathbf{G}g_1 \Leftrightarrow \pi^k \models g_1$ *für alle* $k \geq 0$
- $\pi \models g_1 \,\mathbf{U}\, g_2 \Leftrightarrow \exists k \geq 0$ *mit* $\pi^k \models g_2$ *und* $\forall j < k : \pi^j \models g_1$
- $\pi \models g_1 \,\mathbf{R}\, g_2 \Leftrightarrow \forall k \geq 0 : \pi^k \models g_2$ *oder* $\exists j < k$ *mit* $\pi^j \models g_1$

Aus der Semantik können die folgenden Rekursionsgleichungen hergeleitet werden, die als Beweisregeln verwendet werden können.

- $\pi \models g_1 \,\mathbf{U}\, g_2 \Leftrightarrow (\pi \models g_2)$ oder $(\pi \models g_1$ und $\pi \models \mathbf{X}(g_1 \,\mathbf{U}\, g_2))$
- $\pi \models g_1 \,\mathbf{R}\, g_2 \Leftrightarrow (\pi \models g_2)$ und $(\pi \models g_1$ oder $\pi \models \mathbf{X}(g_1 \,\mathbf{R}\, g_2))$

Die Formel $g_1 \,\mathbf{U}\, g_2$ besagt, dass g_1 solange gilt, bis g_2 erfüllt ist. Die Eigenschaft der Teilformel g_2 gilt also irgendwann nach endlichen Schritten. Die Formel $g_1 \,\mathbf{R}\, g_2$ bedeutet, dass g_2 immer gilt, bis g_1 und g_2 gleichzeitig erfüllt werden. Dabei ist es nicht notwendig, dass g_1 irgendwann erfüllt wird. Es genügt auch, wenn die Eigenschaft von g_2 immer gilt.

Die Bedeutung des **R**-Operators sieht so aus, dass keine Anwendung dafür leicht zu finden ist. Der eigentliche Zweck der Einfügung dieses Operators ist, dass die

Umformung der Eingabeformel in eine positive Form erleichtert werden soll und dadurch der Entwurf eines Algorithmus sowie die Abschätzung der Laufzeitkomplexität vereinfacht wird. Im Folgenden werden die Begriffe eingeführt, um zu erklären, in welcher Form mit welchem Aufwand die Eingabeformel umgewandelt wird.

Definition 2.2.5 (*Formellänge*)
Mit $|f|$ wird die Länge einer Formel f bezeichnet. Sie ist die Anzahl der Knoten im Syntaxbaum der Formel f, also

- $|p| = 1$ *für jedes* $p \in AP$
- $|\neg f| = |f| + 1$
- $|f \vee g| = |f| + |g| + 1$
- $|f \wedge g| = |f| + |g| + 1$
- $|\mathbf{X}f| = |f| + 1$
- $|\mathbf{F}f| = |f| + 1$
- $|\mathbf{G}f| = |f| + 1$
- $|\mathbf{A}f| = |f| + 1$
- $|\mathbf{E}f| = |f| + 1$
- $|f\,\mathbf{U}\,g| = |f| + |g| + 1$
- $|f\,\mathbf{R}\,g| = |f| + |g| + 1$

Die Formellänge ist ein leicht auswählbares Maß, mit dem man die Kompliziertheit der Formel abschätzen kann. Bei vielen Algorithmen über temporale Logiken ist die Formellänge ein wichtiger Faktor zur Berechnung des Laufzeitaufwandes. Unabhängig davon, was für ein Algorithmus benutzt wird, ist die Berechnung mit einer kürzeren Formel normalerweise einfacher als die Berechnung mit einer längeren Formel, wenn die Struktur der Formel gleich bleibt. Wie man in Definition der Semantik von CTL^* sieht, haben die Operatoren $\vee$ und $\wedge$, $\mathbf{A}$ und $\mathbf{E}$ sowie $\mathbf{U}$ und $\mathbf{R}$ eine duale Bedeutung und man braucht daher einen gleichen oder ähnlichen Laufzeitaufwand zur Behandlung solcher Operatoren in Algorithmen. Bei der Negation ist es aber nicht der Fall. Die Menge der Zustände bzw. der Pfade, die die Formel erfüllen, sogenannte Lösungsmenge wird durch Differenzmenge berechnet, so dass der Aufwand der Berechnung der Negation von dem Operand abhängt. Je nachdem, ob

die Lösungsmenge von dem Operand eine Zustandsmenge oder eine Pfadmenge ist, erhält man verschiedene Lösungsmengen für die Negation. Wenn die Negation außerdem kein äußerster Operator der Formel ist, wird die durch Negation berechnete Lösungsmenge mit den höheren Operatoren weiter verarbeitet. Da für die Bildung einer Differenzmenge in der Regel bereits eine exponentielle Laufzeit benötigt wird, erschwert die Negation die Implementierung eines effizienten Algorithmus für die temporalen Logiken. Die Eingabeformeln werden deshalb oft per Preprocessing so transformiert, dass die Negation nur auf der untersten Ebene auftritt, da die Negation auf dem atomaren Proposition einfach zu behandeln ist.

Definition 2.2.6 (*Positive Normalform in CTL^**)
Eine CTL^*-Formel ϕ ist in positiver Normalform, falls die Negation $\neg$ in ϕ nur unmittelbar vor atomaren Propositionen auftritt.

Wir verwenden die folgenden Abkürzungen: $\mathsf{tt} := p \vee \neg p$ für ein beliebiges $p \in AP$ und $\mathsf{ff} := \neg\mathsf{tt}$. Dann ist leicht zu erkennen, dass die Formel $(\mathsf{tt}\,\mathbf{U}\,\phi)$ semantisch äquivalent zu $\mathbf{F}\phi$ ist. Analog ist die Formel $(\mathsf{ff}\,\mathbf{R}\,\phi)$ semantisch äquivalent zu $\mathbf{G}\phi$, da die Formel ff nie erfüllt wird. Die Formel $(\mathsf{ff}\,\mathbf{R}\,\phi)$ wird nur dann erfüllt, wenn die Teilformel ϕ stets erfüllt wird. Man kann die Formeln so umformen, dass in den Formeln nur die Operatoren $\wedge, \vee, \mathbf{A}, \mathbf{E}, \mathbf{U}, \mathbf{R}$ und $\mathbf{X}$ auftreten. Alle $\mathbf{F}$- bzw. $\mathbf{G}$-Operatoren werden nämlich mit dem Operator $\mathbf{U}$ bzw. $\mathbf{R}$ dementsprechend ausgetauscht.

Um die gegebene Formel in positive Normalform umzuwandeln, wird dann die Dualität der Operatoren von $\wedge$ und $\vee$, $\mathbf{A}$ und $\mathbf{E}$, sowie $\mathbf{U}$ und $\mathbf{R}$ angewandt. Die Negation, die nicht direkt vor einer atomaren Proposition vorkommt, kann mit Hilfe der folgenden Regeln auf unterer Ebene geschoben werden.

- $\neg\neg f \equiv f$
- $\neg(f \wedge g) \equiv \neg f \vee \neg g$
- $\neg(f \vee g) \equiv \neg f \wedge \neg g$
- $\neg(f\,\mathbf{U}\,g) \equiv \neg f\,\mathbf{R}\,\neg g$
- $\neg(f\,\mathbf{R}\,g) \equiv \neg f\,\mathbf{U}\,\neg g$
- $\neg\mathbf{A}f \equiv \mathbf{E}(\neg f)$
- $\neg\mathbf{E}f \equiv \mathbf{A}(\neg f)$.

Es ist zu beachten, dass die so resultierende Formel nicht länger als $2 \cdot |f|$ wird, wobei f die ursprüngliche Eingabeformel ist. Der Faktor 2 ist dafür, dass jeder Operator maximal zwei Teilformeln als Operand hat. Nach Anwendung obiger Regeln werden also maximal doppelt so viel Negationen wie die Anzahl der Operatoren in der Formel entstehen. Da die zweifachen Negationen, die während der Umformung der Formel entstehen, mit der Regel $\neg\neg f = f$ weggelassen werden, beträgt die Formellänge zum Schluss insgesamt kürzer oder gleich lang wie $2 \cdot |f|$. Die Laufzeit solcher Umformung ist leicht zu berechnen, und zwar beträgt $\mathcal{O}(|f|)$, da jeder Operator genau einmal betrachtet wird.

Die Formel $\neg(f \,\mathbf{U}\, g)$ kann zwar auch ohne Verwendung des **R**-Operators in Normalform umgewandelt werden, aber die Formellänge verlängert sich dann um mehr als den Faktor 2. Die Formeln mit dem **F**- bzw. **G**-Operator lassen sich wie erwähnt leicht zu einer Formel mit dem **U**- bzw. **R**-Operator umwandeln, weshalb sie oft bei der Definition für die Syntax sowie für die Semantik von CTL^* weggelassen werden.

In dem nächsten Abschnitt werden wir noch den modalen μ-Kalkül einführen, der eine ausdrucksstärkere Logik als CTL^* ist. Die Eingabeformel wird in diesem Arbeit nach dem modalen μ-Kalkül erstellt. Eine Teillogik des modalen μ-Kalküls in der Art von CTL^* wird dennoch als Eingabe erlaubt, da die bestimmten Formeln von CTL^* direkt zu einer μ-Kalkül-Formel umgewandelt werden können.

Definition 2.2.7 ($K \models f$)
Eine Kripke-Struktur K erfüllt eine Formel f (kurz $K \models f$), wenn $\pi \models f$ für jeden Pfad π in K gilt.

Es ist zu bemerken, dass $K \not\models f$ nicht $K \models \neg f$ impliziert, da die Erfüllbarkeit für jeden Pfad gelten muss. Nun betrachten wir zwei Teillogiken von CTL^*, die sehr oft benutzt werden. Die Bedeutung der Operatoren bei den Teillogiken bleibt genauso wie bei CTL^* unverändert und daher verwenden wir die Bezeichnung von CTL^* auch unverändert weiter.

Definition 2.2.8 (*CTL: Computation Tree Logic*)
Die Menge der CTL-Formeln ist wie folgt induktiv definiert:

- *Ist $p \in AP$, dann ist p eine CTL-Formel.*
- *Sind f und g CTL-Formeln, so sind $\neg f, f \wedge g, f \vee g, \mathbf{EX}f, \mathbf{AX}f, \mathbf{EF}f$, $\mathbf{AF}f$, $\mathbf{EG}f$, $\mathbf{AG}f, \mathbf{E}(f \,\mathbf{U}\, g), \mathbf{A}(f \,\mathbf{U}\, g), \mathbf{E}(f \,\mathbf{R}\, g)$ und $\mathbf{A}(f \,\mathbf{R}\, g)$ CTL-Formeln.*

CTL ist eine Teillogik von CTL^*, in der jeder temporale Operator $\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}$ und $\mathbf{R}$ unmittelbar auf einen Pfadquantor $\mathbf{E}$ oder $\mathbf{A}$ folgen muss. Für eine CTL-Formel f kann also $K, s \models f$ geprüft werden, ob f im Zustand s der Kripke-Struktur K gilt.

Bei der Definition von CTL kann man $\mathbf{EF}f, \mathbf{AF}f, \mathbf{EG}f, \mathbf{AG}f, \mathbf{E}(f\,\mathbf{R}\,g)$ und $\mathbf{A}(f\,\mathbf{R}\,g)$ auslassen, da sie mit einer anderen Formel wie z.B. $\mathbf{E}(\mathtt{tt}\,\mathbf{U}\,f) \equiv \mathbf{EF}f$, usw. ausgedrückt werden können.

Eine andere Teillogik von CTL^*, die auch in vielen Anwendungsgebieten eingesetzt wird, ist LTL (Linear Temporal Logic), die wie folgt definiert ist.

Definition 2.2.9 (*LTL: Linear Temporal Logic*)
Die Menge der LTL-Formeln besteht aus Formeln der Form $\mathbf{A}f$*, wobei* f *eine Pfadformel ist, die durch die folgenden Regeln definiert ist:*

- *Ist* $p \in AP$*, dann ist* p *eine Pfadformel.*
- *Sind* f *und* g *Pfadformeln, so sind* $\neg f, f \wedge g, f \vee g, \mathbf{X}f, \mathbf{F}f, \mathbf{G}f, f\,\mathbf{U}\,g$ *und* $f\,\mathbf{R}\,g$ *auch Pfadformeln.*

LTList eine Pfadlogik, in der die Pfadeigenschaften ausgedrückt werden. Für eine LTL-Formel f kann $K, \pi \models f$ geprüft werden, ob der Pfad π der Kripke-Struktur K die Formel f erfüllt. Für ein Zustand s gilt $K, s \models f$ genau dann, wenn $K, \pi \models f$ für alle Pfade π mit $\pi_0 = s$. Die Pfadquantifizierung $\mathbf{A}$ für die LTL-Formeln ist eigentlich eine Konvention, so dass $K, s \models f$ geprüft werden kann. Dadurch ist es auch möglich, LTLbzgl. ihrer Ausdrucksstärke mit CTLzu vergleichen. Man kann auch genauso eine existenzielle Pfadquantifizierung $\mathbf{E}$ als Konvention einführen. Zur Selektierung eines Pfades, der gegebene LTL-Formeln erfüllt, ist die alternative Konvention gut geeignet. Eine solche Anwendung haben wir implementiert und in das METAFrame Projekt integriert [BMSY98].

Es ist offensichtlich, dass die beiden Logiken CTL und LTL in CTL^* enthalten sind. Von Emerson und Halpern [EmHa86] wurde gezeigt, dass CTL und LTL unvergleichbar sind. Die Formel $\mathbf{A}(\mathbf{FG}p)$ ist z.B. eine LTL-Formel, für die es keine äquivalente CTL-Formel gibt. Umgangsprachlich bedeutet diese Formel, dass in allen Pfaden ein Zustand existiert, von dem aus p immer gilt. Andererseits gibt es keine äquivalente LTL-Formel für die CTL-Formel $\mathbf{AG}(\mathbf{EF}p)$. Für die CTL^*-Formel $\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$, die die Disjunktion der beiden Formeln ist, gibt es schließlich keine äquivalente Formel weder in CTLnoch in LTL.

In diesem Abschnitt wurden einige temporale Logiken betrachtet, die in vielen Bereichen angewendet werden. Ein Vorteil dieser Logiken ist, dass die Operatoren leicht erfassbare Bedeutung haben: Die Formeln können umgangssprachlich erklärt werden und umgekehrt die gewünschten Eigenschaften können leicht in eine Formel umgesetzt werden.

Eine andere temporale Logik, die wir in dem nächsten Abschnitt behandeln wollen, basiert auf der sogenannten Fixpunktlogik und umfasst alle temporale Logiken, die wir in diesem Abschnitt vorgestellt haben. Ihre Ausdrucksstärke ist sogar echt größer als CTL^*.

2.3 Der modale μ-Kalkül

In diesem Abschnitt wird der modale μ-Kalkül vorgestellt, der wegen seiner hohen Ausdrucksstärke in vielen Anwendungsgebieten eingesetzt wird. Seine Sprache umfasst z.B. die Sprache von CTL^*. Zunächst werden reaktive Systeme mit Hilfe von Kripke-Transitionssystemen modelliert. Dann wird die Syntax sowie Semantik des modalen μ-Kalküls definiert. Der modale μ-Kalkül dient in dieser Arbeit zur Spezifikation der Eigenschaften von reaktiven Systemen. Danach werden strukturelle Maße der μ-Kalkül-Formeln eingeführt, mit denen die Kompliziertheit einer Formel gemessen werden kann. Sie spielen eine große Rolle bei der Komplexität des Model-Checkings, das in dem nächsten Kapitel vorgestellt wird.

Definition 2.3.1 (*Modell*)
Ein Modell ist ein Kripke-Transitionssystem $M = (S, Act, AP, \rightarrow, L)$*, wobei*

- S *eine endliche Menge von Zuständen,*
- Act *eine endliche Menge von Aktionen,*
- AP *eine endliche Menge von atomaren Propositionen,*
- $\rightarrow \subseteq S \times Act \times S$ *eine Transitionsrelation und*
- $L : S \rightarrow 2^{AP}$ *eine Markierungsfunktion, die jedem Zustand eine Menge von atomaren Propositionen zuordnet.*

Der Einfachheit halber benutzen wir auch die übliche Schreibweise $s \xrightarrow{a} s'$ für $(s, a, s') \in \rightarrow$. Die Größe eines Modells $M = (S, Act, AP, \rightarrow, L)$ wird mit $|M|$ be-

zeichnet, wobei $|M| =_{df} |S| + |\rightarrow|$, d.h. die Größe von M ist die Summe der Anzahl der Zustände und der Anzahl der Elemente der Transitionsrelation.

Um reaktive Systeme zu modellieren, braucht man nicht unbedingt die beiden, die Transitionsrelation und die Markierungsfunktion. Normalerweise reicht es aus, eine von den beiden zu haben. Welche von den beiden man braucht, hängt davon ab, wie das betrachtende reaktive System aufgebaut wird und wie sich verhält. Wir stellen jedoch die beiden zur Verfügung, so dass die reaktiven Systeme möglichst flexibel modelliert werden können.

Im Folgenden wird die Syntax des modalen μ-Kalküls definiert.

Definition 2.3.2 (*Syntax des μ-Kalküls*)
Sei AP die Menge der atomaren Propositionen, Act die Menge der Aktionen, und Var eine Menge von Variablen. Dann ist die Menge der μ-Kalkül-Formeln wie folgt induktiv definiert:

- *Ist $p \in AP$, dann ist p eine μ-Kalkül-Formel.*
- *Ist $X \in Var$, dann ist X eine μ-Kalkül-Formel.*
- *Sind f und g μ-Kalkül-Formeln, dann sind $\neg f$, $f \vee g$ und $f \wedge g$ auch μ-Kalkül-Formeln.*
- *Ist a eine Aktion und f eine μ-Kalkül-Formel, dann sind $\langle a \rangle f$ und $[a]f$ μ-Kalkül-Formeln.*
- *Ist X eine Variable und f eine μ-Kalkül-Formel, dann sind $\mu X.f$ und $\nu X.f$ auch μ-Kalkül-Formeln.*

μ bzw. ν sind der größte bzw. kleinste Fixpunktoperator, während $\langle a \rangle$ und $[a]$ die modalen Operatoren sind. $\langle a \rangle \Phi$ bedeutet umgangssprachlich: „Es gibt einen Nachfolger, der durch Ausführen der Aktion a erreicht wird und für den Φ gilt.“ $[a]\Phi$ bedeutet hingegen: „Für alle Nachfolger, die durch Ausführen der Aktion a erreicht werden, gilt die Formel Φ.“

Es ist zu beachten, dass die Formel $[a]\Phi$ auch gilt, falls die Aktion a überhaupt nicht ausführbar ist. Mit $\langle \cdot \rangle \Phi$ bzw. $[\cdot]\Phi$ bezeichnen wir außerdem $\bigvee_{a \in Act} \langle a \rangle \Phi$ bzw. $\bigwedge_{a \in Act} [a]\Phi$.

Die Semantik $[\![\cdot]\!] : \Phi \to 2^S$ des μ-Kalküls wird bzgl. eines Modells $M = (S, Act, AP, \to, L)$ und einer Umgebung $e : Var \to 2^S$ definiert, wobei die Funktion e eine Belegung der freien Variablen angibt. Wir schreiben $e[S'/X]$ für die Umgebung, die definiert ist durch

$$e[S'/X](Y) = \begin{cases} S' & \text{falls } Y \equiv X \\ e(Y) & \text{sonst} \end{cases}$$

Definition 2.3.3 (*Semantik des μ-Kalküls*)
Die Menge der Zustände, die eine μ-Kalkül-Formel Φ in einer Umgebung e erfüllen, ist wie folgt induktiv definiert:

- $[\![P]\!]_e^M := \{s \in S \mid P \in L(s)\}$
- $[\![X]\!]_e^M := e(X)$
- $[\![\neg\Phi]\!]_e^M := S \backslash [\![\Phi]\!]_e^M$
- $[\![\Phi \vee \Psi]\!]_e^M := [\![\Phi]\!]_e^M \cup [\![\Psi]\!]_e^M$
- $[\![\Phi \wedge \Psi]\!]_e^M := [\![\Phi]\!]_e^M \cap [\![\Psi]\!]_e^M$
- $[\![\langle a \rangle \Phi]\!]_e^M := \{s \mid \exists t : s \xrightarrow{a} t \wedge t \in [\![\Phi]\!]_e^M\}$
- $[\![[a] \Phi]\!]_e^M := \{s \mid \forall t : s \xrightarrow{a} t \Rightarrow t \in [\![\Phi]\!]_e^M\}$
- $[\![\nu X.\Phi]\!]_e^M := \bigcup\{S' \subseteq S \mid [\![\Phi]\!]_{e[S'/X]}^M \supseteq S'\}$
- $[\![\mu X.\Phi]\!]_e^M := \bigcap\{S' \subseteq S \mid [\![\Phi]\!]_{e[S'/X]}^M \subseteq S'\}$

Wir schreiben auch $M, s \models_e \Phi$, falls $s \in [\![\Phi]\!]_e^M$. Ist die Semantik einer Formel Φ nicht von der Umgebung e abhängt, schreiben wir auch $M, s \models \Phi$ bzw. $s \in [\![\Phi]\!]^M$. Wenn das Modell M aus dem Zusammenhang ersichtlich ist, wird es weggelassen.

Im Folgenden werden einige Begriffe über die Formeln des modalen μ-Kalküls eingeführt, die zur Einschränkung der Eingabeformel verwendet werden. In dieser Arbeit werden wir die Formeln nur in einer bestimmten Form als Eingabeformel annehmen, so dass die wichtige Eigenschaft der Semantikfunktion die Monotonie sowie die Stetigkeit bzgl. $\subseteq$ auf der Zustandsmenge erhalten bleibt.

Definition 2.3.4 (*Wohlbenannte Formel*)
Eine Formel Φ heißt wohlbenannt, wenn jede Variable X in ihr höchstens einmal mit μX bzw. νX quantifiziert wird.

Definition 2.3.5 (*Wohlgeformte Formel*)
Eine Formel Φ heißt wohlgeformt, wenn alle freie Variablenvorkommen im Rumpf einer Fixpunktformel im Bereich einer geraden Anzahl von Negationen liegen.

Definition 2.3.6 (*Geschlossene Formel*)
Eine Formel Φ heißt geschlossen, wenn alle in der Formel auftretenden Variablen jeweils durch den Fixpunktoperator ν bzw. μ gebunden werden.

Definition 2.3.7 (*Positive Normalform*)
Eine Formel Φ ist in positiver Normalform, wenn die Negation nur vor atomaren Propositionen oder Aktionen auftreten.

Wir nehmen in dieser Arbeit an, dass die Anzahl der Propositionen bzw. der Aktionen endlich und jede Eingabeformel wohlbenannt, wohlgeformt sowie geschlossen ist. Durch Umbenennung der gebundenen Variablen kann man jede Formel in eine äquivalente und wohlbenannte Formel transformieren. Es ist also keine Einschränkung. Ist eine Formel Φ nicht wohlgeformt, dann gilt eine der wichtigsten Eigenschaften des modalen μ-Kalküls nicht, und zwar ist die Semantikfunktion $[\![\,\Phi]\!]$ nicht monoton bzgl. $\subseteq$ auf der Zustandsmenge. Es ist deshalb eine sinnvolle Einschränkung.

Wir setzen außerdem voraus, dass die Formeln unmittelbar nach der Eingabe in positive Normalform umgewandelt werden. Das folgende Lemma zeigt, dass jede wohlgeformte und geschlossene Formel zu einer semantisch äquivalenten Formel in positiver Normalform mit einer linearen Laufzeitkomplexität auf der Formellänge transformiert werden kann. Mit $\phi_{[X \to X']}$ bezeichnen wir dabei die Formel, die man aus ϕ durch syntaktische Substitution von X durch X' erhält.

Lemma 2.3.8
Jede wohlgeformte und geschlossene Formel Φ des modalen μ-Kalküls ist semantisch äquivalent zu einem Φ' in positiver Normalform.

Beweis: Die Negation, die nicht vor Propositionen auftreten, können mithilfe der folgenden Äquivalenzen nach innen geschoben werden.

$$
\begin{array}{lcl}
\neg\neg\phi & \equiv & \phi \\
\neg(\phi \vee \psi) & \equiv & \neg\phi \wedge \neg\psi \\
\neg(\phi \wedge \psi) & \equiv & \neg\phi \vee \neg\psi \\
\neg\langle a\rangle\phi & \equiv & [a]\neg\phi \\
\neg[a]\phi & \equiv & \langle a\rangle\neg\phi \\
\neg\nu X.\phi & \equiv & \mu X.\neg\phi_{[X\to\neg X]} \\
\neg\mu X.\phi & \equiv & \nu X.\neg\phi_{[X\to\neg X]}
\end{array}
$$

Alle Äquivalenzen außer den Dualität der Fixpunktoperatoren sind leicht einzusehen. Wir zeigen, dass $[\![\neg\mu X.\Phi]\!]^M_e = [\![\nu X.\neg\Phi_{[X\to\neg X]}]\!]^M_e$ für jedes Modell M und jede Umgebung e. Es sei S die Menge der Zustände von M. Dann gelten

$$
\begin{array}{rl}
[\![\neg\mu X.\Phi]\!]^M_e & = S\backslash\bigcap\{T \mid [\![\Phi]\!]^M_{e[T/X]} \subseteq T\} \\
& = \bigcup\{S\backslash T \mid [\![\Phi]\!]^M_{e[T/X]} \subseteq T\} \\
& = \bigcup\{S\backslash T \mid S\backslash[\![\Phi]\!]^M_{e[T/X]} \supseteq S\backslash T\} \\
& = \bigcup\{S\backslash T \mid S\backslash T \subseteq S\backslash[\![\Phi]\!]^M_{e[T/X]}\} \\
& = \bigcup\{S\backslash T \mid S\backslash T \subseteq S\backslash[\![\Phi_{[X\to\neg X]}]\!]^M_{e[(S\backslash T)/X]}\} \\
& = \bigcup\{S\backslash T \mid S\backslash T \subseteq [\![\neg\Phi_{[X\to\neg X]}]\!]^M_{e[(S\backslash T)/X]}\} \\
& = \bigcup\{T' \mid T' \subseteq [\![\neg\Phi_{[X\to\neg X]}]\!]^M_{e[T'/X]}\} \\
& = [\![\nu X.\neg\Phi_{[X\to\neg X]}]\!]^M_e
\end{array}
$$

Da das Hineinschieben eines außen vorkommenden Negationssymbols mit den Äquivalenzen für Fixpunkte im Rumpf der Fixpunktformel immer zwei neue Negationssymbole erzeugt, bleibt die Formel weiter wohlgeformt sowie geschlossen und deshalb werden die Negationssymbole vor Variablen sukzessiv restlos aufgehoben. □

Es sei $\sigma \in \{\mu, \nu\}$ und $\overline{\sigma} := \begin{cases} \nu & \text{falls } \sigma = \mu \\ \mu & \text{falls } \sigma = \nu \end{cases}$

Die Bezeichnung σ wird verwendet, wenn wir Eigenschaften der Fixpunktoperatoren erklären, ohne einen Fixpunktoperator festzuhalten. Im Folgenden werden nun einige Begriffe eingeführt, die zur Analyse der Struktur der μ-Kalkül-Formeln angewandt werden können.

Definition 2.3.9 (*Schachtelungstiefe*)
Die Schachtelungstiefe $st(\Phi)$ der Fixpunktoperatoren in Φ wird wie folgt induktiv definiert:

- $st(P) = st(X) = st(\neg P) = st(\neg X) = 0$ *für jede atomare Proposition P und Variable X.*

- $st(\Phi \wedge \Psi) = st(\Phi \vee \Psi) = max\{st(\Phi), st(\Psi)\}$.
- $st(\langle a \rangle \Phi) = st([a]\Phi) = st(\Phi)$ *für jede Aktion* a.
- $st(\sigma X.\Phi) = st(\Phi) + 1$.

Die Definition besagt, wie oft die Fixpunktoperatoren verschachtelt vorkommen. Es wird nicht beachtet, ob verschiedene Fixpunktoperatoren abwechselnd auftreten, oder nicht. Die Formel $\nu X.(\nu Y.(A \vee Y) \wedge X)$ hat beispielsweise die Schachtelungstiefe 2. Die Schachtelungstiefe ist ein einfaches Maß zu ermitteln, wie kompliziert die Struktur einer Formel ist.

Aufgrund des Unterschieds der semantischen Bedeutung von μ- und ν-Operator ist es jedoch sinnvoll zu unterscheiden, ob verschiedene Fixpunktoperatoren alternierend vorkommen. Im Folgenden werden die Begriffe eingeführt, so dass wir das alternierende Vorkommen der Fixpunktoperatoren behandeln können.

Definition 2.3.10 (σ*-Teilformel*)

- *Eine* σ*-Teilformel ist eine Teilformel mit Hauptkonnektiv* σ*, d.h. der äußerste Operator der Teilformel ist ein Fixpunktoperator.*
- *Eine* σ*-Teilformel* ϕ *von* Φ *ist genau dann maximal, wenn* ϕ *keine* σ*-Teilformel von irgendeiner anderen* σ*-Teilformel von* Φ *ist.*

Definition 2.3.11 (*Alternierungstiefe* [*EmLe86*])
Sei Φ *eine Formel in positiver Normalform. Die Alternierungstiefe* $at(\Phi)$ *ist induktiv folgendermaßen definiert.*

- *Wenn* Φ *maximale geschlossene* σ*-Teilformeln* $\phi_1, \ldots, \phi_n$ *enthält, dann*

$$at(\Phi) = max\{at(\phi_1), \ldots, at(\phi_n), at(\phi')\},$$

  wobei ϕ' *durch Substitution von* $\phi_1, \ldots, \phi_n$ *durch neue atomare Propositionen* $p_1, \ldots, p_n$ *entsteht.*
- *Im anderen Fall wird die Alternierungstiefe folgendermaßen definiert:*
  - $at(P) = at(X) = at(\neg P) = 0$ *für jede atomare Proposition* P *und Variable* X.

- $at(\Phi \wedge \Psi) = at(\Phi \vee \Psi) = max\{at(\Phi), at(\Psi)\}$.
- $at(\langle\cdot\rangle\Phi) = at([\cdot]\Phi) = at(\Phi)$.
- $at(\sigma X.\Phi) = max\{1, at(\Phi), 1 + at(\overline{\sigma}X_1.\phi_1), \ldots, 1 + at(\overline{\sigma}X_n.\phi_n)\}$, *wobei* $\overline{\sigma}X_1.\phi_1, \ldots, \overline{\sigma}X_n.\phi_n$ *die maximale* $\overline{\sigma}$*-Teilformeln von* Φ *sind.*

Die Alternierungstiefe $at(\Phi)$ ist ähnlich wie die Schachtelungstiefe $st(\Phi)$ definiert, mit dem Unterschied, dass hierbei nur die echte Alternierung von größten und kleinsten Fixpunkten in einer Kette gezählt werden. Die Formel $\nu X.(\nu Y.(A \vee Y) \wedge X)$ hat beispielsweise die Alternierungstiefe 1. Außerdem hat die Formel $\nu X.(\mu Y.(A \vee Y) \wedge X)$ auch die Alternierungstiefe 1, da die Teilformel $\mu Y.(A \vee Y)$ geschlossen ist und die erste Regel von Definition 2.3.11 angewandt wird. Solche Schachtelung wird gutartig genannt.

Die Alternierungstiefe hat eine große Bedeutung, da die Formeln unterschiedlicher Alternierungstiefe in unterschiedlichen Komplexitätsklassen liegen. Dies wurde von Bradfield [Brad96] und Lenzi [Len96] unabhängig voneinander gezeigt, und zwar dass die Alternierungshierarchie strikt ist. Später wurde ein einfacherer Beweis fur die Existenz einer Alternierungshierarchie von Arnold [Arn99] nachgelegt.

Ein anderes Maß zur Abschätzung der Kompliziertheit von μ-Kalkül-Formeln ist die reduzierte Alternierungstiefe, die von Cleaveland, Klein und Steffen [CKS92] vorgestellt wurde.

Definition 2.3.12 *Sei* Φ *eine Formel in positiver Normalform. Eine Formel* Φ' *ist verhältnismäßig geschlossen (eng. relatively closed) in Bezug auf* Φ*, wenn .* $FV(\Phi') \subseteq FV(\Phi)$ *gilt, wobei* $FV(\Psi)$ *die Menge der freien Variablen in* Ψ *ist.*

Die Formel $\mu Y.(Y \vee Z)$ ist z.B. nicht geschlossen, jedoch verhältnismäßig geschlossen im Bezug auf $\nu X.(X \wedge \mu Y.(Y \vee Z))$, da die beiden Formeln dieselbe Menge $\{Z\}$ der freien Variablen haben.

Definition 2.3.13 (*Fixpunkt Normalform*) *Sei* Φ *eine Formel in positiver Normalform.* Φ *ist in Fixpunkt Normalform, wenn für jede Teilformel* $\sigma X.\phi$ *von* Φ *die Variable* X *frei in* ϕ *auftritt.*

Die Teilformel $\sigma X.\phi$ kann durch ϕ ersetzt werden, wenn die Variable X nicht in ϕ frei vorkommt.

Definition 2.3.14 (*Reduzierte Alternierungstiefe* [*CKS92*])
Sei Φ eine Formel in Fixpunkt Normalform. Die reduzierte Alternierungstiefe $rt(\Phi)$ ist induktiv folgendermaßen definiert.

- *Wenn Φ maximale verhältnismäßig geschlossene σ-Teilformeln $\phi_1, \ldots, \phi_n$ enthält, dann*

$$rt(\Phi) = max\{rt(\phi_1), \ldots, rt(\phi_n), rt(\phi')\},$$

  wobei ϕ' durch Substitution von $\phi_1, \ldots, \phi_n$ durch neue atomare Propositionen $p_1, \ldots, p_n$ entsteht.

- *Im anderen Fall wird die reduzierte Alternierungstiefe folgendermaßen definiert:*
  - $rt(P) = rt(X) = rt(\neg p) = 0$ *für jede atomare Proposition P und Variable X.*
  - $rt(\Phi \wedge \Psi) = rt(\Phi \vee \Psi) = max\{rt(\Phi), rt(\Psi)\}$.
  - $rt(\langle a \rangle \Phi) = rt([a]\Phi) = rt(\Phi)$ *für jede Aktion a.*
  - $rt(\sigma X.\Phi) = max\{1, rt(\Phi), 1 + rt(\overline{\sigma} X_1.\phi_1), \ldots, 1 + rt(\overline{\sigma} X_n.\phi_n)\}$, *wobei $\overline{\sigma} X_1.\phi_1, \ldots, \overline{\sigma} X_n.\phi_n$ die maximale $\overline{\sigma}$-Teilformeln von Φ sind.*

Die reduzierte Alternierungstiefe $rt(\Phi)$ ist ähnlich wie die Alternierungstiefe $at(\Phi)$ definiert, wobei die Struktur der Formeln noch feiner betrachtet wird. Wenn eine Formel redundante Fixpunktvariable enthält oder verhältnismäßig geschlossene σ-Teilformel besitzt, wird dies bei Bestimmung der reduzierten Alternierungstiefe der Formel berücksichtigt.

Lemma 2.3.15
Für jede μ-Kalkül-Formel Φ gilt: $rt(\Phi) \leq at(\Phi) \leq st(\Phi)$.

Der modale μ-Kalkül ist ausdrucksstärker als CTL^*, so dass die Sprache von CTL^*in der Sprache des modalen μ-Kalküls mit der Alternierungstiefe 2 echt enthalten ist. Eine explizite Übersetzung von CTL^*-Formeln in den modalen μ-Kalkül wurde erst von Dam [Dam94] gezeigt, wobei sie eine doppel-exponentielle Laufzeitkomplexität auf der Formellänge benötigt. Es wurde dann von Bhat und Cleaveland [BhCle96] gezeigt, dass solche Übersetzung in einer exponentiellen Laufzeitkomplexität möglich ist.

In diesem Abschnitt wurde die Syntax sowie Semantik der modalen μ-Kalkül definiert. Die Begriffe wie Schachtelungstiefe, Alternierungstiefe und reduzierte Alternierungstiefe wurden als Maß eingeführt, mit dem man die Kompliziertheit der Formeln ermitteln kann. Der modalen μ-Kalkül wird im nächsten Kapitel noch ausführlicher behandelt. Es werden einige wichtige Sätze vorgestellt und wird ein Algorithmus entworfen, mit dem die Erfüllbarkeit der μ-Kalkül-Formeln in einem Modell geprüft werden kann.

Kapitel 3

Model-Checking

Model-Checking ist ein Verifikationsverfahren, mit dem man prüfen kann, ob reaktive Systeme gewisse Eigenschaften erfüllen.

getestet werden können. Das Verfahren hat sich in den letzten 20 Jahren als eine sehr nützliche Technik erwiesen, so dass viele Model-Checking-Algorithmen in unterschiedlichen Anwendungsbereichen entworfen, implementiert und getestet wurden.

Ein großer Vorteil des Model-Checkings liegt darin, dass die gewünschten Eigenschaften eines Systems formal beschrieben und verifiziert werden können. Bei anderen Verfahren wie zum Beispiel „Simulation und Testen“, muss man die Testfälle auswählen, die während des Systemlaufs vorkommen können. Da man normalerweise aus zeitlichen Gründen nicht alle Fälle betrachten kann, ist keine hundertprozentige Sicherheit garantiert. Vor allem kann ein menschlicher Denkfehler beim Testen übersehen werden.

Im Gegensatz dazu ist das Model-Checking ein Verfahren, in dem alle mögliche Fälle überdeckt werden. Die gewünschten Eigenschaften werden in eine logische Sprache transformiert und geprüft, ob sie in einem gegebenen System erfüllt werden. Ein weiterer Vorteil des Model-Checkings ist, dass es völlig automatisch durchgeführt werden kann, so dass man Zeit und Kosten sparen kann.

In diesem Kapitel wird ein Model-Checking-Algorithmus vorgestellt. Zunächst werden die grundlegenden Begriffe, sowie einige wichtige Sätze für das Model-Checking angegeben. Danach wird ein Model-Checking-Algorithmus vorgestellt und schließlich die Berechnungsschritte sowie einige Verbesserungen des Algorithmus [CKS92] anhand eines Beispiels illustriert.

3.1 Fixpunktsatz

In diesem Abschnitt werden grundlegende Definitionen und die für den weiteren Verlauf dieser Arbeit benötigten Sätze bereitgestellt. Zunächst werden die Begriffe wie Halbordnung sowie Kette eingeführt. Dann wird der Fixpunktsatz von Knaster-Tarski sowie Kleene vorgestellt, der eine zentrale Rolle zur Verifikation der mit dem modalen μ-Kalkül ausgedrückten Eigenschaften in einem Modell spielt. Eine algorithmische Umsetzung der Fixpunktsätze wird erst in dem nächsten Abschnitt vorgestellt.

Definition 3.1.1 (*Partielle Ordnung, Halbordnung*)
Sei $\mathcal{D}$ eine Menge. Eine zweistellige Relation $\sqsubseteq$ auf $\mathcal{D}$ heißt partielle Ordnung oder Halbordnung, falls für alle $d_1, d_2, d_3 \in \mathcal{D}$ gilt:

- $d_1 \sqsubseteq d_1$ (*Reflexivität*)
- $d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_3$ *impliziert* $d_1 \sqsubseteq d_3$ (*Transitivität*)
- $d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_1$ *impliziert* $d_1 = d_2$ (*Antisymmetrie*)

Das Paar $(\mathcal{D}, \sqsubseteq)$ nennen wir dann eine partiell geordnete Menge. Für eine gegebene partielle Ordnung $\sqsubseteq_{\mathcal{D}}$ verwenden wir auch die Bezeichnung $\sqsupseteq_{\mathcal{D}}$ mit $d_1 \sqsubseteq d_2 \Leftrightarrow d_2 \sqsupseteq d_1$. Es ist zu beachten, dass zwei Elemente einer partiell geordneten Menge nicht unbedingt miteinander vergleichbar sein müssen. Um zu verdeutlichen, auf welcher Menge eine Relation definiert ist, verwenden wir auch die Bezeichnung $\sqsubseteq_{\mathcal{D}}$.

Es ist zu beachten, dass die Teilmengenrelation $\subseteq$ auf der Potenzmenge 2^M einer Menge M eine partielle Ordnung ist.

Definition 3.1.2 (*Supremum*)
Sei $\sqsubseteq$ eine partielle Ordnung auf $\mathcal{D}$. Ein Element $d \in \mathcal{D}$ ist kleinste obere Schranke (*Supremum*) *einer Teilmenge $\mathcal{T} \subseteq \mathcal{D}$, wenn die folgenden Bedingungen gelten:*

- $d_1 \sqsubseteq d$ *für alle* $d_1 \in \mathcal{T}$ (*obere Schranke*)
- $d \sqsubseteq d'$ *für alle obere Schranke* d' *von* $\mathcal{T}$.

Definition 3.1.3 (*Infimum*)
Sei $\sqsubseteq$ eine partielle Ordnung auf $\mathcal{D}$. Ein Element $d \in \mathcal{D}$ ist größte untere Schranke (*Infimum*) *einer Teilmenge $\mathcal{T} \subseteq \mathcal{D}$, wenn die folgenden Bedingungen gelten:*

- $d \sqsubseteq d_1$ *für alle* $d_1 \in \mathcal{T}$ (*untere Schranke*)
- $d' \sqsubseteq d$ *für alle untere Schranke* d' *von* $\mathcal{T}$.

Mit $\mathsf{sup}(\mathcal{T})$ bzw. $\mathsf{inf}(\mathcal{T})$ bezeichnen wir ein Supremum bzw. ein Infimum von $\mathcal{T} \subseteq \mathcal{D}$.

Definition 3.1.4 (*Kette*)
Sei $\sqsubseteq$ *eine partielle Ordnung auf* $\mathcal{D}$. *Eine Teilmenge* $\mathcal{K} \subseteq \mathcal{D}$ *heißt Kette, falls für alle* $d_1, d_2 \in \mathcal{K}$ *gilt:* $d_1 \sqsubseteq d_2$ *oder* $d_2 \sqsubseteq d_1$.

Ist $\mathcal{K}$ eine Kette einer partiell geordnete $(\mathcal{D}, \sqsubseteq)$ und besitzt $\mathcal{K}$ ein Supremum bzw. ein Infimum, dann bezeichnen wir dies insbesondere mit $\bigsqcup_{\mathcal{D}} \mathcal{K}$ bzw. $\bigsqcap_{\mathcal{D}} \mathcal{K}$. Wenn die Grundmenge $\mathcal{D}$ endlich ist, ist jede Kette $\mathcal{K} \subseteq \mathcal{D}$ auch endlich und es gelten nach der Definition $\bigsqcup \mathcal{K} \in \mathcal{K}$ und $\bigsqcap \mathcal{K} \in \mathcal{K}$.

Definition 3.1.5 (*Kettenvollständigkeit*)
Eine partielle Ordnung heißt kettenvollständig, falls jede ihrer Ketten ein Supremum sowie ein Infimum besitzt.

Für den Fall $\mathcal{T} = \mathcal{D}$ wird die Bezeichnung

$$\top := \mathsf{sup}(\mathcal{D}) \text{ sowie } \bot := \mathsf{inf}(\mathcal{D})$$

verwendet. Die Teilmengerelation $\subseteq$ auf der Potenzmenge 2^M einer Menge M ist z.B. eine kettenvollständige partielle Ordnung.

Definition 3.1.6 (*Monotonie*)
Sei $\sqsubseteq$ *eine kettenvollständige partielle Ordnung auf* $\mathcal{D}$. *Eine Funktion* $f : \mathcal{D} \to \mathcal{D}$ *heißt monoton, wenn* $d_1 \sqsubseteq d_2 \Rightarrow f(d_1) \sqsubseteq f(d_2)$ *für alle* $d_1, d_2 \subseteq \mathcal{D}$ *gilt.*

Definition 3.1.7 (*Stetigkeit*)
Sei $\sqsubseteq$ *eine kettenvollständige partielle Ordnung auf* $\mathcal{D}$. *Eine Funktion* $f : \mathcal{D} \to \mathcal{D}$ *heißt stetig, falls* $f(\bigsqcup \mathcal{K}) = \bigsqcup f(\mathcal{K})$ *sowie* $f(\bigsqcap \mathcal{K}) = \bigsqcap f(\mathcal{K})$ *für jede nichtleere Kette* $\mathcal{K}$ *in* $\mathcal{D}$ *gilt.*

Die Stetigkeit ist eine stärkere Eigenschaft als die Monotonie, so dass jede stetige Funktion bereits monoton ist: Für alle $a, b \in \mathcal{D}$ mit $a \sqsubseteq b$ ist $\mathcal{K} := \{a, b\}$ eine Kette, da $a \sqcup b = b$ und $a \sqcap b = a$ gelten, wobei $a \sqcup b := \bigsqcup\{a, b\}$ sowie $a \sqcap b := \bigsqcap\{a, b\}$. Aus der Stetigkeit von f ergibt sich dann

$$f(a) \sqsubseteq f(a) \sqcup f(b) = \bigsqcup f(\mathcal{K}) = f(\bigsqcup \mathcal{K}) = f(a \sqcup b) = f(b).$$

Somit ist f monoton.

Die umgekehrte Richtung gilt normalerweise nicht.
Eine monotone Funktion f auf einem endlichen $\mathcal{D}$ ist jedoch stetig, da jede Kette $\mathcal{K} \subseteq \mathcal{D}$ endlich ist, so dass $d_{min}, d_{max} \in \mathcal{K}$ mit $d_{min} \sqsubseteq d \sqsubseteq d_{max}$ für alle $d \in \mathcal{K}$ existiert, und $\bigsqcup f(\mathcal{K}) = f(d_{max}) = f(\bigsqcup \mathcal{K})$ sowie $\bigsqcap f(\mathcal{K}) = f(d_{min}) = f(\bigsqcap \mathcal{K})$ gilt.

Definition 3.1.8 (*Fixpunkt*)
Ein Fixpunkt fix *einer Funktion* $f : \mathcal{D} \to \mathcal{D}$ *ist ein Element* $x \in \mathcal{D}$ *mit* $f(x) = x$.

Der kleinste bzw. größte Fixpunkt einer Funktion f wird in dieser Arbeit mit $\mathsf{fix}_\mu f$ bzw. $\mathsf{fix}_\nu f$ bezeichnet. Sind Verwechslungen über die Funktion f ausgeschlossen, dann schreiben wir auch abkürzend fix_μ sowie fix_ν.

Satz 3.1.9 (*Fixpunktsatz von Knaster-Tarski* [*Tar55*])
Sei $\sqsubseteq$ *eine kettenvollständige partielle Ordnung auf* $\mathcal{D}$ *und* $f : \mathcal{D} \to \mathcal{D}$ *eine stetige Funktion. Dann existiert der kleinste Fixpunkt* fix_μ *sowie der größte Fixpunkt* fix_ν*, der sich wie folgt berechnen lässt:*

$$\mathsf{fix}_\mu := \bigsqcap\{a \in \mathcal{D} \mid f(a) \sqsubseteq a\} \text{ und } \mathsf{fix}_\nu := \bigsqcup\{a \in \mathcal{D} \mid a \sqsubseteq f(a)\}$$

Der Fixpunktsatz von Knaster-Tarski besagt, dass der kleinste Fixpunkt sowie der größte Fixpunkt einer stetigen Funktion eindeutig bestimmt werden kann. Für den praktischen Zweck bei der Bestimmung eines Fixpunktes ist der Satz jedoch nicht gut geeignet, da man den Funktionswert $f(a)$ für alle $a \in \mathcal{D}$ berechnet. Der folgende Fixpunktsatz von Kleene ermöglicht uns hingegen, dass man den kleinsten Fixpunkt bzw. den größten Fixpunkt durch iterative Anwendung der Funktion f bestimmen kann. Um die mehrfachen Anwendungen der Funktion f einfach zu bezeichnen, definieren wir die Funktion $f^i : \mathcal{D} \to \mathcal{D}$ induktiv, so dass

- $f^0(\bot) := \bot$ bzw. $f^0(\top) := \top$

- $f^i(\bot) := f(f^{i-1}(\bot))$ bzw. $f^i(\top) := f(f^{i-1}(\top))$ für alle $i \geq 1$.

Satz 3.1.10 (*Fixpunktsatz von Kleene [Kl52]*)
Sei $\sqsubseteq$ eine kettenvollständige partielle Ordnung auf $\mathcal{D}$ und $f : \mathcal{D} \to \mathcal{D}$ eine stetige Funktion. Dann gelten

$$\mathsf{fix}_\mu = \bigsqcup\{f^i(\bot) \mid i \geq 0\} \text{ und } \mathsf{fix}_\nu = \bigsqcap\{f^i(\top) \mid i \geq 0\}.$$

Die Teilmengenrelation $\subseteq$ ist offensichtlich eine kettenvollständige partielle Ordnung auf der Potenzmenge 2^S einer Zustandsmenge S. Wird es nachgewiesen, dass die Semantikfunktion $S \mapsto [\![\phi]\!]^M_{e[S'/X]}$ stetig ist, dann kann der obige Satz wie das folgende Korollar angewandt werden.

Wir haben vorausgesetzt, dass alle μ-Kalkül-Formeln in dieser Arbeit wohlgeformt sowie geschlossen sind und vor der Berechnung der Fixpunkte bereits in positiver Normalform umgewandelt werden. Diese Einschränkung ist notwendig, damit die Semantikfunktion monoton und stetig ist. Es ist zu beachten, dass die Formeln in positiver Normalform auch geschlossen bleiben.

Da die Zustandsmenge S endlich ist, ist ihre Potenzmenge auch endlich. Wir brauchen also lediglich zu zeigen, dass die Semantikfunktion monoton ist, um ihre Stetigkeit nachzuweisen.

Lemma 3.1.11
Für alle μ-Kalkül-Formeln Φ in positiver Normalform, alle Variablen $X \in Var$ und alle Modelle $M = (S, Act, AP, \to, L)$ mit $S_1, S_2 \subseteq S$ gilt:

$$S_1 \subseteq S_2 \Rightarrow [\![\Phi]\!]_{e[S_1/X]} \subseteq [\![\Phi]\!]_{e[S_2/X]}$$

Beweis: (durch strukturelle Induktion nach Φ)

- Für den Fall $\Phi = p \in AP$ oder $\Phi = Y \neq X$ gilt
  $[\![\Phi]\!]_{e[S_1/X]} = [\![\Phi]\!]_{e[S_2/X]}$
- Für den Fall $\Phi = X$ gilt
  $[\![\Phi]\!]_{e[S_1/X]} = [\![X]\!]_{e[S_1/X]} = S_1 \subseteq S_2 = [\![X]\!]_{e[S_2/X]} = [\![\Phi]\!]_{e[S_2/X]}$

- Für den Fall $\Phi = \phi_1 \vee \phi_2$ gelten die folgenden Gleichungen und Mengeninklusion:
$$\begin{aligned}
[\![\Phi]\!]_{e[S_1/X]} &= [\![\phi_1 \vee \phi_2]\!]_{e[S_1/X]} \\
&= [\![\phi_1]\!]_{e[S_1/X]} \cup [\![\phi_2]\!]_{e[S_1/X]} \text{ (nach Definition von } [\![\cdot]\!]) \\
&\subseteq [\![\phi_1]\!]_{e[S_2/X]} \cup [\![\phi_2]\!]_{e[S_2/X]} \text{ (nach I.V. und wegen } S_1 \subseteq S_2) \\
&= [\![\phi_1 \vee \phi_2]\!]_{e[S_2/X]} \text{ (nach Definition von } [\![\cdot]\!]) \\
&= [\![\Phi]\!]_{e[S_2/X]}
\end{aligned}$$
- Für den Fall $\Phi = \phi_1 \wedge \phi_2$ verfährt man analog.
- Für den Fall $\Phi = \langle a \rangle \phi$ gelten:
$$\begin{aligned}
[\![\Phi]\!]_{e[S_1/X]} &= [\![\langle a \rangle \phi]\!]_{e[S_1/X]} \\
&= \{s \mid \exists\ s' : s \xrightarrow{a} s' \wedge s' \in [\![\phi]\!]_{e[S_1/X]}\} \text{ (nach Definition von } \langle\,\rangle) \\
&\subseteq \{s \mid \exists\ s' : s \xrightarrow{a} s' \wedge s' \in [\![\phi]\!]_{e[S_2/X]}\} \text{ (wegen } S_1 \subseteq S_2) \\
&= [\![\langle a \rangle \phi]\!]_{e[S_2/X]} \\
&= [\![\Phi]\!]_{e[S_2/X]}
\end{aligned}$$
- Für den Fall $\Phi = [a]\phi$ verfährt man analog.
- Für den Fall $\Phi = \mu Y.\phi$ führen wir eine Fallunterscheidung durch.
  - Falls $X = Y$ ist, gelten
$$\begin{aligned}
[\![\Phi]\!]_{e[S_1/X]} &= [\![\mu Y.\phi]\!]_{e[S_1/X]} \\
&= \bigcap\{S' \subseteq S \mid [\![\phi]\!]_{e[S'/Y,S_1/X]} \subseteq S'\} \text{ (nach Definition von } [\![\cdot]\!]) \\
&= \bigcap\{S' \subseteq S \mid [\![\phi]\!]_{e[S'/Y]} \subseteq S'\} \text{ (da } X = Y) \\
&= \bigcap\{S' \subseteq S \mid [\![\phi]\!]_{e[S'/Y,S_2/X]} \subseteq S'\} \\
&= [\![\mu Y.\phi]\!]_{e[S_2/X]} \text{ (nach Definition von } [\![\cdot]\!]) \\
&= [\![\Phi]\!]_{e[S_2/X]}
\end{aligned}$$
  - Falls $X \neq Y$ ist, gelten
$$\begin{aligned}
[\![\Phi]\!]_{e[S_1/X]} &= [\![\mu Y.\phi]\!]_{e[S_1/X]} \\
&= \bigcap\{S' \subseteq S \mid [\![\phi]\!]_{e[S'/Y,S_1/X]} \subseteq S'\} \text{ (nach Definition von } [\![\cdot]\!]) \\
&\subseteq \bigcap\{S' \subseteq S \mid [\![\phi]\!]_{e[S'/Y,S_2/X]} \subseteq S'\} \\
&\qquad \text{(da } X \neq Y \text{ und nach I.V. mit } S_1 \subseteq S_2) \\
&= [\![\mu Y.\phi]\!]_{e[S_2/X]} \text{ (nach Definition von } [\![\cdot]\!]) \\
&= [\![\Phi]\!]_{e[S_2/X]}
\end{aligned}$$
- Für den Fall $\Phi = \nu X.\Psi$ verfährt man analog.

Insgesamt gilt die Behauptung. □

Die folgende Folgerung zeigt, dass der Fixpunktsatz von Kleene zur Berechnung der Fixpunkte direkt angewandt werden kann.

Korollar 3.1.12
Sei $M = (S, Act, AP, \rightarrow, L)$ ein Modell, e eine Umgebung und $\mu X.\phi$ bzw. $\nu X.\phi$ eine μ-Kalkül-Formel. Dann ist $[\![\mu X.\phi]\!]_e^M$ der kleinste (bzgl. $\subseteq$) Fixpunkt der Abbildung $S' \mapsto [\![\phi]\!]_{e[S'/X]}^M$ mit $S' \subseteq S$. Genauso ist $[\![\nu X.\phi]\!]_e^M$ der größte Fixpunkt dieser Abbildung.

Beweis: Nach dem Fixpunktsatz 3.1.10 von Kleene sowie Lemma 3.1.11 und wegen Endlichkeit der Potenzmenge 2^S gilt die Behauptung. □

Den kleinsten bzw. größten Fixpunkt einer stetigen Funktion f kann man also nach dem Fixpunktsatz von Kleene dadurch bestimmen, dass man die Funktionswerte $f^1(\bot), f^2(\bot), \cdots, f^i(\bot)$ bzw. $f^1(\top), f^2(\top), \cdots, f^i(\top)$ berechnet, bis $f^i(\bot) = f^{i+1}(\bot)$ bzw. $f^i(\top) = f^{i+1}(\top)$ gilt. Da die Funktionswerte $f^1(\bot), f^2(\bot), \cdots, f^i(\bot)$ bzw. $f^1(\top), f^2(\top), \cdots, f^i(\top)$ der Reihenfolge nach aufbauend berechnet werden können, ist der Fixpunktsatz von Kleene zur Implementierung gut geeignet. Man benötigt zwar noch eine möglichst optimale Datenstruktur sowie eine effiziente Verwaltung der Daten, aber der obige Satz kann bereits als ein impliziter Algorithmus angesehen werden, in dem eine Iteration zur Berechnung des Fixpunktes durchgeführt wird. In dem nächsten Abschnitt wird ein Model-Checking-Algorithmus vorgestellt, der unmittelbar auf dem Fixpunktsatz von Kleene basiert. Dann wird eine Verbesserung des Algorithmus gezeigt.

3.2 Model-Checking für den modalen μ-Kalkül

In diesem Abschnitt wird zunächst ein globaler Model-Checking-Algorithmus für den modalen μ-Kalkül vorgestellt, der prüft, ob die Eingabeformel in jedem Zustand eines gegebenen Modells (Kripke-Transitionssystem) erfüllt ist. Der Algorithmus kann als eine direkte Umsetzung der Semantik des modalen μ-Kalküls angesehen werden. Zur Behandlung der Fixpunktoperatoren wird allerdings der Fixpunktsatz von Kleene angewandt. Der Algorithmus berechnet den Fixpunkt sukzessiv nach der Struktur der Eingabeformel. Für jeden Operator der Formel wird die Berechnungsfunktion rekursiv aufgerufen, so dass die Fixpunkte der Teilformeln zuerst berechnet werden und dann daraus der Fixpunkt der Formel bestimmt wird. Falls der betrachtende Operator ein Fixpunktoperator ist, wird eine repeat-until-Schleife

durchgeführt, so dass der Fixpunkt iterativ berechnet wird. Der Algorithmus hat eine einfache Struktur, so dass man ihn leicht verstehen kann, aber er hat eine hohe Laufzeitkomplexität. Der Laufzeitaufwand ist exponentiell bzgl. Schachtelungstiefe der Eingabeformel. Danach wird ein Model-Checking-Algorithmus von Emerson und Lei [EmLe86] vorgestellt, dessen Laufzeit exponentiell bzgl. Alternierungstiefe ist. Anschließend wird die Abfolge der Berechnungschritte anhand eines Beispiels betrachtet.

In dem weiteren Verlauf dieser Arbeit wird Kripke-Transitionssystem stets als Modell verwendet, wie sie in dem letzten Kapitel definiert wurden. Wir gehen also davon aus, dass ein Modell $M = (S, Act, AP, \rightarrow, L)$ gegeben wobei S eine endliche Menge von Zuständen, Act eine endliche Menge von Aktionen, AP eine endliche Menge von atomaren Propositionen, $\rightarrow \subseteq S \times Act \times S$ eine Transitionsrelation und $L : S \rightarrow 2^{AP}$ eine Markierungsfunktion ist, die jedem Zustand eine Menge von atomaren Propositionen zuordnet.

Im Folgenden wird ein Algorithmus mc_naiv in Pseudo-Code vorgestellt, der die Menge der Zustände aus einem Modell M berechnet, in denen eine Eingabeformel Φ gilt. Der zweite Parameter der Funktion mc_naiv steht für die Belegung der Variablen. mc_naiv(Φ, e) bedeutet, dass die Berechnung der Funktion mc_naiv mit der Formel Φ weitergemacht wird, in der die Variablen nach der Belegung e ersetzt werden. Da diese Belegung während der Berechnung neu initialisiert und geändert wird, ist es praktisch, sie als Parameter zu übergeben.

function mc_naiv(Φ : *Formel,* e : *Belegung der Fixpunktvariablen*) : *Zustandsmenge*
// Gib die Menge der Zustände aus, die die Formel Φ *unter der Umgebung*
// e erfüllen.
var S_{new} : *Zustandsmenge*
begin
  if $\Phi = p$ then return $\{s \mid p \in L(s)\}$
  if $\Phi = \neg p$ then return $\{s \mid p \notin L(s)\}$
  if $\Phi = X$ then return $e(X)$
  if $\Phi = \phi_1 \wedge \phi_2$ then return mc_naiv$(\phi_1, e) \cap$ mc_naiv(ϕ_2, e)
  if $\Phi = \phi_1 \vee \phi_2$ then return mc_naiv$(\phi_1, e) \cup$ mc_naiv(ϕ_2, e)
  if $\Phi = \langle a \rangle \Psi$ then return $\{s \mid \exists\, t : s \xrightarrow{a} t \wedge t \in$ mc_naiv$(\Psi, e)\}$
  if $\Phi = [a] \Psi$ then return $\{s \mid \forall\, t : s \xrightarrow{a} t \Rightarrow t \in$ mc_naiv$(\Psi, e)\}$
  if $\Phi = \mu X.\Psi$ then

```
        S_new := ∅
        repeat
            e(X) := S_new
            S_new := mc_naiv(Ψ, e)
        until e(X) = S_new
        return S_new
    if Φ = νX.Ψ then
        S_new := S
        repeat
            e(X) := S_new
            S_new := mc_naiv(Ψ, e)
        until e(X) = S_new
        return S_new
end
```

Wie wir in dem letzten Kapitel vorausgesetzt haben, ist jede Eingabeformel wohlgeformt sowie geschlossen und wird per Preprocessing zu einer semantisch äquivalenten Formel in positiver Normalform transformiert. Die Regeln von dem Lemma 2.3.8 werden dafür angewandt. Es ist zu beachten, dass die Geschlossenheit der Formel erhalten bleibt. Außerdem ist die Länge der neu entstehenden Formel nur einen konstanten Faktor (nämlich maximal 2) länger als die ursprünglichen Formellänge. Das bedeutet, dass die Transformation der Eingabeformel auf der Laufzeitkomplexität keinen Einfluss hat.

Bei Ausführung der Funktion $\mathsf{mc\_naiv}(\Phi, e)$ wird die Formel Φ in Teilformeln zerlegt und die Funktionswerte für die Teilformeln werden zuerst berechnet. Da die rekursiven Aufrufe auf dieselbe Funktion eingeschränkt ist und keine andere Funktion innerhalb der Funktion benutzt wird, braucht man zur Analyse der Funktion nur zu betrachten, welche Berechnung für den einzelnen Operator durchgeführt wird.

Ist der aktuell betrachtende Operator kein Fixpunktoperator, dann wird die Semantikfunktion von Definition 2.3.3 so realisiert, wie sie definiert ist. Ist der Operator ein Fixpunktoperator ist, kann man zwar auch die Semantikfunktion genauso realisieren. Jedoch müssen dann alle mögliche Belegungen der Variablen ausprobiert werden, um die Menge der Zustandsmengen zu bestimmen, die die Bedingung der Definition erfüllen, damit die Vereinigung bzw. der Durchschnitt von den Zustands-

mengen berechnet werden kann. Dieser Vorgang ist normalerweise sehr aufwendig, und deshalb werden die Zustandsmengen in der Regel auf andere Weise selektiert, so dass die Berechnung insgesamt einfacher wird. Normalerweise wird der Fixpunktsatz von Kleene hierfür angewandt. Wir betrachten zunächst, wie diese Fälle in der Funktion mc_naiv behandelt werden. Dann werden wir zeigen, dass die Berechnungsschritte in mc_naiv genau dem Fixpunktsatz von Kleene entspricht.

Für den Fixpunkoperator ν bzw. μ werden die Belegungen der Variablen mit S bzw. $\emptyset$ initialisiert und und die neue Belegung berechnet. Die Berechnung der neuen Belegung wird iterativ weiter durchgeführt, bis sich die Belegung stabilisiert hat, d.h. bis die alte und neue Belegungen identisch sind. Enthält die Eingabeformel nur eine Art der ν- bzw. μ-Operatoren, ist die Berechnung der Fixpunkte relativ einfach, da die von einer Variable belegten Zustände immer weniger bzw. mehr werden (Monotonieeigenschaft). Wenn die Formel jedoch mit ν- und μ-Operatoren alternierend geschachtelt ist, muss die Belegung der Fixpunktvariable von Teilformeln jedes Mal erneut initialisiert werden, sobald die Belegung der Fixpunktvariable der Formel verändert wird.

Wir definieren eine Funktion $F^i_{\Phi,e}$ wie folgt, so dass wir die Veränderung der Zustandsmenge S_{new} in der Funktion mc_naiv bzgl. dem Fixpunktsatz von Kleene genau betrachten können:

- $F_{\Phi,e}(S') := [\![\Phi]\!]_{e[S'/X]}$
- $F^0_{\Phi,e}(S') := S'$
- $F^{i+1}_{\Phi,e}(S') := F_{\Phi,e}(F^i_{\Phi,e}(S'))$ für alle $i \geq 0$.

Durch die repeat-until-Schleife wird demnach $F^i_{\Phi,e}(S)$ bzw. $F^i_{\Phi,e}(\emptyset)$ für die Formel $\nu X.\Phi$ bzw. $\mu X.\Phi$ berechnet, bis $F^i_{\Phi,e}(S) = F^{i+1}_{\Phi,e}(S)$ bzw. $F^i_{\Phi,e}(\emptyset) = F^{i+1}_{\Phi,e}(\emptyset)$ gilt. Die Iteration wird erst dann gestoppt, wenn der minimale Index i gefunden wird, mit dem die Abbruchbedingung erfüllt wird. Die Berechnungsschritte für $\nu X.\Phi$ unterscheiden sich von denen für $\mu X.\Phi$ dadurch, dass die Zustandsmenge S_{new} mit S anstatt mit $\emptyset$ initialisiert wird. Um diesen Unterschied zu verdeutlichen, wurde die Initialisierungsroutine von S_{new} in der Funktion mc_naiv hervorgehoben.

Da die Funktion $F_{\Phi,e}$ monoton nach Lemma 3.1.11 ist und die Potenzmenge 2^S endlich ist, ist die Funktion $F_{\Phi,e}$ stetig. Nach dem Fixpunktsatz 3.1.10 sowie Korollar 3.1.12 gelten

$$\mathsf{fix}_\mu = \bigcap\{F^i_{\Phi,e}(\emptyset) \mid i \geq 0\} \text{ und } \mathsf{fix}_\nu = \bigcup\{F^i_{\Phi,e}(S) \mid i \geq 0\}.$$

Es ist zu beachten, dass die Mengeninklusionen

$$F^0_{\Phi,e}(\emptyset) \subseteq F^1_{\Phi,e}(\emptyset) \subseteq \cdots \subseteq F^i_{\Phi,e}(\emptyset) \subseteq \cdots \text{ und}$$
$$F^0_{\Phi,e}(S) \supseteq F^1_{\Phi,e}(S) \supseteq \cdots \supseteq F^i_{\Phi,e}(S) \supseteq \cdots \text{ gelten,}$$

da die Funktion $F_{\Phi,e}$ monoton und 2^S endlich ist. Daraus folgt

$$\mathsf{fix}_\mu = \{F^i_{\Phi,e}(\emptyset) \mid F^i_{\Phi,e}(\emptyset) = F^{i+1}_{\Phi,e}(\emptyset) \wedge i \text{ ist minimal}\} \text{ sowie}$$
$$\mathsf{fix}_\nu = \{F^i_{\Phi,e}(S) \mid F^i_{\Phi,e}(S) = F^{i+1}_{\Phi,e}(S) \wedge i \text{ ist minimal}\},$$

was exakt der Berechnung von mc_naiv für $\mu X.\Psi$ bzw. $\nu X.\Psi$ entspricht.

Der Model-Checking-Algorithmus sieht dann wie folgt aus:

```
function modelchecking_naiv(M : Modell, Φ : Formel) : Boolean
// Prüft, ob die Eingabeformel Φ in allen Zuständen des Modells M erfüllt wird.
var e := 0 :  Belegung der Fixpunktvariablen
begin
   if S = mc_naiv(Φ, e) then return „Ja“
                        else return „Nein“
end
```

Die Korrektheit des folgenden Satzes ergibt sich aus Korollar 3.1.12 und obigen Erklärungen.

Satz 3.2.1
Sei M ein endliches Modell und Φ eine geschlossene μ-Kalkül-Formel in positiver Normalform. Dann kann man mit der Funktion modelchecking_naiv(M, Φ) *prüfen, ob die Formel Φ in allen Zuständen von M erfüllen wird, d.h. $M \models \Phi$.*

Im Folgenden wird die Laufzeitkomplexität der Funktion modelchecking_naiv grob abgeschätzt.

Satz 3.2.2
Die Laufzeit der Funktion modelchecking_naiv(M, Φ) *beträgt $\mathcal{O}\left(|M| \cdot |\Phi| \cdot (|S| \cdot |\Phi|)^k\right)$, wobei k die Schachtelungstiefe der Formel Φ ist.*

Beweis: Da die Funktion modelchecking_naiv offenbar den gleichen Laufzeitaufwand wie mc_naiv hat, brauchen wir nur die Laufzeit von mc_naiv abzuschätzen.
Wenn man die rekursiven Ausführungen der Funktion mc_naiv für die Teilformeln nicht einbezieht, kostet die Ausführung einer booleschen bzw. einer modalen Operation maximal $\mathcal{O}(|M|)$. Da die Funktion mc_naiv für jede Teilformel einmal aufgerufen wird, hat man den Faktor $|\Phi|$.
Wir betrachten nun, wie oft die Funktion mc_naiv rekursiv durchgeführt werden kann. Für jeden Fixpunktoperator wird eine repeat-until-Schleife durchgeführt, in der die Funktion mit der Teilformel wieder aufgerufen wird. Während der Ausführung der Schleife wird die Zustandsmenge S_{new} als Belegung für die mit dem Fixpunktoperator gebundene Variable immer wieder erneut berechnet. Wenn man die so berechneten Zustandsmengen S_{new} nacheinander auflistet, erhält man eine Kette, d.h. eine Folge der echten Mengeninklusion, da die Funktion mc_naiv monoton bzgl. Knotenmenge ist. Da die Zustandsmenge S_{new} mit S bzw. $\emptyset$ für den größten bzw. kleinsten Fixpunktoperator initialisiert und iterativ berechnet wird, kann die erneute Berechnung von S_{new} maximal $|S|$-mal ausgeführt werden.
Andererseits entspricht die Schachtelungstiefe der maximalen Rekursionstiefe. Die Laufzeitkomplexität beträgt also insgesamt $\mathcal{O}(|M| \cdot |\Phi| \cdot (|S| \cdot |\Phi|)^k)$, wobei k die Schachtelungstiefe der Formel Φ ist. □

Die Funktion mc_naiv hat eine einfache Struktur, so dass sie leicht implementiert werden kann. Abgesehen davon, dass unsere Abschätzung der Laufzeit grob ist, hat die Funktion eine sehr schlechte Laufzeitkomplexität. Es ist zu beachten, dass die mit einem Fixpunktoperator gebundene Variable bei jedem Aufruf der Funktion mc_naiv erneut mit S_{new} initialisiert wird. Hierbei wird kein Unterschied gemacht, wie die Formel bzw. Fixpunktoperator aufgebaut ist. Falls zwei direkt geschachtelte Fixpunktoperatoren gleich sind, bleibt die Richtung der Mengeninklusionen auf Iterationsberechnung gleich, so dass eine erneute Initialisierung der Variable mit S_{new} für den Fixpunktoperator der Teilformel nicht in jedem Durchlauf der Iteration erforderlich ist.

Im Folgenden wird ein Model-Checking-Algorithmus eval_fp vorgestellt, der in [EmLe86] entworfen wurde. Der Algorithmus ist eine verbesserte Version des zuvor vorgestellten Model-Checking-Algorithmus mc_naiv. Die Grundidee des Algorithmus eval_fp ist, dass man unterscheidet, welche Variablen zusammen in einer Schleife berechnet werden können.

Der exponentielle Faktor der Komplexität von mc_naiv ist die Schachtelungstiefe der Eingabeformel, weil die repeat-until-Schleife für jeden Fixpunktoperator durchgeführt wird. Wegen der rekursiven Ausführung der Funktion werden somit die mit einem Fixpunktoperator gebundenen Variablen ständig erneut initialisiert. Wenn ein Fixpunktoperator direkt nacheinander in Formel auftritt, wie z.B. $\mu X.\mu Y.\Phi(X,Y)$, dann kann die Belegung der Variablen X und Y mit einem Schleifendurchlauf berechnet werden. Die beiden Variablen werden am Anfang mit der leeren Zustandsmenge initialisiert und während der Iterationsberechnung immer größere Zustandsmenge belegt. Wegen der Monotonieeigenschaft des μ-Operators braucht die Variable Y nicht erneut initialisiert zu werden, auch wenn die Belegung der Variable X geändert wird.

```
function eval_fp(Φ : Formel, e : Belegung der Fixpunktvariablen ) : Zustandsmenge
// Gib die Menge der Zustände aus, die die Formel Φ unter der Umgebung
// e erfüllen.
var S_new : Zustandsmenge
begin
   if Φ = p  then return {s | p ∈ L(s)}
   if Φ = ¬p  then return {s | p ∉ L(s)}
   if Φ = X then return e(X)
   if Φ = φ1 ∧ φ2 then return eval_fp(φ1, e) ∩ eval_fp(φ2, e)
   if Φ = φ1 ∨ φ2 then return eval_fp(φ1, e) ∪ eval_fp(φ2, e)
   if Φ = ⟨a⟩Ψ then return {s | ∃ t : s -a-> t ∧ t ∈ eval_fp(Ψ, e)}
   if Φ = [a]Ψ then return {s | ∀ t : s -a-> t ⇒ t ∈ eval_fp(Ψ, e)}
   if Φ = νX.Ψ then
      for each μ-Teilformel μY.φ von Ψ do
         e(Y) := ∅
      repeat
         S_new := e(X)
         e(X) := eval_fp(Ψ, e)
      until e(X) = S_new
      return S_new
   if Φ = μX.Ψ then
      for each ν-Teilformel νY.φ von Ψ do
         e(Y) := S
      repeat
```

$S_{new} := e(X)$
$e(X) :=$ eval_fp(Ψ, e)
until $e(X) = S_{new}$
return S_{new}
end

Die obige Funktion berechnet die Menge der Zustände, in denen die Formel Φ mit der Variablenbelegung e erfüllt. Die Variablen werden am Anfang mit S bzw. $\emptyset$ initialisiert, wenn sie mit dem größten bzw. kleinsten Fixpunktoperator gebunden sind. Der gesamte Algorithmus sieht dann wie folgt aus:

function modelchecking(M : *Modell,* Φ : *Formel*) : *Boolean*
// Prüft, ob die Eingabeformel Φ *in allen Zuständen des Modells* M *erfüllt wird.*
var e : *Belegung der Fixpunktvariablen*
begin
Finde alle ν*-Teilformeln* $\nu X_1.\phi_1, \ldots, \nu X_k.\phi_k$ *von* Φ
for each $1 \leq i \leq k$ do $e(X_i) := S$
Finde alle μ*-Teilformeln* $\mu Y_1.\phi_1, \ldots, \mu Y_{k'}.\phi_{k'}$ *von* Φ
for each $1 \leq i \leq k'$ do $e(Y_i) := \emptyset$
if $S =$ eval_fp(Φ, e) then return *„Ja“*
else return *„Nein“*
end

In dem Algorithmus modelchecking werden alle Fixpunktvariablen mit S bzw. $\emptyset$ initialisiert und dann wird die Funktion eval_fp mit der Eingabeformel aufgerufen, so dass die Menge der Zustände berechnet wird, die die Eingabeformel Φ erfüllen. Wenn alle Zustände die Formel erfüllen, gibt der Algorithmus die Antwort „Ja“, ansonsten „Nein“.

Wir betrachten nun den Unterschied zwischen den Funktionen mc_naiv und eval_fp. Die beiden Funktionen sind sehr ähnlich, bis der betrachtende Operator ein Fixpunktoperator ist. Die Funktion mc_naiv initialisiert die Fixpunktvariablen jedes Mal mit $\emptyset$ bzw. S für einen μ- bzw. ν-Operator, wenn die Funktion aufgerufen wird. Die Funktion eval_fp übergibt die Belegung der Fixpunktvariablen beim rekursiven Aufruf der Funktion für die Teilformel weiter. Die Fixpunktvariablen werden nur

dann neu initialisiert, wenn der Typ des betrachtenden Fixpunktoperators von ν zu μ bzw. von μ zu ν wechselt. Die Sequenz der repeat-until-Schleife kann länger werden, aber der exponentielle Faktor wird dafür kleiner, und zwar nicht mehr die Schachtelungstiefe, sondern die Alternierungstiefe.

Im Folgenden wird die Berechnung der Fixpunkte anhand eines Beispiels illustriert.

Beispiel 3.2.3 *Sei ein Modell $M = (S, Act, AP, \rightarrow, L)$ gegeben, wobei*

- $S = \{s, t, u, v\}$,
- $Act = \{a\}$,
- $AP = \{A\}$,
- $\rightarrow = \{(s,a,s), (s,a,t), (t,a,u), (u,a,s), (u,a,v), (v,a,v)\}$,
  $L(s) = \emptyset$, $L(t) = \{A\}$, $L(u) = \{A\}$ *und* $L(v) = \{A\}$.

Für die Formel $\Phi = \nu X.\mu Y.([a]((A \wedge X) \vee Y))$ wird die Erfüllbarkeit der Zustände geprüft.

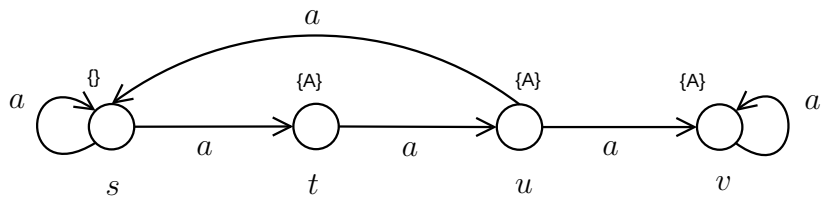

Abbildung 3.1: Modell

Intuitive Interpretation der Eingabeformel ist, dass die Proposition A unendlich oft auf jedem a Pfad gilt. Zur einfacheren Darstellung der Berechnung werden zusätzliche Variablen wie folgt eingeführt:

$$\begin{aligned} Z_1 &:= [a]((A \wedge X) \vee Y) \\ Z_2 &:= (A \wedge X) \vee Y \\ Z_3 &:= A \wedge X \\ Z_4 &:= A \end{aligned}$$

Wir verwenden die Funktion F_ν bzw. F_μ zur Darstellung der Iteration für den ν- bzw. μ-Operator. Dann kann man die Berechnung der Variablenbelegungen wie folgt tabellarisch darstellen:

X	Y	Z_4	Z_3	Z_2	Z_1	
S	$\emptyset$	$\{t,u,v\}$	$\{t,u,v\}$	$\{t,u,v\}$	$\{t,v\}$	$F^1_{\mu,S}(\emptyset)$
S	$\{t,v\}$	$\{t,u,v\}$	$\{t,u,v\}$	$\{t,u,v\}$	$\{t,v\}$	$F^2_{\mu,S}(\emptyset) = F^1_{\nu}(S)$
$\{t,v\}$	$\emptyset$	$\{t,u,v\}$	$\{t,v\}$	$\{t,v\}$	$\{v\}$	$F^1_{\mu,\{t,v\}}(\emptyset)$
$\{t,v\}$	$\{v\}$	$\{t,u,v\}$	$\{t,v\}$	$\{t,v\}$	$\{v\}$	$F^2_{\mu,\{t,v\}}(\emptyset) = F^2_{\nu}(S)$
$\{v\}$	$\emptyset$	$\{t,u,v\}$	$\{v\}$	$\{v\}$	$\{v\}$	$F^1_{\mu,\{v\}}(\emptyset)$
$\{v\}$	$\{v\}$	$\{t,u,v\}$	$\{v\}$	$\{v\}$	$\{v\}$	$F^2_{\mu,\{v\}}(\emptyset) = F^3_{\nu}(S)$

Abbildung 3.2: Veränderung der Variablenbelegung

Die Variablen X bzw. Y werden zunächst mit S bzw. $\emptyset$ initialisiert, da sie mit dem Fixpunktoperator ν bzw. μ gebunden sind. Die Belegung der Variable Z_4 bleibt konstant mit $\{t,u,v\}$. Dann wird berechnet, auf welchen Zuständen die Teilformeln mit der Belegung erfüllt sind. Die erste Zeile zeigt die Bewertung aller Teilformeln mit dem Initialwert S bzw. $\emptyset$ für die Fixpunktvariablen X bzw. Y.

In der zweiten Zeile wird die Variable Y nach dem bisherigen Ergebnis neu bewertet, so dass $e(Y) := \{t,v\}$. Anschließend werden die restlichen Teilformeln mit der Belegung S bzw. $\{t,v\}$ für X bzw. Y bewertet. Da die Belegung der Variable Y sich stabilisiert hat, d.h. $F^2_{\mu,S}(\emptyset) = F^3_{\mu,S}(\emptyset)$, wird die Iteration für den μ-Operator beendet. Dann wird die Bewertung der Variable X mit dem bisherigen Ergebnis neu berechnet.

Die Variable X erhält eine neue Belegung $\{t,v\}$ aus dem Ergebnis $F^2_{\mu,S}(\emptyset) = \{t,v\}$. Da die Bewertung von X sich verändert hat, wird die Bewertung der Variable Y nun neu berechnet. Die Variable Y wird mit $\emptyset$ neu initialisiert. Dann werden die restlichen Teilformeln unter der Belegung von X und Y bewertet.

In der vierten Zeile befindet sich die Bewertung der Teilformeln und daraus folgt die neue Bewertung von Y mit $e(Y) := \{v\}$. Wenn man die Iteration für den μ-Operator durchlaufen lässt, findet man heraus, dass $F^2_{\mu,\{t,v\}}(\emptyset) = F^3_{\mu,\{t,v\}}(\emptyset)$ gilt. Somit hat sich die Bewertung der Variable X_2 stabilisiert und wir beenden die Iteration für den μ-Operator.

In der fünften Zeile sieht man die Bewertung der Teilformeln unter der Belegung der Variablen X bzw. Y mit $\{v\}$ bzw. $\emptyset$. Anschließend wird die Variable Y mit $\{v\}$ neu initialisiert und dann die Bewertung der Teilformeln unter der geänderten Variablenbelegung weiter berechnet.

Zum Schluss erhält man die Belegung der sechsten Zeile, da $F^2_{\mu,\{v\}}(\emptyset) = F^3_{\mu,\{v\}}(\emptyset)$ sowie $F^3_\nu(S) = F^4_\nu(S) = \{v\}$ gilt. Die Formel Φ ist lediglich in dem Zustand v erfüllt und ansonsten nicht.

Normalerweise wird die Bewertung der Teilformeln nicht in obiger Form sondern in einer Matrix gespeichert, dessen Einträge jeweils ein Wahrheitswert für (ϕ, m) sind. $(\phi, m) = 1$ bedeutet, dass die Formel ϕ in dem Zustand m erfüllt ist. $(\phi, m) = 0$ bedeutet hingegen, dass die Formel in dem Zustand m nicht erfüllt ist. Unter Benutzung einer solchen Matrix ist die Bewertung der Zustände bzgl. Teilformeln leicht veränderbar. Außerdem kann man die Einträge als Zwischenergebnis effektiv verwalten und wiederverwenden.

	s	t	u	v
X	1	1	1	1
Y	0	0	0	0
Z_1	0	1	0	1
Z_2	0	1	1	1
Z_3	0	1	1	1

	s	t	u	v
X	1	1	1	1
Y	0	1	0	1
Z_1	0	1	0	1
Z_2	0	1	1	1
Z_3	0	1	1	1

	s	t	u	v
X	0	1	0	1
Y	0	0	0	0
Z_1	0	0	0	1
Z_2	0	1	0	1
Z_3	0	1	0	1

	s	t	u	v
X	0	1	0	1
Y	0	0	0	1
Z_1	0	0	0	1
Z_2	0	1	0	1
Z_3	0	1	0	1

	s	t	u	v
X	0	0	0	1
Y	0	0	0	0
Z_1	0	0	0	1
Z_2	0	0	0	1
Z_3	0	0	0	1

	s	t	u	v
X	0	0	0	1
Y	0	0	0	1
Z_1	0	0	0	1
Z_2	0	0	0	1
Z_3	0	0	0	1

Abbildung 3.3: Diese Tabellen beinhalten die Veränderung der Zustandsbewertungen bzgl. Teilformeln.

Die erste Matrix entspricht der ersten Variablenbelegung in Abb. 3.2 und die zweite Matrix der zweiten Variablenbelegung, usw. Man erstellt am Anfang eine Matrix und aktualisiert die Bewertungen im Lauf der Berechnung dementsprechend.

Die Einträge $(0, 1, 1, 1)$ für die Variable Z_4 bleiben unverändert, da sie mit dem Modell bereits festgelegt werden, in welchem Zustand die Proposition A erfüllt ist, und werden deshalb ausgelassen.

Man kann die zusätzlich eingeführten Variablen des Beispiels 3.2.3 in Teilformeln

einsetzen, so dass die Teilformeln kürzer werden und ihre Bewertung vereinfacht wird. Man erhält dann die Gleichungen $Z_1 = [a]\, Z_2, Z_2 = Z_3 \vee Y, Z_3 = Z_4 \wedge X$ und $Z_4 = A$. Erweitert man solche Gleichung für die Fixpunktoperation, erhält man ein Gleichungssystem für die modale μ-Kalkül-Formel. Cleaveland, Klein und Steffen haben einen Model-Checking-Algorithmus in [CKS92] vorgestellt, in dem die zu untersuchende Formel in ein rekursives Gleichungssystem über μ-Kalkül-Variablen umwandelt wird. Durch sorgfältige Analyse des Gleichungssystems sowie geschickte Verwaltung der Variablenbelegung wird die Laufzeitkomplexität des Model-Checking-Algorithmus so verbessert, dass sie

$$\mathcal{O}\left(|M| \cdot |E| \cdot \left(\frac{|S| \cdot |E|}{rt(E)}\right)^{rt(E)-1}\right)$$

beträgt, wobei $|M|$ bzw. $|E|$ die Größe des Modells bzw. des Gleichungssystems und $|S|$ die Anzahl der Zustände im Modell ist.

Basierend auf dem Model-Checking-Algorithmus [CKS92] werden wir einen Spiel-Algorithmus in dem nächsten Kapitel entwerfen. Die grundlegenden Datenstrukturen dieses Model-Checking-Algorithmus werden wir auch für unseren Spiel-Algorithmus verwenden. In den zwei folgenden Abschnitten werden wir deshalb auf sie eingehen, wobei die Funktionsweise des Model-Checking-Algorithmus ebenfalls erläutert wird.

3.3 Gleichungssysteme

In diesem Abschnitt wird die Erstellung der Gleichungssysteme für die modalen μ-Kalkül-Formeln behandelt.

Definition 3.3.1 (*Gleichungssystem* [*CKS92*])
Die modalen μ-Kalkül-Formeln können in ein Gleichungssystem der folgenden Form umgewandelt werden.

$$\begin{bmatrix} X_1 & \trianglelefteq\!\triangleright_1 & \Phi_1 \\ X_2 & \trianglelefteq\!\triangleright_2 & \Phi_2 \\ & \vdots & \\ X_n & \trianglelefteq\!\triangleright_n & \Phi_n \end{bmatrix}$$

wobei $\trianglelefteq\!\triangleright_i \in \{=, =_\nu, =_\mu\}$ für alle $i \in \mathbb{N}$ gilt.

$X_1, \ldots, X_n$ sind Variablen und $\Phi_1, \ldots, \Phi_n$ Formeln, die keinen Fixpunktoperator enthalten. Die Gleichung $X_i =_\nu \Phi_i$ bzw. $X_i =_\mu \Phi_i$ stellt dabei eine maximale bzw. minimale Fixpunktformel $\nu X_i.\Phi$ bzw. $\mu X_i.\Phi$ dar. Von einer Formel zu einer semantisch äquivalenten Gleichungssystem wird so transformiert, dass man jeder Teilformel eine eigene Variable zugeordnet und die rechten Seiten jeder Gleichung entweder atomar oder eine einfache Disjunktion oder Konjunktion von zwei Variablen oder eine durch eine Modalität bewachte Variable sind.

Beispiel 3.3.2 *Wir betrachten ein Gleichungssystem, das die μ-Kalkül-Formel $\Phi = \nu X.\mu Y.[a]((A \wedge X) \vee Y)$ ausdrückt.*

$$\begin{bmatrix} X_1 & =_\nu & X_2 \\ X_2 & =_\mu & [a]X_3 \\ X_3 & = & X_4 \vee X_2 \\ X_4 & = & X_5 \wedge X_1 \\ X_5 & = & A \end{bmatrix}$$

Nach syntaktischen Substitutionen $[A/X_5]$, $[(X_5 \wedge X_1)/X_4]$ und $[(X_4 \vee X_2)/X_3]$ auf dem Gleichungssystem bleiben nun noch zwei erste Gleichungen: $X_1 =_\nu X_2$ und $X_2 =_\mu [a]((A \wedge X_1) \vee X_2)$. Ersetzt man das ν- bzw. μ-Gleichheitszeichen durch den ν- bzw. μ-Operator, dann hat man die Formel $\nu X_1.\mu X_2.[a]((A \wedge X_1) \vee X_2)$.

Wir ersparen uns, die Semantik solcher Gleichungssysteme formal anzugeben. In Semantik wird dafür gesorgt, dass eine Variable in einem Zustand eines Kripke-Transitionssystems genau dann gültig ist, wenn die entsprechende Teilformel in dem Zustand gilt. Es ist zu beachten, dass unendlich viele Gleichungssysteme existieren, die semantisch äquivalent sind. Um eine eindeutige Transformation von einer μ-Kalkül-Formel zu einem bestimmten Gleichungssystem zu gewährleisten, brauchen wir eine Übersetzungsfunktion, die immer Gleichungssysteme systematisch von gegebenen μ-Kalkül-Formeln konstruiert.

Bei einer solchen Übersetzungsfunktion wird es besonders auf die Erstellung einer Variablenordnung geachtet, so dass die Struktur der Formel wiederspiegelt wird. Die Variablenordnung entspricht der Reihenfolge des Vorkommens der Teilformeln von links nach rechts sowie der Größe der Teilformeln von der größten nach der kleinsten.

Im Beispiel 3.3.2 kann das Gleichheitszeichen $=$ durch $=_\nu$ bzw. $=_\mu$ ersetzt werden, da die Belegung der Variablen auf der linken Seite solcher Gleichungen nicht durch die

Initialisierung der Variablen sondern durch Berechnung der Formel auf der rechten Seite bestimmt werden. Die Gleichung $X_3 = X_4 \vee X_2$ hat beispielsweise die gleiche Bedeutung wie $X_3 =_\mu X_4 \vee X_2$, da die initiale Belegung der Variable X_3 keine Auswirkung darauf hat, welche Belegung die Variable X_3 schließlich erhalten wird. Das Gleichungssystem kann demnach wie folgt transformiert werden:

$$\begin{bmatrix} X_1 & =_\nu & X_2 \\ X_2 & =_\mu & [a]X_3 \\ X_3 & =_\mu & X_4 \vee X_2 \\ X_4 & =_\mu & X_5 \wedge X_1 \\ X_5 & =_\mu & A \end{bmatrix}$$

Wir betrachten nun eine Übersetzungsfunktion, die ein Gleichungssystem E erstellt, das mit einer Eingabeformel Φ semantisch äquivalent ist. Der erste Parameter der Funktion make_eqs ist eine Gleichung, in der eine unbenutzte Variable sowie die Teilformel, die zum Gleichungssystem transformiert werden soll, enthalten ist. Die Variablen werden dabei von dem niedrigsten zu dem höchsten Index nummeriert.

```
function translate(Φ : Eingabeformel ) : Gleichungssystem
// Berechne ein Gleichungssystem E, das zur gegebenen Eingabeformel Φ
// semantisch äquivalent ist, und gib es aus.
var E : Gleichungssystem
begin
   E := 0 // E ist am Anfang leer.
   make_eqs(⟨X_1 = Φ⟩, E)
   return E
end
```

```
function make_eqs(⟨X_i =_σ Φ⟩ : Gleichung,  E : Gleichungssystem) : int
// Füge aktuelle Gleichung in das Gleichungssystem E hinzu und gib den
// nächsten freien Index der Variable aus.
var j : int
begin
   if Φ = p ∈ AP then
      E := E + ⟨X_i =_σ p⟩
      return  i + 1
   if Φ = Y ∈ Var then
```

$E := E + \langle X_i =_\sigma Y \rangle$
return $i + 1$
if $\Phi = K\Psi$ then // K *ist entweder* $\langle a \rangle$ *oder* $[a]$ *für ein* $a \in Act$
$E := E + \langle X =_\sigma K X_{i+1} \rangle$
return make_eqs $(\langle X_{i+1} =_\sigma \Psi \rangle, E)$
if $\Phi = \phi \; op \; \psi$ with $op \in \{\vee, \wedge\}$ then
$j :=$ make_eqs$(\langle X_{i+1} =_\sigma \phi \rangle, E)$
$E := E + \langle X_i =_\sigma X_{i+1} \; op \; X_j \rangle$
return make_eqs $(\langle X_j =_\sigma \psi \rangle, E)$
if $\Phi = \sigma' Y.\Psi$ with $\sigma' \in \{\nu, \mu\}$ then
$E := E + \langle X_i =_{\sigma'} X_{i+1} \rangle$
$\Psi := \Psi[X_{i+1}/Y]$
return make_eqs $(\langle X_{i+1} =_{\sigma'} \Psi \rangle, E)$
end

Die Funktion translate bestimmt X_1 als die erste unbenutzte Variable und erstellt die Gleichung $X_1 = \Phi$, wobei Φ die Eingabeformel ist. Danach wird die Funktion make_eqs mit dem leeren Gleichungssystem E ausgeführt. In der Funktion make_eqs wird dann die Formel in Teilformeln zerlegt, die jeweils einer unbenutzten Variable zugeordnet werden. Durch rekursive Ausführung der Funktion make_eqs und wegen des Hochzählens des Variablenindexes i beim Hinzufügen einer Gleichung bleibt X_i stets die erste unbenutzte Variable.

Die Reihenfolge der Nummerierung von Variablen entspricht, wie eine Tiefensuche auf dem Syntaxbaum der Formel durchgeführt wird. Und zwar wird eine Gleichung erstellt, wenn ein Operator oder eine Proposition auf dem Syntaxbaum der Formel bei der Tiefensuche besucht wird. Auf der linken Seite der Gleichung wird eine unbenutzte Variable hinzugefügt, deren Index möglichst niedrig nummeriert wird. Auf der rechten Seite der Gleichung steht eine Formel in der Form, dass die Operanden mit Variablen ersetzt werden. Wenn eine Proposition auf dem Syntaxbaum besucht wird, wird eine Gleichung gestellt, in der die Proposition an eine neue Variable zugewiesen wird. Wenn eine Variable betrachtet wird, wird keine neue Gleichung gestellt.

Die Variablen auf der rechten Seite erhalten somit einen höheren Index als die Variable auf der linken Seite in jeder Gleichung, solange sie nicht mit einem Fixpunktoperator gebunden sind. Für die Variablen, die mit einem Fixpunktoperator

gebunden sind, ist es wichtig, eine einheitliche Variablenordnung einzuhalten, da ein Gleichungssystem je nach Nummerierung der Variablen verschiedene Bedeutungen haben kann. Das Gleichungssystem $[\langle X_i =_\nu [a]X_i \wedge X_j\rangle, \langle X_j =_\mu [a]X_j \vee X_i\rangle]$ ist semantisch äquivalent mit $\nu X_i.([a]X_i \wedge \mu X_j([a]X_j \vee X_i))$, falls $i < j$ gilt. Für $i > j$ bedeutet das Gleichungssystem jedoch $\mu X_j.([a]X_j \vee \nu X_i([a]X_i \wedge X_j))$.

Im Folgenden wird die Vorgehensweise des Model-Checking-Algorithmus [CKS92] erläutert, in dem ein Gleichungssystem von der zu untersuchenden Formel erstellt und die Abhängigkeit zwischen den Gleichungen analysiert wird.

Zunächst wird ein gerichteter Graph erstellt, dessen Knoten die Variablen des Gleichungssystems sind und dessen Kanten die Abhängigkeit zur Bewertung der Variablen darstellen. Von dem Knoten einer Variable, die auf der rechten Seite einer Gleichung auftritt, wird eine Kante zum Knoten der Variable auf der linken Seite dieselber Gleichung hinzugefügt. Die Kanten werden jeweils mit einem Operator $\vee, \wedge, [\,], \langle\,\rangle, \varepsilon$ beschriftet, der in der Formel auf der rechten Seite der Gleichung vorkommt. Die Beschriftung ε mit $X_i \xrightarrow{\varepsilon} X_j$ wird verwendet, wenn die Formel auf der rechten Seite keinen Operator enthält, d.h. wenn eine Gleichung $X_j \lhd\rhd_i X_i$ in dem Gleichungssystem existiert.

Dann werden die stark zusammenhängenden Komponenten des Graphen ausgerechnet. Die Variablen der Knoten in einer stark zusammenhängenden Komponente beeinflussen sich gegenseitig bei ihre Bewertung bzgl. Zustände des Modells. Es ist verständlich, da solche Variablen sowie Gleichungen die Teilformeln einer Fixpunktformel darstellen. Betrachtet man jede stark zusammenhängende Komponente als einen Block, dann erhält man einen gerichteten azyklischen Graphen, dessen Knoten Blöcke sind. Diese Blöcke werden topologisch sortiert, so dass eine Reihenfolge bestimmt wird, welcher Block zuerst bearbeitet werden soll.

Enthält eine stark zusammenhängende Komponente zwei Knoten von Fixpunktvariablen für unterschiedliche Fixpunktoperationen, dann wird die Belegung der tiefer liegenden Variablen via Backtracking bestimmt, so dass sie erneut initialisiert und berechnet wird. Das ist der Hauptgrund der exponentiellen Laufzeitkomplexität. Um den Exponent möglichst klein zu halten, werden die Abhängigkeiten zwischen den Variablen analysiert, so dass eine effektive Reihenfolge zur Bestimmung der Belegung von Variablen bestimmt wird. Außerdem wird festgestellt, welche Variable für die weitere Berechnung neu initialisiert werden muss, wenn die Belegung einer Fixpunktvariable geändert wird. Somit bleibt die unnötige Neuini-

tialisierung sowie Berechnung der Variablenbelegung gespart. Die Laufzeitkomplexität des Model-Checking-Algorithmus in [CKS92] hängt dadurch von reduzierter Alternierungstiefe der Formel ab.

Die Belegung der Variablen wird in einer Matrix gespeichert, dessen Einträge jedoch keine Wahrheitswerte wie in Abb. 3.3 sondern Zählerstände sind. In (X_i, m) wird beispielsweise die Anzahl der Einträge gespeichert, die mit ihrem Wahrheitswert für die Bewertung von (X_i, m) direkt beeinflussen. Wenn der Wahrheitswert eines Eintrags geändert wird, wird dies an den Zählerstand der anderen Einträge weitergeleitet. Somit wird vermieden, dass der Einfluss einer Bewertung auf den anderen Einträgen unnötig mehrmals betrachtet wird. Um die Abhängigkeiten zwischen den Einträgen effektiv zu bestimmen und zu verwalten, kann man einen Graphen erstellen, der in dem nächsten Abschnitt vorgestellt wird.

3.4 Abhängigkeitsgraphen

In diesem Abschnitt definieren wir zunächst einen Graphen, so dass die Veränderung der Bewertungen von Zuständen bzgl. Variablen leicht bestimmt und ihr Einfluss auf den anderen Bewertungen einfach verwaltet werden kann.

Für die Beschreibung der Abhängigkeitsgraphen verwenden wir eine Relation $\leq$ über die Menge der Formeln, die wie folgt definiert wird: Mit $\phi \leq \psi$ bezeichnen wir, dass die Formel ϕ eine Teilformel der Formel ψ ist. Üblicherweise wird $\not\leq$ für die Negation von $\leq$ verwendet. Es ist offensichtlich, dass die Relation $\leq$ eine partielle Ordnung ist. Wird die $\leq$-Relation auf der Menge von Teilformeln einer Formel Φ erstellt, dann ist die Formel Φ das einzige maximale Element.

Definition 3.4.1 (*Abhängigkeitsgraphen bzgl. Formel*)
Sei $M = (S, Act, AP, \rightarrow, L)$ ein Modell und Φ eine Formel.
Ein Abhängigkeitsgraph ist ein gerichteter Graph $G = (V, E)$, wobei

- *V eine endliche Menge von Knoten mit $V := \{(\phi, s) \mid \phi \leq \Phi$ und $s \in S\}$*
- *E eine endliche Menge von Kanten ist, die zwei Knoten wie folgt verbinden: Für alle $\phi \leq \Phi$ und $s \in S$*
  - *$\phi = \phi_1 \wedge \phi_2 \Rightarrow ((\phi_1, s), (\phi, s)) \in E$ und $((\phi_2, s), (\phi, s)) \in E$*
  - *$\phi = \phi_1 \vee \phi_2 \Rightarrow ((\phi_1, s), (\phi, s)) \in E$ und $((\phi_2, s), (\phi, s)) \in E$*

- $\phi = \mu X.\psi \Rightarrow ((\psi, s), (\phi, s)) \in E$
- $\phi = \nu X.\psi \Rightarrow (((\psi, s), (\phi, s)) \in E$
- $\phi = [a]\psi \Rightarrow ((\psi, t), (\phi, s)) \in E$ *für jedes* $(s, a, t) \in \rightarrow$ *des Modells* M
- $\phi = \langle a \rangle \psi \Rightarrow ((\psi, t), (\phi, s)) \in E$ *für jedes* $(s, a, t) \in \rightarrow$ *des Modells* M
- $\phi = X \Rightarrow ((\psi, s), (\phi, s)) \in E$ *für* $\psi = \sigma X.\psi'$ *mit* $\sigma \in \{\mu, \nu\}$

Die Richtung der Kanten zeigt, wie die Bewertung der Knoten auf dem anderen beeinflusst. Beispielsweise werden die Bewertungen von Teilformeln (ϕ_1, s) und (ϕ_2, s) zu der Formel $(\phi_1 \wedge \phi_2, s)$ übertragen, da $((s \models \phi_1) \wedge (s \models \phi_2)) \Rightarrow (s \models \phi_1 \wedge \phi_2)$ gilt. Die Kante $((\mu X.\phi, s), (X, s))$ lässt sich so erklären, dass die Fixpunktvariable X ihre Bewertung durch Initialisierung oder Iteration bekommt.

Da wir die modalen μ-Kalkül-Formeln in ein Gleichungssystem umwandeln, brauchen wir eine weitere Definition von Abhängigkeitsgraphen bzgl. eines Gleichungssystems.

Definition 3.4.2 (*Abhängigkeitsgraphen bzgl. Gleichungssystem*)
Sei $M = (S, Act, AP, \rightarrow, L)$ *ein Modell und* $[\langle X_1 \triangleleft\!\triangleright_1 \phi_1 \rangle, \ldots, \langle X_m \triangleleft\!\triangleright_m \phi_m \rangle]$ *ein Gleichungssystem. Ein Abhängigkeitsgraph ist ein gerichteter Graph* $G = (V, E)$*, wobei*

- V *eine endliche Menge von Knoten mit* $V := \{(X_i, s) \mid 1 \leq i \leq m$ *und* $s \in S\}$
- E *eine endliche Menge von Kanten ist, die zwei Knoten wie folgt verbinden: Für alle* $s \in S$ *und* $X_i, X_j, X_k \in \{X_1, \ldots, X_m\}$
  - $X_i \triangleleft\!\triangleright_i X_j \wedge X_k \Rightarrow ((X_j, s), (X_i, s)) \in E$ *und* $((X_k, s), (X_i, s)) \in E$
  - $X_i \triangleleft\!\triangleright_i X_j \vee X_k \Rightarrow ((X_j, s), (X_i, s)) \in E$ *und* $((X_k, s), (X_i, s)) \in E$
  - $X_i \triangleleft\!\triangleright_i [a]X_j \Rightarrow ((X_j, t), (X_i, s)) \in E$ *für jedes* $(s, a, t) \in \rightarrow$ *des Modells* M
  - $X_i \triangleleft\!\triangleright_i \langle a \rangle X_j \Rightarrow ((X_j, t), (X_i, s)) \in E$ *für jedes* $(s, a, t) \in \rightarrow$ *des Modells* M
  - $X_i \triangleleft\!\triangleright_i X_j \Rightarrow ((X_j, s), (X_i, s)) \in E$

Es ist zu beachten, dass die Richtung der Kanten so bestimmt ist, dass die auf der rechten Seite einer Gleichung gewonnene Bewertung auf die linke Seite der Gleichung übertragen wird. Das Prinzip der Kantenrichtung bleibt also gleich, dass immer die Teilformeln zuerst und danach die aus den Teilformeln bestehende Formel bewertet.

Ein wesentlicher Unterschied zu dem Abhängigkeitsgraphen mit einer Formel liegt darin, dass die Fixpunktformeln hierbei nicht als Teilformel vorkommen. Dafür wird die Variablenordnung berücksichtigt. Wenn eine Gleichung $X_i \lhd\rhd_i \phi_i$ eine Variable X_j mit $i \geq j$ enthält, dann ist X_j eine Fixpunktvariable der Fixpunktformel, die durch die Gleichung $X_j \lhd\rhd_j \phi_j$ dargestellt wird.

Wir betrachten das Beispiel 3.2.3 wieder und erstellen einen Abhängigkeitsgraphen für die Gleichungen $X_1 =_\nu X_2, X_2 =_\mu [a]X_3, X_3 =_\mu X_4 \vee X_2, X_4 =_\mu X_5 \wedge X_1$ sowie $X_5 =_\mu A$, die im Beispiel 3.3.2 erzeugt wurden.

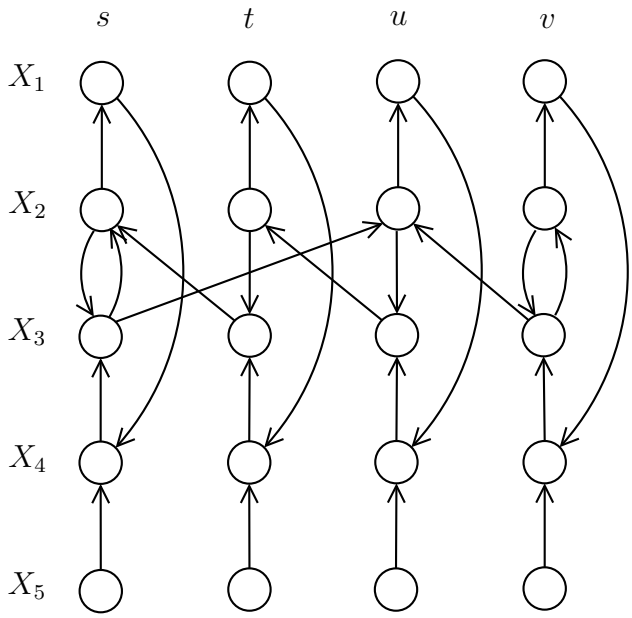

Abbildung 3.4: Abhängigkeitsgraph

Mit Hilfe von Abhängigkeitsgraphen ist leicht festzustellen, zwischen welchen Knoten eine Abhängigkeit im Bezug auf ihre Bewertung besteht. In Abb. 3.3 wurde 1 bzw. 0 als Matrixeintrag benutzt, so dass er genau dem Wahrheitswert tt bzw. ff der Knotenbewertung entspricht. Es wurde dabei keinen Unterschied gemacht, ob die dazugehörige Variable zum ν- oder μ-Block gehört.

Zur Bestimmung der Variablenbelegung erstellen wir diesmal zwar eine Matrix ähnlich wie in Abb. 3.3, deren Einträge jedoch jeweils ein Zählerstand und kein Wahrheitswert sind. In dem Feld von (X_i, m) wird die Anzahl der Knoten gespeichert, die mit ihrem Wahrheitswert die Bewertung des Knotens von (X_i, m) beeinflussen. Die Vorgängerknoten im Abhängigkeitsgraphen gehören zu den solchen. Dabei wird

die Zugehörigkeit der Variablen zum ν- bzw. μ-Block sowie der Operator auf der rechten Seite der Gleichung berücksichtigt.

Wenn ein Knoten (X_i, m) mit dem Wahrheitswert δ initialisiert wird, zeigt der Zählerstand von (X_i, m), wie viele Vorgängerknoten ihren Wahrheitswert von δ zu $\overline{\delta}$ wechseln müssen, damit der Knoten (X_i, m) den Wahrheitswert $\overline{\delta}$ erhält. Jedesmal wenn ein Vorgängerknoten seinen Wahrheitswert von δ zu $\overline{\delta}$ wechselt, wird der Zählerstand um 1 vermindert. Die Knoten behalten ihre Initialbewertung, solange ihrer Zählerstand größer als 0 ist. Auf diese Weise stellt jeder Zählerstand gleichzeitig auch einen Wahrheitswert dar.

Im Folgenden wird die Berechnung der Variablenbelegung unter Benutzung eines Zählers tabellarisch dargestellt. Die ausführliche Vorgehensweise dieser Berechnung wurde in [CKS92] vorgestellt.

	s	t	u	v
X_1	1	1	1	1
X_2	2	1	2	1
X_3	1	1	1	1
X_4	2	1	1	1

	s	t	u	v
X_1	1	1	1	1
X_2	1	0	1	0
X_3	1	0	0	0
X_4	1	0	0	0

	s	t	u	v
X_1	0	1	0	1
X_2	2	1	2	1
X_3	1	1	1	1
X_4	2	1	1	1

	s	t	u	v
X_1	0	1	0	1
X_2	1	1	1	0
X_3	1	0	1	0
X_4	2	0	1	0

	s	t	u	v
X_1	0	0	0	1
X_2	2	1	2	1
X_3	1	1	1	1
X_4	2	1	1	1

	s	t	u	v
X_1	0	0	0	1
X_2	2	1	1	0
X_3	1	1	1	0
X_4	2	1	1	0

Abbildung 3.5: Diese Tabellen beinhalten die Veränderung der Zählerstände.

Es ist zu beachten, dass $[X_1]$ einen ν-Block und $[X_2, X_3, X_4, X_5]$ einen μ-Block bildet. X_5 wird dabei ausgelassen, da die Knotenbewertungen für diese Variable konstant bleiben. Die Knoten in einem ν-Block werden mit dem Wahrheitswert tt und die Knoten in einem μ-Block mit ff initialisiert, wenn sie keinen konstanten Wahrheitswert haben.

Der Wahrheitswert des Knotens (X_2, s) wird beispielsweise mit ff initialisiert, da X_2 zum μ-Block gehört. Die rechte Seite der Gleichung $X_2 =_\mu [a]X_3$ enthält den

[ ]-Operator. Für den Zählerstand von (X_2, s) werden deshalb alle Vorgängerknoten des Knotens (X_2, s) gezählt, die mit dem Wahrheitswert ff initialisiert werden. Am Anfang beträgt der Zählerstand 2 und wird dann um 1 vermindert, wenn der Wahrheitswert des Vorgängerknotens (X_3, u) bzw. (X_3, t) von ff zu tt geändert wird. Erreicht der Zählerstand von (X_2, s) den Wert 0, dann bedeutet dies, dass die beiden Knoten (X_3, u) und (X_3, t) bereits den Wahrheitswert tt erhalten haben und der Knoten (X_2, s) den Wahrheitswert tt erhält.

Der Zählerstand von (X_3, s) wird hingegen mit 1 initialisiert, obwohl der Knoten (X_3, s) zwei Vorgängerknoten hat. Das kommt daher, dass die Formel auf der rechten Seite der Gleichung $X_3 =_\mu X_4 \vee X_2$ den $\vee$-Operator hat. Sobald ein Vorgängerknoten (X_2, s) oder (X_4, s) den Wert tt hat, erhält der Knoten (X_3, s) den Zählerstand 0, d.h. den Wahrheitswert tt.

In der ersten Tabelle befinden sich die initialen Zählerstände, wobei die Zugehörigkeit der Variablen zum ν- bzw. μ-Block sowie der Operator auf der rechten Seite der Gleichung berücksichtigt wird. In die Zählerstände für die Variable X_4 werden die konstanten Wahrheitswerte für X_5 bereits einbezogen, so dass die Einträge $(2, 1, 1, 1)$ anstatt $(2, 2, 2, 2)$ betragen.

In der zweiten Tabelle werden zunächst die initialen Wahrheitswerte (tt, tt, tt, tt) des ν-Blocks $[X_1]$ in die Zählerstände des μ-Blocks $[X_2, X_3, X_4]$ integriert, so dass die Zählerstände für X_4 $(1, 0, 0, 0)$ betragen. Dann werden die Einflüsse der geänderten Wahrheitswerte von (X_4, t), (X_4, u) und (X_4, v) auf den restlichen Knoten im μ-Block wirken lassen. Wird ein Wahrheitswert dadurch geändert, dann werden die Zählerstände in dem μ-Block dementsprechend weitergehend angepasst, bis sie stabilisiert werden.

In der dritten Tabelle werden die Zählerstände des ν-Blocks $[X_1]$ nach dem Ergebnis des μ-Blocks der zweiten Tabelle korrigiert, so dass der Zählerstand von (X_1, s) sowie von (X_1, u) um 1 vermindert wird, da der Zählerstand von (X_2, s) und (X_1, u) größer als 0 und somit ihrer Wahrheitswert ff ist. Die Zählerstände des μ-Blocks werden dann erneut initialisiert. Die Bestimmung der Zählerstände wird auf diese Weise fortgesetzt, bis alle Zählerstände des ν- sowie μ-Blocks stabilisiert werden.

Durch Erstellung eines Abhängigkeitsgraphen und Benutzung eines Zählers können die Einflüsse der Knotenbewertung auf den anderen Knoten effizient weitergeleitet werden. In [CKS92] und [Kl94] wurde ein Model-Checking-Algorithmus vorgestellt, dessen Datenstruktur zu diesem Zweck geeignet gewählt wurde. Alle Knoten werden

durch topologische Sortierung selektiert, so dass die Knoten nach einer Berechnungsreihenfolge gruppiert werden, wobei jede Gruppe möglichst viele Knoten enthält, deren Bewertungen in einer Iteration berechnet werden können. In jedem Knoten wird dann ein Zähler eingebaut, um zu ermitteln, wie viele Vorgängerknoten eine bestimmte Bewertung noch erhalten müssen, damit die Knotenbewertung geändert wird. Wenn der Zähler eines Knotens den Wert 0 zur Änderung der Bewertung erreicht, wird der Knoten in die Warteschlange aufgenommen. Die Knoten werden aus der Warteschlange nach und nach herausgenommen. Die Bewertungsänderung des Knotens aus der Warteschlange wird dann zu den nachfolgenden Knoten übertragen. Dieser Vorgang wird solange wiederholt, bis kein Knoten mehr in der Warteschlange vorhanden ist. Durch geschickte Verwendungen einer Abhängigkeitsgraphen sowie der Warteschlange erreicht der Model-Checking-Algorithmus in [CKS92] die Laufzeitkomplexität

$$\mathcal{O}\left(|M| \cdot |E| \cdot \left(\frac{|S| \cdot |E|}{rt(E)}\right)^{rt(E)-1}\right)$$

wobei $|M|$ bzw. $|E|$ die Größe des Modells bzw. des Gleichungssystems, $|S|$ die Anzahl der Zustände im Modell und $rt(E)$ die reduzierte Alternierungstiefe des Gleichungssystems ist. Eine nähere Beschreibung über die Implementierung des Algorithmus ist in [Kl94] und [CKS92] zu finden.

In [LBCJM94] wurde ein Model-Checking-Algorithmus vorgestellt, in dem die Monotonieeigenschaft weitgehend ausgenutzt wird, so dass die Laufzeitkomplexität $\mathcal{O}(|M| \cdot |\Phi| \cdot (|S| \cdot |\Phi|)^{\lceil at(\Phi)/2 \rceil})$ beträgt. Der Speicherbedarf des Algorithmus kann leider sehr gross $\mathcal{O}((|S| \cdot |\Phi|)^{\lceil at(\Phi)/2 \rceil})$ sein.

Mit einem Model-Checking-Algorithmus kann festgestellt werden, ob die Eingabeformel in allen Zuständen eines Modells erfüllt ist. Das Ergebnis des Model-Checkings zeigt also normalerweise, in welchen Zuständen die Formel gilt bzw. nicht gilt, aber keine Begründung dafür. Wenn die Begründung des Ergebnisses von Interesse ist, kann ein Model-Checking-Algorithmus eingesetzt werden, mit dem eine Beweisstruktur aufgebaut wird. Wenn eine Formel Φ in einem Zustand s nicht erfüllt ist, wird ein Gegenbeispiel mit einem möglichen fehlerhaften Ablauf des Modells bzgl. der Formel Φ konstruiert, um die Fehlersuche zu erleichtern. Ein Gegenbeispiel zu $s \models \Phi$ ist nichts anderes als ein Beweisablauf für $s \models \neg\Phi$. In dem nächsten Abschnitt betrachten wir ein solches Verfahren.

3.5 Tableausystem

In diesem Abschnitt betrachten wir eine Beweisstruktur, mit der die Erfüllbarkeit der μ-Kalkül-Formeln im Modell verifiziert werden kann. Das Beweisverfahren wurde in [StWa89] für das lokale Model-Checking vorgestellt.

Um $s \models \Phi$ zu zeigen, kann man eine Art Beweisbaum aufbauen, der dies beweist. Ein Beweisbaum entsteht durch Anwendung von so genannten Beweisregeln. Dabei wird mit der zu beweisenden Aussage $s \models \Phi$ angefangen. Die Beweisregeln lassen sich auf folgende Weise motivieren: Soll $s \models P$ oder $s \models \neg P$ für eine Proposition P gezeigt werden, so braucht man nur die Definition der Markierungsfunktion L des Modells $M = (S, Act, AP, \rightarrow, L)$ zu betrachten. Soll $s \models \phi \wedge \psi$ gezeigt werden, so muss aufgrund der Semantik sowohl $s \models \phi$ als auch $s \models \psi$ gezeigt werden.

Für den Beweis von $s \models \Phi$ untersucht man die Berechnungsfolgen der Formel im Bezug auf den erreichten Zuständen im Modell, ob sie mit einer gewissen Eigenschaft enden. Da ein Beweis etwas Syntaktisches ist, wird im Folgenden $s \vdash \Phi$ statt $s \models \Phi$ geschrieben. Der einzige Startknoten (Wurzel) ist $s \vdash \Phi$ und die nachfolgenden Knoten sind jeweils ein nächster Berechnungsschritt. Üblicherweise wird so aufgebaute Beweisstruktur „Tableau“ genannt.

Soll $s \vdash \sigma X.\Phi$ bewiesen werden, so kann dies dadurch gezeigt werden, dass $s \vdash \sigma X.\Phi[\sigma X.\Phi/X]$ bewiesen wird. Da eine Berechnungsfolge für die μ-Kalkül-Formeln unendlich lang werden kann, wird die Berechnungsfolge von $s \vdash \sigma X.\Phi$ auf dem Knoten $s \vdash X$ gestoppt. Die weitere Berechnungsfolge von dem Knoten $s \vdash X$ aus ist bereits bekannt. Mit dem Mechanismus des Ausrollens wird vermieden, dass der Beweisbaum unendlich gross wird.

Die Fixpunktoperatoren können jedoch geschachtelt werden, wie z.B. $\nu X.\mu Y.\Phi$. Für das ν wird eine Berechnungsfolge konstruiert, die sich aber aus Berechnungsfolgen für das μ zusammensetzt. Die einzeln Berechnungsfolgen für das μ werden unabhängig voneinander betrachtet. Aus diesem Grund wird für den Mechanismus des Ausrollen die Syntax des modalen μ-Kalküls um propositionale Konstantensymbole $U, V, \ldots$ erweitert. Beim Ausrollen von $\sigma X.\Phi$ wird $\Phi[\sigma X.\Phi/X]$ durch $\Phi[U/X]$ ersetzt und dies wird in einer Umgebung Δ mit $\Delta(U) = \sigma X.\Phi$ gespeichert. Normalerweise wird die Umgebung in der Form $s \vdash_\Delta \Phi$ mitgeführt. Wenn Δ aber leer oder nicht von Bedeutung ist, lassen wir es weg.

Im Folgenden werden die Beweisregeln definiert, die zeigen, welcher Berechnungs-

schritt möglich ist.

Definition 3.5.1 *(Beweisregeln)*

$R_\wedge \quad \dfrac{s \vdash_\Delta \phi \wedge \psi}{s \vdash_\Delta \phi \quad s \vdash_\Delta \psi}$

$R_{\vee_1} \quad \dfrac{s \vdash_\Delta \phi \vee \psi}{s \vdash_\Delta \phi}$

$R_{\vee_2} \quad \dfrac{s \vdash_\Delta \phi \vee \psi}{s \vdash_\Delta \psi}$

$R_{\langle\rangle} \quad \dfrac{s \vdash_\Delta \langle a \rangle \phi}{s' \vdash_\Delta \phi}$ *für* $s \xrightarrow{a} s'$

$R_{[]} \quad \dfrac{s \vdash_\Delta [a]\phi}{s_1 \vdash_\Delta \phi \ldots s_n \vdash_\Delta \phi}$ *für* $\{s_1, \ldots, s_n\} = \{s' \mid s \xrightarrow{a} s'\}$

$R_\sigma \quad \dfrac{s \vdash_\Delta \sigma X.\phi}{s \vdash_{\Delta'} U}$ *, wobei* $\Delta' = \Delta + [U \mapsto \sigma X.\phi]$ *und* U *von allen Konstantensymbolen in* Δ *verschieden ist.*

$R_U \quad \dfrac{s \vdash_\Delta U}{s \vdash_\Delta \phi[U/X]}$ *, wobei* $\Delta(U) = \sigma X.\phi$ *und es keinen Knoten* $s \vdash_{\Delta'} U$ *oberhalb* $s \vdash_\Delta U$ *gibt.*

Definition 3.5.2 *(Tableau, erfolgreich)*
Ein Tableau für $s \vdash \Phi$ *ist ein maximaler Beweisbaum, dessen Wurzel* $s \vdash \Phi$ *ist. Die unmittelbaren Nachfolger eines Blattes werden durch Anwendung einer der Tableau-Regeln bestimmt. Maximalität bedeutet dabei, dass keine Regel auf den Blättern anwendbar ist.*

Ein Tableau ist genau dann erfolgreich für $s \vdash \Phi$*, wenn es endlich ist und jedes Blatt* $t \vdash_\Delta \phi$ *eine der folgenden Bedingungen erfüllt:*

- $\phi = P$ *und* $P \in L(t)$
- $\phi = \neg P$ *und* $P \notin L(t)$
- $\phi = [a]\psi$
- $\phi = U$ *und* $\Delta(U) = \nu X.\psi$

Um zu prüfen, ob $s \models \Phi$ gilt, wird ein Tableau aufgebaut. Ist ein Tableau für $s \vdash \Phi$ erfolgreich, so gilt $s \models \Phi$. Wenn das Modell endlich ist, dann gilt auch die Umkehrung [StWa89].

Das Tableau-Verfahren kann als eine Art operationale Semantik der Fixpunktoperatoren beim μ-Kalkül angesehen werden. Die Fixpunkte werden ausgerollt und die Erfolgsbedingungen an ein Tableau sind so, dass es im Fall des kleinsten Fixpunktes keinen Zyklus im Tableau geben darf. In [StWa89, Cl90] wurde ein Tableau-Verfahren zum lokalen Model-Checking des modalen μ-Kalküls vorgestellt. Das Tableau-Verfahren kann ebenfalls für das globale Model-Checking angewendet werden ([Ki96]).

Die Erstellung eines Tableaus kann hilfreich sein, zu verstehen, warum $s \models \Phi$ gilt oder nicht gilt. Ein Beweisablauf bzw. ein Gegenbeispiel wird dabei automatisch generiert. Ein Nachteil bleibt jedoch, dass der Benutzer keine Auswahlmöglichkeit hat, welche Berechnungsfolge verfolgt werden soll. Wenn die Fixpunktoperatoren geschachtelt in der Formel auftreten, kann es vorkommen, dass mehrere erfolgreiche Tableaus konstruierbar sind. In [Ki96] wurde zwar eine interaktive Konstruktion eines Tableaus für den Fall vorgeschlagen, in welche Richtung der untersuchende Ablauf verfolgt werden soll. Jedoch wird solche Konstruktion nach der Berechnung des Model-Checkings ausgeführt, so dass eine zusätzliche Laufzeit dafür benötigt wird.

In diesem Kapitel wurde der Fixpunktsatz von Knaster-Tarski sowie von Kleene vorgestellt und dann ein Model-Checking-Algorithmus gezeigt, der auf dem Fixpunktsatz basiert. Danach wurde ein Gleichungssystem sowie ein Abhängigkeitsgraph definiert. Dann wurde ein Model-Checking-Algorithmus mit einer verbesserte Laufzeitkomplexität vorgestellt, in dem die beiden Datenstrukturen effektiv benutzt werden. Zum Schluss haben wir ein Tableau-Verfahren betrachtet, in dem eine Beweisstruktur aufgebaut wird.

Wir werden in dem nächsten Kapitel einen neuen Algorithmus entwerfen, mit dem das Model-Checking-Problem für modalen μ-Kalkül gelöst wird. Das Model-Checking-Problem wird dabei spieltheoretisch interpretiert. Wir werden dafür einen Graphen erstellen, der ähnlich wie eine Beweisstruktur aufgebaut wird. Der Graph enthält jedoch die Informationen, warum $s \models \phi$ bzw. $s \not\models \phi$ gilt, und zwar für jeden Zustand s des Modells und jede Teilformel ϕ. Nach Interesse des Benutzers kann dann ein Beweisablauf interaktiv konstruiert werden, indem die dazugehörigen Informationen von dem Graphen abgelesen werden. Die Basisdatenstruktur sowie Berechnungsabfolge unseres Algorithmus wird von dem Model-Checking-Algorithmus [CKS92] hergeleitet, so dass die Vorteile von dem beibehalten werden.

Wir werden zunächst die grundlegenden Begriffe und Definitionen über das Zwei-Personen-Spiel einführen und dann ein Spiel-Algorithmus entwerfen, der für den Einsatz zum Model-Checking gut geeignet ist.

Kapitel 4

Spielbasiertes Model-Checking

Wir entwickeln in diesem Kapitel einen neuen spielbasierten Algorithmus, mit dem wir das globale Model-Checking-Problem für modalen Kalkül lösen können. Das Model-Checking-Problem wird dabei spieltheoretisch interpretiert. Unser Algorithmus nimmt den im vorherigen Kapitel definierten Abhängigkeitsgraphen als Eingabe und berechnet nicht nur die Erfüllbarkeit der Teilformeln in den Zuständen sondern auch ihre Begründung dazu. Wir werden die Monotonieeigenschaft der Fixpunktoperatoren so ausnutzen, dass wir viele Zwischenergebnisse der Berechnung beibehalten. Zu den Zwischenergebnissen gehören die Bewertung der Knoten und die Begründung. Am Ende erhalten wir eine korrekte Begründung der Bewertungen aller Knoten.

4.1 Begriffe und Definitionen über Spiele

In diesem Abschnitt werden die grundlegenden Begriffe und Definitionen eingeführt, die für den Entwurf unseres Algorithmus benötigt werden. Zunächst werden die Spiele auf endlichen Graphen definiert, die Zwei-Personen-Spiel oder Paritätsspiel genannt werden, und dann die leicht erkennbaren Eigenschaften betrachtet.

Definition 4.1.1 *(Spielgraph)*
Ein Spielgraph $G = (V_0, V_1, E, p)$ ist ein endlicher gerichteter Graph, wobei

- *V eine Knotenmenge,*
- *V_0 und V_1 disjunkte Teilmengen von V mit $V_0 \cup V_1 = V$,*

- $E \subseteq V \times V$ *eine Kantenmenge und*
- $p : V \to \mathbb{N}$ *eine Prioritätsfunktion, wobei die höchste(kleinste) sowie niedrigste(größte) Priorität für jeden Spielgraphen vorbestimmt ist.*

Bemerkung: In dieser Arbeit wird angenommen, dass die Priorität lückenlos definiert wird. Die höchste Priorität beginnt mit 0 bzw. 1 und es gibt keinen Knoten v mit $p(v) = m \geq 2$, wenn kein Knoten u mit $p(u) = (m-1)$ existiert. Dies ist keine echte Einschränkung und kann durch Umordnung der Priorität einfach erreicht werden.

Werden mehrere Spielgraphen $G_1, G_2, \ldots$ gleichzeitig behandelt, dann werden die zugehörigen Knotenmengen bzw. die Kantenmengen mit $V_{G_1}, V_{G_2}, \ldots$ bzw. mit $E_{G_1}, E_{G_2}, \ldots$ bezeichnet.

Definition 4.1.2 *(Spiel, Spielregel)*
Ein Spiel besteht aus einem Spielgraphen $G = (V_0, V_1, E, p)$ mit einem Startknoten $v \in V$ und wird von zwei Spielern (0 und 1) auf folgende Weise gespielt:

Der Spieler $i \in \{0, 1\}$ wählt von dem aktuellen Knoten v den nächsten Knoten w mit $(v, w) \in E$ aus, wenn $v \in V_i$ gilt.

Beim Spielen wird also ein Pfad nach Auswahl der nächsten Knoten von den beiden Spielern aufgebaut. Diesen Pfad nennen wir Spiellauf.

Definition 4.1.3 *(Spiellauf)*
Ein Spiellauf $v_0 v_1 v_2 \cdots$ ist ein endlich oder unendlich langer Pfad in einem Spielgraphen, wobei v_0 der Startknoten ist.

Für jeden aktuellen Knoten $v \in V_i$ eines Spiellaufes wählt der Spieler i einen Nachfolgerknoten, so dass der Spiellauf fortgesetzt wird. Ein Spiel endet somit nur dann mit einem endlich langen Spiellauf, wenn der letzte Knoten keinen Nachfolgerknoten hat. Ansonsten wird das Spiel unendlich weiter gespielt, so dass ein unendlich langer Spiellauf entsteht. Wenn ein Spielgraph Schleifen enthält, sind unendlich viele unendliche Spielläufe konstruierbar. Ein Spiellauf wird auch eine Partie genannt.

Wie bei jedem Spiel sind die Gewinnbedingungen ein wichtiger Bestandteil der Spiele, damit die Spielläufe sinnvoll bewertet werden können. Zur Erklärung der Gewinnbedingungen werden die folgenden Bezeichnungen verwendet:

- Mit π_i wird der i-te Knoten auf dem Pfad π und mit $|\pi|$ die Länge des Pfades bezeichnet, d.h. für $\pi = v_0v_1v_2v_3v_4$ gilt z.B. $\pi_0 = v_0$, $\pi_1 = v_1$ und $|\pi| = 4$.
- Mit $Inf(\pi)$ wird die Menge der Knoten bezeichnet, die auf einem Pfad π unendlich oft auftreten, d.h.

$$Inf(\pi) := \{v \mid \textit{Für alle } k \in \mathbb{N} \text{ existiert ein } i > k \text{ mit } \pi_i = v\}.$$

- Mit $minInf(\pi)$ wird der kleinste Funktionswert $p(v)$ von den Knoten $v \in Inf(\pi)$ bezeichnet, d.h.

$$minInf(\pi) := min\{p(v) \mid v \in Inf(\pi)\}.$$

Oft wird auch $maxInf(\pi)$ anstatt $minInf(\pi)$ definiert, so dass

$$maxInf(\pi) := max\{p(v) \mid v \in Inf(\pi)\}.$$

Welche Funktion man braucht, hängt lediglich davon ab, wie der Spielgraph sowie die Prioritätsfunktion p definiert wird und vor allem wie die Gewinnbedingung aussieht. Die Gewinnbedingungen, die in dieser Arbeit benutzt werden, sind wie folgt definiert:

Definition 4.1.4 *(Gewinnbedingungen)*

- *Wenn ein Spiel mit einem endlich langen Spiellauf π endet, wobei $\pi_{|\pi|} \in V_i$ mit $i \in \{0, 1\}$ gilt, dann gewinnt der Spieler $(1 - i)$ das Spiel. Der Knoten $\pi_{|\pi|}$ hat in dem Fall keinen Nachfolgerknoten. Ansonsten hätte der Spieler i nach den Spielregeln weiter gespielt.*
- *Wenn ein unendliches Spiel mit einem Spiellauf π stattfindet, d.h. $|\pi| = \infty$, dann gewinnt der Spieler i das Spiel, wobei $i = (minInf(\pi) \bmod 2)$.*

Die Gewinnbedingungen unterscheiden sich je nachdem, ob ein Spiel mit einer endlichen Länge beendet oder unendlich weiter gespielt wird. Die Gewinnbedingung für einen endlich langen Spiellauf kann man weglassen, wenn der Spielgraphen so geändert wird, dass alle Spiele unendlich lang werden. Man kann z.B. zwei zusätzliche Knoten $z_0 \in V_0$ und $z_1 \in V_1$ mit der Kanten (z_0, z_0) und (z_1, z_1) hinzufügen, so dass der Spieler i von dem Knoten z_i aus das Spiel gewinnt. Dann wird für jede Senke $v \in V_0$ eine Kante (v, z_1) zu z_1 hinzugefügt. Jede Senke $v \in V_1$ erhält ebenfalls

eine ausgehende Kante (v, z_0). Wir lassen jedoch die endlichen Spiele zu, da dies unserer Intuition nahe liegt.

Wir werden die Abhängigkeitsgraphen von dem letzten Kapitel nach einer kleinen Modifikation als Spielgraphen verwenden. Im Lauf dieses Kapitels wird erläutert, wie das Model-Checking-Problem durch ein Spiel-Problem erfasst wird. Im Folgenden wird ein Spielgraph zur Illustration angegeben, wie ein Spiel verläuft.

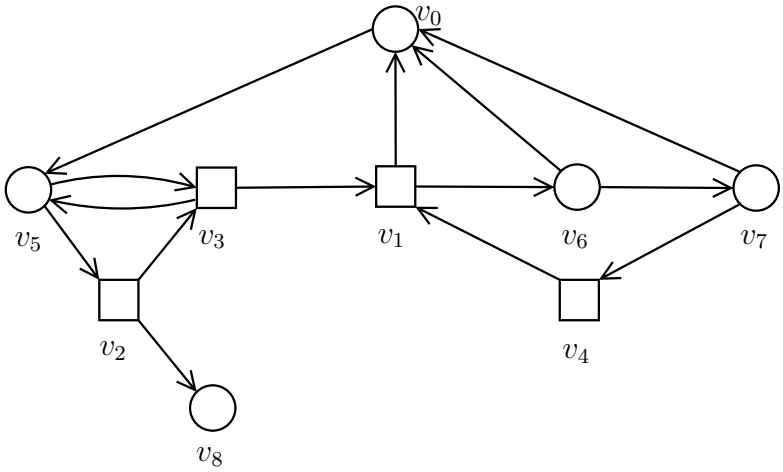

Abbildung 4.1: Spielgraph

Beispiel 4.1.5 *Ein Spielgraph $G = (V_0, V_1, E, p)$ sei wie Abb. 4.1 gegeben, wobei $V_0 = \{v_0, v_5, v_6, v_7, v_8\}$, $V_1 = \{v_1, v_2, v_3, v_4\}$ und $p(v_k) = k$ für alle $k \in \{0, \cdots, 8\}$ gelten.*

In Abb. 4.1 werden die Knoten $v \in V_0$ des Spielers 0 als Kreise dargestellt und die Knoten $v \in V_1$ des Spielers 1 als Quadrate. Der Knoten v_8 hat keinen Nachfolgerknoten. Wenn eine Partie mit dem Knoten v_8 beginnt, gewinnt der Spieler 1 sofort das Spiel nach der ersten Gewinnbedingung in Definition 4.1.4.

Wenn der Gewinner auf einem Knoten bestimmt ist, beeinflusst dies die Partien, die mit einem anderen Knoten beginnen. Beispielsweise kann der Spieler 1 von dem Knoten v_2 aus das Spiel gewinnen, indem er v_8 als nächsten Knoten wählt. Dies führt dazu, dass der Spieler 0 von dem Knoten v_5 aus nicht den Knoten v_2 als nächsten Knoten wählen soll. Also bleibt der Knoten v_3 als Alternative. Wenn der Spieler 1 von v_3 aus den Knoten v_5 als nächsten Knoten wählt, werden die Knoten v_5 und v_3 wiederholt gespielt und der Spieler 1 gewinnt das Spiel nach der zweiter Gewinnbedingung in Definition 4.1.4 und wegen $minInf(v_3 v_5 v_3 \cdots) = 3$.

Es ist zu beachten, dass der Spieler 1 von dem Knoten v_2 aus nicht den Knoten v_3 wählen soll, obwohl von dem Knoten v_3 aus der Spieler 1 die Spiele gewinnen kann. In dem Fall kann eine Schleife $v_2v_3v_5v_2\cdots$ als Spiellauf entstehen, wodurch der Spieler 0 das Spiel gewinnt. Es ist festzulegen, dass die Spieler die bestimmten Knoten wählen müssen, um Spiele zu gewinnen. Variierend verschiedene Nachfolgerknoten zu wählen oder einfach die Knoten zu wählen, von dem aus der Spieler das Spiel gewinnt, führt also nicht zu einem guten Taktik. Wenn ein Spieler während eines Spiels von einem Knoten aus zwei verschiedene Nachfolgerknoten ausgewählt hat und das Spiel gewinnt, kann er bei der ersten Auswahl schon die zweite Auswahl vorziehen, so dass er das Spiel frühzeitig gewinnt. Das bedeutet, dass die Spieler die nächsten Knoten unabhängig von der Reaktion des Gegenspielers so wählen kann, dass der weitere Spiellauf für ihn am günstigsten wird.

Wie die beiden Spieler spielen, d.h. wie sie die nächsten Knoten wählen, lässt sich durch eine Funktion ϱ darstellen, so dass jeder Funktionswert $\varrho(v)$ ein Nachfolgerknoten von v ist. In jedem Spiellauf kann der nächsten Knoten dann nach dem Funktionswert bestimmt werden. Diese Funktion nennen wir Strategie.

Definition 4.1.6 *(Strategie)*
Eine Funktion $\varrho : U \to V$ heißt Strategie, wenn $(u, \varrho(u)) \in E$ für alle Knoten $u \in U$ mit $U \subseteq V$ gilt, die keine Senke sind.

Durch eine Strategie kann bestimmt werden, wie ein Spiel gespielt wird. Solange der aktuelle Knoten mindestens eine ausgehende Kante besitzt, wird das Spiel gemäß der Strategie fortgesetzt. Die Spiele werden sofort gestoppt, wenn eine Senke erreicht wird. Der Spieler, der von der Senke aus den nächsten Knoten wählen soll, verliert das Spiel nach der Gewinnbedingung von Definition 4.1.4. Deshalb ist der Funktionswert $\varrho(v)$ für die Senke v nicht relevant, weil die Spiele von solchen Knoten v aus sowieso nicht fortgesetzt werden. Es ist zu beachten, dass die Strategien in dieser Arbeit speicherfrei (eng.: memoryless) sind, d.h. sie sind nur von dem letzten Knoten eines Spiellaufes abhängig.

Definition 4.1.7 *(Gewinnstrategie)*
Eine Strategie $\varrho : U \to V$ ist genau dann eine Gewinnstrategie für einen Spieler $i \in \{0, 1\}$, wenn jede Partie, in der ein Knoten $u \in (U \cap V_i)$ besucht wird, von dem Spieler i dadurch gewonnen wird, dass der nächste Knoten stets gemäß der Strategie ϱ gewählt wird, und unabhängig davon, wie der Gegenspieler spielt.

Die obige Definition besagt, dass der Spieler i von jedem Knoten $u \in (U \cap V_i)$ aus immer einen Spiellauf konstruieren kann, so dass er das Spiel gewinnt. Dies geschieht unabhängig davon, wie der andere Spieler $(1-i)$ spielt. Der Spieler i gewinnt aber auch die Partien, die mit einem Knoten $u' \in V_{(1-i)}$ beginnen, für den der Spieler $(1-i)$ keine Gewinnstrategie hat. Das geschieht z.B., wenn u' keinen Nachfolgerknoten hat oder alle Nachfolgerknoten von u' zu $(U \cap V_i)$ gehören.

Definition 4.1.8 *(Gewinnmenge)*
Eine Knotenmenge W ist genau dann eine Gewinnmenge für einen Spieler i, wenn der Spieler i jede Partie gewinnen kann, die mit einem Knoten $v \in W$ beginnt, unabhängig davon, wie der Gegenspieler spielt.

In dieser Arbeit wird die Gewinnstrategie für den Spieler 0 bzw. 1 mit σ bzw. τ bezeichnen. Eine Gewinnmenge für den Spieler 0 bzw. 1 wird außerdem mit W_0 bzw. W_1 bezeichnet. Der folgende Satz besagt, dass die Knotenmenge V für jedes Spiel eindeutig in zwei Gewinnmengen geteilt werden kann. Das heißt, dass die Existenz einer Gewinnstrategie für diese Gewinnmengen garantiert wird.

Satz 4.1.9 *([EmJu91, Mo91])*
Für jedes Spiel gibt es eine eindeutige Partition der zwei Gewinnmengen W_0 und W_1 mit $W_0 \cup W_1 = V$, so dass mindestens eine Gewinnstrategie für die Knotenmenge $(V_0 \cap W_0)$ sowie $(V_1 \cap W_1)$ existiert.

Es ist zu beachten, dass der Definitionsbereich einer Gewinnstrategie keine Teilmenge einer Gewinnmenge sein muss, da keine Einschränkung für die Knoten $v \in (U \backslash V_i)$ definiert wurde. Um zu wissen, ob und wie ein Spieler das Spiel gewinnen kann, wenn es von einem Knoten startet, muss sowohl die Aufteilung der Gewinnmengen als auch eine Gewinnstrategie berechnet werden. Wir werden deshalb in dieser Arbeit Gewinnstrategien σ und τ berechnen, so dass $Def(\sigma)$ bzw. $Def(\tau)$ eine Gewinnmenge für den Spieler 0 bzw. 1 ist, wobei $(Def(\sigma) \cup Def(\tau)) = V$ gilt.

Bei der Bestimmung einer Gewinnstrategie ist schwierig zu entscheiden, welche ausgehende Kante und somit welchen Nachfolgerknoten als nächsten Knoten von jedem aktuellen Knoten ausgewählt werden soll, falls dies nicht trivial zu bestimmen ist. Es ist zu beachten, dass viele unterschiedliche Gewinnstrategien für ein Spiel existieren können. Wir betrachten zunächst einige grundlegende Eigenschaften der Gewinnmengen sowie der Strategien.

Lemma 4.1.10 *Gegeben sei σ bzw. τ eine Gewinnstrategie für den Spieler 0 bzw. 1 mit $\sigma : W_0 \to V$ bzw. $\tau : W_1 \to V$, wobei W_0 bzw. W_1 eine Gewinnmenge für den Spieler 0 bzw. 1 mit $W_0 \cup W_1 = V$ ist. Dann gelten*

- $W_0 \cap W_1 = \emptyset$,
- $v \in W_0 \cap V_0 \Rightarrow \sigma(v) \in W_0$,
- $v \in W_0 \cap V_1 \Rightarrow ((v, w) \in E \Rightarrow w \in W_0)$,
- $v \in W_1 \cap V_1 \Rightarrow \tau(v) \in W_1$ *und*
- $v \in W_1 \cap V_0 \Rightarrow ((v, w) \in E \Rightarrow w \in W_1)$.

Beweis: Es gilt offensichtlich nach der Definition von Gewinnstrategie und Gewinnmenge. □

In diesem Abschnitt wurden Definitionen und Begriffe über Spiele bereitgestellt, die im Lauf dieses Kapitels relevant sind. Dabei haben wir uns nur auf die Spiele eingeschränkt, deren Strategien speicherfrei bestimmt werden können. Es gibt auch weitere Spiele mit anderen Spielregeln und Gewinnbedingungen. In [McN93, Tho95] wurden Spiele vorgestellt, deren Spielgraphen jeweils ω-Automaten sind, deren Eingabeworte unendlich lang sind. Für die Sprachen, die von einem ω-Automaten erkannt werden, werden unterschiedliche Akzeptierbedingungen definiert, die Gewinnbedingungen im Spiel entsprechen. Zu den Akzeptierbedingungen gehören beispielsweise, ob mindestens ein Knoten aus der Endknotenmenge $F \subseteq V$ unendlich oft besucht wird, d.h. $Inf(\pi) \cap F \neq \emptyset$ (Büchi-Bedingung), oder ob die unendlich oft auftretenden Knoten eine Endknotenmenge ist, so dass $Inf(\pi) \in \mathcal{F}$ für die Menge $\mathcal{F} \subseteq 2^V$ der Endknotenmengen gilt (Müller-Bedingung).

In [Tho97] wurden u.a. diese Gewinnbedingungen und ihre Beziehungen detailliert behandelt. In [McN93] findet man einen spielbasierten Algorithmus mit der Müller-Bedingung, mit dem eine Gewinnstrategie durch Rekursion auf kleineren Spielgraphen berechnet wird. Dieser Algorithmus kann nicht direkt für das Model-Checking-Problem mit μ-Kalkül angewendet werden, da die Gewinnbedingung zuvor in einer angemessenen Form angepasst werden muss. Für eine solche Anpassung braucht man bereits einen exponentiellen Laufzeitaufwand, da man alle erlaubte Schleifen in dem Spielgraphen berechnen muss. Der Ansatz des Algorithmus bleibt jedoch interessant. Das Spiel-Problem wird in kleineren Teilproblemen zerlegt und die Lösung der Teilprobleme wird in das ursprüngliche Problem zum Nachprüfen eingesetzt.

In dem folgenden Abschnitt werden wir einen spielbasierten Algorithmus entwerfen, der für das Model-Checking-Problem mit μ-Kalkül direkt eingesetzt werden kann. Der Algorithmus basiert auf dem Model-Checking-Algorithmus in [CKS92] und hat einen ähnlichen Ansatz wie der Algorithmus von [McN93].

4.2 Strategie-Synthese

In diesem Abschnitt wird ein neuer Algorithmus entworfen, mit dem eine Gewinnstrategie berechnet wird, wobei die Spielregeln und Gewinnbedingungen der Spiele wie in dem vorherigen Abschnitt definiert sind. Wir werden zunächst die leichten Fälle und dann den eigentlichen Problemfall behandeln.

Wenn ein Spielgraph keine Schleife enthält, kann eine Gewinnstrategie leicht bestimmt werden. Man selektiert die Knoten ohne Nachfolger, für die man den Gewinner nach der ersten Gewinnbedingung in Definition 4.1.4 sofort bestimmen kann. Danach kann man die restlich verbleibenden Knoten mit einer rückwärtsgerichteten Tiefensuche analysieren. Dabei wird weiter nur die erste Gewinnbedingung angewandt. Wir betrachten jeden Vorgängerknoten v von $(W_0 \cup W_1)$, ob ein Spieler von dem Knoten v aus den Gegenspieler zum Verlieren zwingen kann. Dies ist der Fall, wenn der Knoten $v \in V_i$ eine ausgehende Kante (v, w) mit $w \in W_i$ besitzt, oder wenn der Knoten $v \in V_{1-i}$ nur die ausgehenden Kanten (v, w) mit $w \in W_i$ besitzt. Um solche Knoten zu selektieren, definieren wir im Folgenden eine Funktion, in der eine rückwärtsgerichtete Tiefensuche durchgeführt wird. Diese Funktion werden wir in dem Hauptteil des Algorithmus benutzen, um die Gewinnmenge sowie Gewinnstrategie zu erweitern.

Definition 4.2.1 *(Forcestrategie)*
Es seien ein Spielgraph $G = (V_0, V_1, E, p)$ mit $V = V_0 \cup V_1$ und eine Knotenmenge $A \subseteq V$ gegeben. Eine Funktion $\varrho : U \to V$ heißt eine Forcestrategie nach A für einen Spieler $i \in \{0, 1\}$, wenn der Spieler i von jedem Knoten $v \in ((U \cap V_i) \backslash A)$ aus den Gegenspieler $(1 - i)$ dadurch zu einem Knoten $v \in A$ zu spielen zwingen kann, dass er stets gemäß der Strategie ϱ spielt.

Definition 4.2.2 *(Forcemenge, maximale Forcemenge)*
Eine Knotenmenge $U \subseteq V$ heißt Forcemenge nach A für einen Spieler i, wenn der

Spieler i von jedem Knoten $u \in U$ aus den Gegenspieler (1-i) zwingen kann, zu einem Knoten aus A zu spielen. Eine Forcemenge U nach A heißt maximal, wenn kein Forcemenge U' nach A mit $U \subset U'$ existiert.

Es ist zu beachten, dass die Zielknoten aus A stets zur Forcemenge gehören, da sie bereits erreicht werden, ohne einen Spiellauf zu konstruieren. Für diese Knoten wird keine Forcestrategie definiert. Da wir uns in dieser Arbeit sowohl für eine Forcestrategie als auch für eine maximale Forcemenge interessieren, werden wir eine Forcestrategie berechnen, deren Definitionsbereich eine maximale Forcemenge ist. Im Folgenden wird eine notwendige Bedingung vorgestellt, die zur Bestimmung einer solchen Forcestrategie benutzt wird.

Lemma 4.2.3
Ist eine Forcemenge U nach A für einen Spieler i maximal, dann gibt es keine Kante (v, w) mit $v \in (U \cap V_{1-i})$ und $w \in (V \backslash (U \cup A))$.

Beweis: Gäbe es eine solche Kante, dann würde der Spieler $(1-i)$ von dem Knoten v aus den Knoten w spielen. Wenn der Spieler i trotzdem den Gegenspieler zwingen kann, zu einem Knoten $u \in A$ zu spielen, dann gilt $w \in U$ wegen Maximalität. Ansonsten gilt $v \notin U$. Widerspruch. □

In Abb. 4.2 wird eine Funktion dargestellt, in der eine Forcestrategie durch eine rückwärtsgerichtete Tiefensuche berechnet wird. Es wird dabei ein Zähler *count* verwendet, so dass die Forcestrategie effizient bestimmt werden kann.

Mit V_H bzw. E_H bezeichnen wir die Knotenmenge bzw. die Kantenmenge eines Teilgraphen H in einem Spielgraphen G. Für jeden Knoten $v \in V_H$ bezeichnen wir mit $in(v)$ bzw. $out(v)$ die Menge der Kanten, die zu dem Knoten v eingehen bzw. von dem Knoten v ausgehen, d.h. $in(v) := \{(u, v) \in E_H\}$ und $out(v) := \{(v, w) \in E_H\}$. Die Bezeichnungen werden für die Knotenmengen $K \subseteq V_H$ erweitert, so dass $in(K) := \bigcup_{v \in K} in(v)$ und $out(K) := \bigcup_{v \in K} out(v)$.
Für die Vorgänger- bzw. Nachfolgerknoten von einem Knoten v benutzen wir die Bezeichnung $pred(v)$ bzw. $succ(v)$, so dass $pred(v) := \{u \mid (u, v) \in E_H\}$ bzw. $succ(v) := \{w \mid (v, w) \in E_H\}$ gilt. Für eine Knotenmenge K bedeutet $pred(K)$ bzw. $succ(K)$ eine Vereinigung der Kantenmenge $pred(K) := \bigcup_{v \in K} pred(v)$ bzw. $succ(K) := \bigcup_{v \in K} succ(v)$.

Bei der Funktion force in Abb. 4.2 werden die Kanten aus $(E_H \cup in(A))$ betrachtet. Die Knotenmenge A muss dabei nicht unbedingt eine Teilmenge von V_H sein.

```
force (H : Teilgraph, A : Knotenmenge, i : Spieler) : Strategie
 var  ϱ : Strategie
      v.count : Int (* Zähler *)
      worklist : Stack
 1.  for each v ∈ V_H do
 2.     if (v ∈ V_i) then (* Der Spieler i wählt den nächsten Knoten *)
 3.        v.count := 1 (* Mindestens eine Kante zur Forcemenge ist zu finden *)
 4.     else (* Der Gegenspieler wählt den nächsten Knoten *)
 5.        v.count := |succ(v) ∩ (V_H ∪ A)|
           (* Alle Nachfolgerknoten müssen zur Forcemenge gehören *)
 6.  for each v ∈ A do (* Init. *)
 7.     v.count := 0
 8.     worklist.push(v) (* Startknoten für Tiefensuche *)
 9.  while worklist ≠ ∅ do
10.     v := worklist.pop ()
11.     for each u ∈ (pred(v) ∩ V_H ∩ {u | u.count > 0}) do (* Relevante Vorgängerknoten *)
12.        u.count = u.count − 1 (* Zählerstand um 1 verringert *)
13.        if (u.count = 0) then (* Der Knoten u gehört zu Forcemenge *)
14.           ϱ(u) := v (* Forcestrategie erweitert *)
15.           worklist.push(u)
16.  return ϱ
```

Abbildung 4.2: Diese Funktion berechnet eine Forcestrategie ϱ nach der Knotenmenge A für den Spieler i, so dass die Forcemenge $Def(\varrho)$ maximal ist.

Lemma 4.2.4 *Die Funktion* force(H, A, i) *berechnet eine Forcestrategie ϱ nach der Knotenmenge A für den Spieler i, so dass die Forcemenge $Def(\varrho)$ maximal ist.*

Beweis: Man erkennt leicht, dass hierbei eine rückwärtsgerichtete Tiefensuche durchgeführt wird, die von der Knotenmenge A aus gestartet wird (in Zeilen 9-15). Die Strategie ϱ ist am Anfang die leere Funktion. In Zeilen 1-5 wird die notwendige und ausreichende Bedingung mit dem Zähler $v.count$ festgestellt, wann der Spieler i von einem Knoten v einen Spiellauf für sich günstig konstruieren kann. Für jeden

Knoten $v \in V_H$ zeigt $v.count$ die minimale Anzahl der Nachfolgerknoten, die in der Knotenmenge $(Def(\varrho) \cup A)$ enthalten sein müssen.

Für die Knoten $v \in V_i$ braucht der Spieler i lediglich eine Kante (v, w) mit $w \in (Def(\varrho) \cup A)$ zu finden, da der Spieler i von dem Knoten v aus einen nächsten Knoten wählen darf. Von den Knoten $v \in V_{(1-i)}$ aus wählt jedoch der Spieler $(1 - i)$ den nächsten Knoten, weshalb alle Nachfolgerknoten von v in der Menge $(Def(\varrho) \cup A)$ enthalten sein müssen.

Bei der rückwärtsgerichteten Tiefensuche in Zeilen 9-15 wird der Zähler $v.count$ für $v \in V_H$ um 1 verringert, wenn eine neue Kante (v, w) mit $w \in (Def(\varrho) \cup A)$ behandelt wird. Ein Knoten $v \in V_H$ wird genau dann in dem Stack *worklist* hinzugefügt, wenn sein Zählerstand 0 wird. Da der Stack *worklist* am Anfang mit den Knoten $v \in A$ initialisiert wird (in Zeilen 6-8), gilt die folgende Invariante:

$$\forall\ v \in (Def(\varrho) \cup A) : v.count = 0$$

Die Knoten $v \in (Def(\varrho) \cup A)$ gehören also zur Forcemenge und die Funktion ϱ ist eine Forcestrategie nach A.

Die Maximalität der Forcemenge ergibt sich daraus, dass alle Knoten, die die Bedingung $((v \in V_i \wedge \exists (v, w) \in E : w \in (Def(\varrho) \cup A)) \vee (v \in V_{(1-i)} \wedge \forall (v, w) \in E : w \in (Def(\varrho) \cup A)))$ erfüllt sind, durch die rückwärtsgerichtete Tiefensuche erreicht werden und ihrer Zählerstand irgendwann 0 wird. Das bedeutet, dass alle Knoten, die zur Forcemenge gehören, irgendwann in *worklist* hinzugefügt werden. Da die Funktion ϱ für jeden Knoten aus *worklist* definiert wird, ist die Forcemenge $Def(\varrho)$ nach A maximal. □

Lemma 4.2.5 *Die Funktion* force(H, A, i) *benötigt einen Laufzeitaufwand von* $\mathcal{O}(|E_H| + |in(A)|)$.

Beweis: Es gilt offensichtlich, da eine rückwärtsgerichtete Tiefensuche durchgeführt wird, so dass jede Kante höchstens einmal untersucht wird. □

Wenn die Knotenmenge A eine Gewinnmenge für den Spieler i ist, ist die berechnete Forcemenge $Def(\varrho)$ auch eine Gewinnmenge, da der Spieler i von jedem Knoten $v \in Def(\varrho)$ aus einen Spiellauf konstruieren kann, in dem ein Knoten aus A erreicht wird. Also kann eine Gewinnmenge durch die Benutzung der force-Funktion erweitert werden.

Die berechnete Forcestrategie ϱ ist jedoch nicht unbedingt eine Gewinnstrategie, auch wenn die Knotenmenge A eine Gewinnmenge ist. Bei der Komposition der Forcestrategie ϱ mit einer Gewinnstrategie für A kann passieren, dass es festgestellt wird, dass die Strategie ϱ keine Gewinnstrategie ist. Wir zeigen anhand eines Beispiels, dass die Reihenfolge der Behandlung von Knoten wichtig ist, d.h. für welchen Knoten eine Gewinnstrategie zuerst bestimmt werden muss.

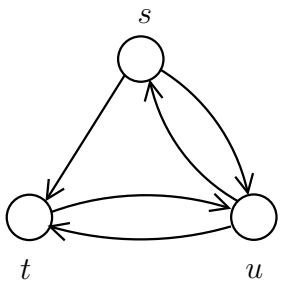

Abbildung 4.3: Spielgraph

Beispiel 4.2.6 *Sei ein Spielgraph* $G = (V_0, V_1, E, p)$ *mit* $s, t, u \in V_0$ *wie Abb. 4.3 gegeben, wobei* $p(s) = 0$ *und* $p(t) = p(u) = 1$. *Weiter sei* $E = \{(s,t), (s,u), (u,s), (t,u), (u,t)\}$.

Der Spieler 0 wählt bei jedem Knoten den nächsten Knoten, da $V_0 = V$ gilt. Es ist leicht zu erkennen, dass der Spieler 0 eine Gewinnstrategie σ mit $\sigma(s) = t, \sigma(t) = u$ und $\sigma(u) = s$ hat. Er kann von jedem Knoten aus einen unendlich langen Spiellauf $\pi = (\cdots \to s \to t \to u \to s \to \cdots)$ konstruieren, so dass er das Spiel gewinnt, da $minInf(\pi) = 0$ gilt.

Wenn wir annehmen, dass die Gewinnmenge $A := \{s, t\}$ zuerst bestimmt und dann eine Forcestrategie für den Knoten u berechnet wird, kann die berechnete Forcestrategie entweder $\varrho(u) = s$ oder $\varrho(u) = t$ sein. Welche Kante als Forcestrategie aufgenommen wird, hängt lediglich davon ab, in welcher Reihenfolge die Knoten s und t untersucht werden.

Die Forcestrategie $\varrho(u) = t$ ist jedoch keine Gewinnstrategie, da eine Schleife $\pi' = (\cdots \to t \to u \to t \to u \to \cdots)$ als Spiellauf konstruiert wird, wobei $minInf(\pi') = 1$ gilt. Ob die berechnete Forcestrategie ϱ eine Gewinnstrategie ist, hängt also von der Gewinnstrategie für die Knotenmenge A ab.

Ein Spiellauf, der nach der Force- sowie Gewinnstrategie konstruiert wird, kann eine Schleife bilden, wodurch die zweite Gewinnbedingung von Definition 4.1.4 in Betracht gezogen werden muss. Wenn diese Möglichkeit bei der Bestimmung der Gewinnstrategie nicht berücksichtigt wird, kann die Richtigkeit der Gewinnstrategie im Zusammenhang mit der Forcestrategie nicht gewährleistet werden.

Die Aufteilung $(\{s,t\},\{u\})$ der Knotenmenge $\{s,t,u\}$ ist jedoch sehr künstlich und geschieht in der Regel nicht. Eine Gewinnstrategie kann beispielsweise wie folgt bestimmt werden:

Die Knotenmenge $\{s,u,t\}$ kann nach Prioritäten in zwei Mengen $\{s\}$ und $\{u,t\}$ eingeteilt werden. Es wird zunächst angenommen, dass der Knoten s zur Gewinnmenge W_0 für den Spieler 0 gehört. Der Grund dieser Annahme liegt darin, dass die Priorität $p(s)$ des Knotens s am kleinsten ist, wobei $(p(s) \bmod 2 = 0)$.

Der Spieler 0 hat also einen Vorteil auf dem Knoten u, so dass er das Spiel gewinnt, wenn der Knoten u unendlich oft dem Spiellauf auftritt. Mit der Annahme $s \in W_0$ kann man dann eine Forcestrategie für die Knoten u und t bestimmen. Die Funktion $\mathsf{force}(G,\{s\},0)$ liefert eine Forcestrategie ϱ mit $\varrho(u) = s$ und $\varrho(t) = u$. Danach muss noch die Richtigkeit der Annahme $s \in W_0$ geprüft werden. Mit einer Gewinnstrategie $\sigma(s) = t$ bzw. $\sigma(s) = u$ wird dies bestätigt und somit ist die berechnete Forcestrategie eine Gewinnstrategie. Insgesamt hat man eine Gewinnstrategie σ mit $\sigma(u) = s$, $\sigma(t) = u$, und $\sigma(s) = t$ bzw. $\sigma(s) = u$.

Die obige Beschreibung ist eine kurze Form eines Algorithmus, wobei die Reihenfolge der Behandlung von Knoten gleich wie beim Model-Checking-Algorithmus in dem vorherigen Kapitel ist. Der Unterschied liegt darin, dass hierbei nicht nur die Bewertung der Knoten sondern auch eine Gewinnstrategie berechnet werden. Wenn ein Spielgraph mehrere Schleifen enthält und vor allem die Knoten verschiedene Prioritäten haben, lässt sich eine Gewinnstrategie schwierig bestimmen.

Wir werden zunächst die Knoten selektieren, für die eine Gewinnstrategie einfach berechnet werden kann. Dazu gehören die Senken und die maximale Forcemenge nach ihnen. In Abb. 4.4 wird eine Funktion dargestellt, die eine Forcestrategie nach Senken berechnet. Der Definitionsbereich der Forcestrategie ist die maximale Forcemenge nach Senken und somit eine Gewinnmenge. Unser Algorithmus läuft dann für den Teilgraphen mit den restlichen Knoten.

Wir verwenden die Bezeichnungen W_0, W_1, σ sowie τ weiter für die Gewinnmengen bzw. Gewinnstrategien.

```
init (H : Spielgraph) : ⟨Strategie, Strategie, Knotenmenge, Knotenmenge⟩
 var  W_0, W_1 : Gewinnmenge
      σ, τ : Gewinnstrategie
 1.  for each v ∈ V_H do
 2.     if (out(v) = ∅) then (* Senken gesammelt *)
 3.        if (v ∈ V_0) then
 4.           W_1 := W_1 ∪ {v}
 5.        else
 6.           W_0 := W_0 ∪ {v}
 7.  σ := force (H, W_0, 0) (* Forcestrategie nach W_0 berechnet *)
 8.  τ := force (H, W_1, 1) (* Forcestrategie nach W_1 berechnet *)
 9.  W_0 := W_0 ∪ Def(σ) (* Gewinnmenge erweitert *)
10.  W_1 := W_1 ∪ Def(τ)
11.  return ⟨σ, τ, W_0, W_1⟩
```

Abbildung 4.4: Diese Funktion berechnet eine Forcestrategie σ bzw. τ nach Senken für den Spieler 0 bzw. 1 sowie die maximale Forcemenge W_0 bzw. W_1 in dem Spielgraphen H.

Lemma 4.2.7 *Die durch* init(H) *berechnete Knotenmenge W_0 bzw. W_1 ist eine Gewinnmenge für den Spieler 0 bzw. 1. Außerdem ist σ bzw. τ eine Gewinnstrategie.*

Beweis: Wir zeigen zunächst, dass W_0 eine Gewinnmenge für den Spieler 0 ist. In Zeilen 1-6 werden die Knoten $v \in V_1$, die keine ausgehende Kante besitzen, in W_0 gesammelt. Nach der ersten Gewinnbedingung in Definition 4.1.4 gehören solche Knoten zur Gewinnmenge W_0. Danach wird die force-Funktion mit der Gewinnmenge W_0 ausgeführt (in Zeile 7).

Für den Fall, dass die Knotenmenge W_0 leer ist, ist die berechnete Forcestrategie σ die leere Funktion. Die Knotenmenge $Def(\sigma)$ dann leer und folglich W_0 auch. Offensichtlich ist W_0 eine Gewinnmenge.

Wir betrachten nun den anderen Fall, dass die Knotenmenge W_0 nach Zeile 6 nicht leer ist. Da die Knotenmenge W_0 bis Zeile 6 nur Senken $v \in V_1$ enthält, kann keine Schleife als Spiellauf entstehen, wenn das Spiel von einem Knoten $v \in Def(\sigma)$ aus gestartet wird und der Spieler 0 den nächsten Knoten gemäß der Forcestrategie σ für alle $v \in (Def(\sigma) \cap V_0)$ wählt. Das heißt, dass der Spieler 0 von den Knoten

$v \in Def(\sigma)$ aus den Spieler 1 zwingen kann, nach einer Senke $v \in V_1$ zu spielen. Also ist die Knotenmenge W_0 immer noch eine Gewinnmenge nach der Operation der Zeile 9 und die Forcestrategie σ eine Gewinnstrategie für den Spieler 0.

Analog ist W_1 eine Gewinnmenge und τ eine Gewinnstrategie für den Spieler 1. □

Lemma 4.2.8 *Die Funktion* $\mathsf{init}(H)$ *benötigt einen Laufzeitaufwand von* $\mathcal{O}(|V_H| + |E_H|)$.

Beweis: Die Funktion $\mathsf{force}(H, W_0, 0)$ sowie $\mathsf{force}(H, W_1, 1)$ benötigt jeweils eine Laufzeitkomplexität von $\mathcal{O}(|E_H|)$. Da die for-Schleife in Zeile 1-6 eine Laufzeit von $\mathcal{O}(|V_H|)$ braucht, beträgt die Laufzeitkomplexität der Funktion $\mathsf{init}(H)$ insgesamt $\mathcal{O}(|E_H| + |V_H|)$. □

Nach der Ausführung von $\mathsf{init}(H)$ erhält man zwei disjunkte Gewinnmengen W_0 und W_1. Existiert mindesten ein Knoten $v \notin (W_0 \cup W_1))$ in dem Spielgraphen H, dann bedeutet dies, dass der Spielgraph mindestens eine Schleife enthält, für die die zweite Gewinnbedingung von Definition 4.1.4 berücksichtigt werden muss.

Eine einmalige Ausführung der Tiefensuche ist normalerweise wegen Schleifen nicht ausreichend, um zu entscheiden, ob eine Strategie eine Gewinnstrategie ist. Die möglichen Spielläufe müssen systematisch geprüft werden. Das Ergebnis wird temporär gespeichert und im Lauf der weiteren Berechnung korrigiert.

Ein zentrales Problem beim Entwurf eines spielbasierten Model-Checking-Algorithmus ist es zu entscheiden, welches Zwischenergebnis von welcher Berechnungsphase man behalten bzw. verwerfen soll. Wir werden in diesem Abschnitt einen neuen Algorithmus entwickeln, in dem die Spielgraphen systematisch untersucht werden, so dass ein sukzessiver Aufbau einer Gewinnstrategie ermöglicht wird.

Die Knoten, die zur Forcemenge nach einer Senke gehören, können durch Ausführung der Funktion $\mathsf{init}(H)$ einfach selektiert werden. Wir entwerfen deshalb einen Algorithmus, in dem die restlichen Knoten bewertet werden. Die Knoten sind entweder in einer Schleife enthalten oder haben mindestens einen Nachfolgerknoten, vom dem aus eine Schleife erreichbar ist.

Bei unserem Algorithmus werden wir Strategien als Zwischenergebnis berechnen, die jeweils eine Gewinnstrategie oder eine Forcestrategie sind. Damit eine Forcestrategie $\varrho : U \to V$ nach einer Knotenmenge A eine Gewinnstrategie wird, müssen die Knoten $v \in A$ zunächst zur Gewinnmenge gehören. Dies ist jedoch keine ausreichende

Bedingung. Dafür muss die Gewinnstrategie für A zusammen mit der Forcestrategie ϱ noch geprüft werden, ob sie als eine Gewinnstrategie bleibt. Ist eine Strategie für A unabhängig von der Forcestrategie ϱ eine Gewinnstrategie, dann ist die Forcestrategie ϱ auch eine Gewinnstrategie. Wir werden solche Gewinnstrategien bzw. Forcestrategien berechnen, die als Gewinnstrategie verwendet werden.

Im Folgenden wird die Vorgehensweise unseres Algorithmus zur Bestimmung einer Gewinnstrategie sowie Gewinnmenge kurz erläutert. Die Knoten $v \in V$ werden nach ihren Priorität partitioniert.

Zunächst wird angenommen, dass die Knoten v mit $(p(v) \bmod 2) = 0$ bzw. 1 zu W_0 bzw. zu W_1 gehören. Danach wird die Korrektheit der Annahme sukzessiv nachgeprüft, ob der Gegenspieler die Annahme widerlegen kann. Wenn z.B. ein Knoten $v \in W_0$ zur Forcemenge nach W_1 gehört, bedeutet dies, dass die Annahme nicht korrekt ist. Dann wird die Forcestrategie für den Knoten v als Gewinnstrategie für den Spieler 1 gespeichert und es gilt nun $v \in W_1$.

Anschließend muss die Aufteilung der Gewinnmengen sowie die Gewinnstrategie für die Knoten w mit $p(w) > p(v)$ neu initialisiert und ihre Korrektheit nachgeprüft werden, indem eine eine Rekursion nach der Reihenfolge von Prioritäten durchgeführt wird.

Zum Schluss wird eine Gewinnstrategie für die Knoten bestimmt, die zur aktuell behandelnden Partition gehören, und deren Zugehörigkeit zur Gewinnmenge von dem Gegenspieler nicht widerlegt werden kann.

Die Spiele, die in dieser Arbeit behandelt werden, sind Zweierspiel, d.h. die Spielsituationen der beiden Spieler sind komplementär. Wenn ein Spieler i von einem Knoten $u \in U$ aus den Gegenspieler $(1-i)$ nicht zwingen kann, nach einem Knoten $a \in A$ zu spielen, dann bedeutet dies, dass der Gegenspieler $(1-i)$ einen Spiellauf konstruieren kann, der in dem Bereich außerhalb von A bleibt.

Satz 4.2.9 *Es seien $H = (V_0, V_1, E, p)$ ein Spielgraph, W_0 bzw. W_1 eine Gewinnmenge für den Spieler 0 bzw. 1 und C eine komplementäre Knotenmenge mit $C = (V_H \backslash (W_0 \cup W_1))$, wobei alle Knoten aus C eine höchste Priorität m besitzen, d.h. $p(v) = m$ für alle $v \in C$ und $p(w) \geq m$ für alle $w \in (W_0 \cup W_1)$. Außerdem habe jeder Knoten $v \in (C \cap V_i)$ mindestens einen Nachfolgerknoten, aber kein Knoten aus C gehöre zur Forcemenge nach $W_{(1-i)}$ für den Spieler $(1-i)$, wobei $i := (m \bmod 2)$. Dann gelten:*

- *Die Knotenmenge* $(C \cup W_i)$ *ist eine Gewinnmenge für den Spieler* i*.*
- *Eine Gewinnstrategie* ϱ *für den Spieler* i *auf* C *kann dadurch bestimmt werden, dass eine Kante* $(v, \varrho(v))$ *mit* $\varrho(v) \in (C \cup W_i)$ *für jeden Knoten* $v \in (C \cap V_i)$ *gewählt wird.*

Beweis: Nach der Voraussetzung gehört kein Knoten aus C zur Forcemenge nach $W_{(1-i)}$ für den Spieler $(1-i)$. Daraus folgt direkt

- $v \in (C \cap V_i) \Rightarrow (\exists w \in succ(v) : w \in (C \cup W_i))$ und
- $v \in (C \cap V_{(1-i)}) \Rightarrow (\forall w \in succ(v) : w \in (C \cup W_i))$.

Also kann der Spieler i eine Strategie ϱ bestimmen, so dass $\varrho(v) \in (C \cup W_i)$ für jedes $v \in (C \cap V_i)$ gilt. Es bleibt noch zu zeigen, dass die Knotenmenge $(C \cup W_i)$ eine Gewinnmenge ist, und dass die Strategie ϱ eine Gewinnstrategie für den Spieler i ist.

Wir betrachten dafür die Spielläufe, die unter Verwendung einer solchen Strategie ϱ konstruiert werden. Da W_i nach der Voraussetzung eine Gewinnmenge für den Spieler i ist, gibt es eine Gewinnstrategie für diese Gewinnmenge. Wir nehmen also an, dass wir eine solche Gewinnstrategie auf W_i wählen. Die Spielläufe, die mit einem Knoten $v \in W_i$ beenden, oder in denen ausschließlich die Knoten aus W_i besucht werden, brauchen wir nicht zu betrachten.

Für den Fall, dass ein Spiellauf mit einem Knoten $v \in C$ beendet, gilt $v \in V_{(1-i)}$, da jeder Knoten aus $(C \cap V_i)$ mindestens einen Nachfolgerknoten besitzt. Der Spieler i gewinnt das Spiel nach der ersten Gewinnbedingung in Definition 4.1.4.

Wenn ein Spiellauf unendlich lang ist, wobei mindestens ein Knoten $v \in C$ unendlich oft besucht wird, dann gewinnt der Spieler i das Spiel, da der Knoten v die höchste Priorität m mit $i = (m \bmod 2)$ besitzt.

Insgesamt ist die Knotenmenge $(C \cup W_i)$ also eine Gewinnmenge und die Strategie ϱ eine Gewinnstrategie für den Spieler i in H. □

Es ist zu beachten, dass die Gewinnstrategie ϱ für den Spieler i auf C des Satzes 4.2.9 unabhängig von einer Gewinnstrategie auf W_i bestimmt werden kann. In unserem Algorithmus wird eine Iteration mit einem Spielgraphen gestartet, in der eine Rekursion enthalten ist. Die Vorbedingungen des Satzes 4.2.9 werden genau

der Abbruchbedingung der Iteration entsprechen, so dass die Gewinnstrategie nach Durchführung der Iteration wie in dem Satz erweitert werden kann.

Im Folgenden wird eine Funktion comp dargestellt, in der ein Nachfolgerknoten aus $(C \cup W_i)$ als Strategie ausgewählt wird. Die Vorbedingungen des Satzes 4.2.9 müssen bzgl. eines Spielgraphen H und der Gewinnmenge $W_{(1-i)}$ erfüllt sein, die in der Funktion nicht explizit angegeben werden. Diese Funktion kann nur dann sinnvoll benutzt werden.

comp(C, W_i : *Knotenmenge*, i : *Spieler*) : *Strategie*
 var ϱ : *Strategie*
 1. for each $v \in (C \cap V_i)$
 2. select $e = (v, w) \in E$ with $w \in (C \cup W_i)$
 3. $\varrho(v) := w$
 4. return ϱ

Abbildung 4.5: Diese Funktion bestimmt eine Strategie $\varrho : (C \cap V_i) \rightarrow (C \cup W_i)$.

Lemma 4.2.10 *Die Funktion* comp(C, W_i, i) *benötigt einen Laufzeitaufwand von* $\mathcal{O}(|out(C)|)$.

Beweis: Jeder Knoten $v \in C$ wird genau einmal behandelt, so dass eine ausgehende Kante aus v gewählt wird. □

Wir stellen noch eine zusätzliche Funktion comp2 vor, die im speziellen Fall an der Stelle von comp-Funktion benutzt werden kann. Wenn die Knotenmenge W_i leer ist, kann man auch eingehende Kanten zu C anstatt ausgehende Kanten nach C durchsuchen, um eine Strategie $\varrho : (C \cap V_i) \rightarrow C$ zu bestimmen. Die Vorbedingungen des Satzes 4.2.9 müssen für diese Funktion auch erfüllt sein, so dass die Existenz einer Kante (u, v) mit $u \in C$ für jedes $v \in (C \cap V_i)$ garantiert wird.

Die Funktion comp2 werden wir im Fall verwenden, dass die Priorität aller Knoten von Spielgraphen denselben Modulo-Wert hat. Durch Benutzung der Funktion comp2 wird die Abschätzung der Laufzeitkomplexität unseres Algorithmus erleichtert.

Lemma 4.2.11 *Die Funktion* comp2(C, i) *benötigt eine Laufzeit von* $\mathcal{O}(|in(C)|)$.

comp2(C : *Knotenmenge*, i : *Spieler*) : *Strategie*
var ϱ : *Strategie*
1. for each $v \in C$
2. for each $(u, v) \in E$
3. if $u \in (C \cap V_i) \ \wedge \ \varrho(u) = undef$. then
4. $\varrho(u) := v$
5. return ϱ

Abbildung 4.6: Diese Funktion bestimmt eine Strategie $\varrho : (C \cap V_i) \to C$.

Beweis: Offensichtlich. □

Im Folgenden wird ein Algorithmus main entworfen, in dem hauptsächlich die Funktion synthesize nach gewisser Vorbereitung aufgerufen wird. Mit der Funktion init wird zuerst die triviale Gewinnstrategie sowie Gewinnmenge berechnet. Dann wird die Funktion synthesize für die restlichen Knoten ausgeführt, die in Abb. 4.8 vorgestellt wird.

main (H : *Spielgraph*) : $\langle$*Strategie, Strategie, Knotenmenge, Knotenmenge*$\rangle$
var $\sigma, \tau, \sigma', \tau'$: *Strategie*
W_0, W_1, K : *Knotenmenge*
1. $\langle \sigma, \tau, W_0, W_1 \rangle := \mathsf{init}(H)$ (∗ *Forcestrategie nach Senken bestimmt* ∗)
2. $K := V_H \backslash (W_0 \cup W_1)$ (∗ *restliche Knoten selektiert* ∗)
3. $\langle \sigma', \tau' \rangle := \mathsf{synthesize}(H|_K)$ (∗ *Gewinnstrategie berechnet* ∗)
4. $W_0 := W_0 \cup Def(\sigma')$, $W_1 := W_1 \cup Def(\tau')$ (∗ *Gewinnmengen vereinigt* ∗)
5. $\sigma := \sigma + \sigma'$, $\tau := \tau + \tau'$ (∗ *Gewinnstrategien vereinigt* ∗)
6. return $\langle \sigma, \tau, W_0, W_1 \rangle$

Abbildung 4.7: Diese Funktion berechnet die Gewinnmenge W_0 bzw. W_1 sowie eine Gewinnstrategie σ bzw. τ für den Spieler 0 bzw. 1 in dem Spielgraphen H.

Wir bezeichnen mit V_H bzw. E_H die Knotenmenge bzw. die Kantenmenge von H und mit $H|_K$ einen Teilgraphen von H, der auf der Knotenmenge K eingeschränkt ist, d.h. $V_{H|_K} = K$ und $E_{H|_K} \subseteq K \times K$.

Wir verwenden die Bezeichnung K insbesondere für die Menge der Knoten, deren Gewinnstrategie noch bestimmt werden soll. Beim Aufruf der Funktion

synthesize $(H : \textit{Spielgraph}) : \langle \textit{Strategie, Strategie} \rangle$
var $i, m, m' : \textit{Integer}$
$K, K', \widehat{K} : \textit{Knotenmenge}$
$\xi, \xi', \widehat{\xi}, \rho, \rho', \widehat{\rho} : \textit{Strategie}$
1. if $(V_H = \emptyset)$ then $(* \textit{ Spielgraph ist leer } *)$
2. return $\langle 0, 0 \rangle$ $(* \textit{ Nullfunktion zurückgegeben } *)$
3. else $(* \textit{ Spielgraph nicht leer } *)$
4. $m := min\{p(v) \mid v \in V_H\}$ $(* \textit{ Höchste Priorität bestimmt } *)$
5. $i := (m \bmod 2)$ $(* \textit{ Spieler } i \textit{ hat Vorteil } *)$
6. if $(\{v \in V_H \mid (p(v) \bmod 2) \neq i\} = \emptyset)$ then $(* \textit{ Keine weitere Rekursion } *)$
7. if $(i = 0)$ then return $\langle \mathsf{comp2}\,(V_H, i), 0 \rangle$ $(* \textit{ Gewinnstrategie für Spieler } 0 *)$
8. else return $\langle 0, \mathsf{comp2}\,(V_H, i) \rangle$ $(* \textit{ Gewinnstrategie für Spieler } 1 *)$
9. else $(* \textit{ Existiert Knoten mit niedriger Priorität } *)$
10. $K := V_H$ $(* \textit{ Suchraum mit } K \textit{ gekennzeichnet } *)$
11. $m' := min\{p(v) \mid v \in V_H \wedge ((p(v) \bmod 2) \neq i)\}$
$(* \textit{ Nächst höchste Priorität mit anderem Modulo-Wert bestimmt } *)$
12. $\widehat{K} := \{v \in K \mid p(v) < m'\}$, $K' := K \backslash \widehat{K}$
$(* \textit{ Menge der Knoten mit höchster Priorität von Restlichen unterschieden } *)$
13. repeat
14. if $(\widehat{K} \neq \emptyset)$ then $\widehat{\xi} := \mathsf{force}\,(H|_{K'}, \widehat{K}, i)$ $(* \textit{ Forcestrategie nach } \widehat{K} \textit{ bestimmt } *)$
15. else $\widehat{\xi} := 0$
16. if $(i = 0)$ then $\langle \xi', \rho' \rangle := \mathsf{synthesize}\,(H|_{(K' \backslash Def(\widehat{\xi}))})$ $(* \textit{ Rekursion } *)$
17. else $\langle \rho', \xi' \rangle := \mathsf{synthesize}\,(H|_{(K' \backslash Def(\widehat{\xi}))})$
18. $\rho := \rho + \rho'$ $(* \ \rho' \textit{ zur Gewinnstrategie addiert } *)$
19. $K := K \backslash Def(\rho')$, $K' := K' \backslash Def(\rho')$
$(* \textit{ Gewinnmenge } Def(\rho') \textit{ für den Spieler } (1 - i) \textit{ herausgenommen } *)$
20. if $(\widehat{K} \neq \emptyset \wedge Def(\rho') \neq \emptyset)$ then
21. $\widehat{\rho} := \mathsf{force}\,(H|_K, Def(\rho'), (1 - i))$ $(* \textit{ Forcestrategie nach } Def(\rho') \textit{ bestimmt } *)$
22. $\rho := \rho + \widehat{\rho}$ $(* \ \widehat{\rho} \textit{ zur Gewinnstrategie addiert } *)$
23. $K := K \backslash Def(\widehat{\rho})$, $\widehat{K} := \widehat{K} \backslash Def(\widehat{\rho})$, $K' := K' \backslash Def(\widehat{\rho})$ $(* \textit{ Monotonie } *)$
24. else $\widehat{\rho} := 0$
25. until $(Def(\widehat{\rho}) = \emptyset)$ $(* \textit{ bis stabilisiert } *)$
26. $\xi := \widehat{\xi} + \xi' + \mathsf{comp}\,(\widehat{K}, (Def(\widehat{\xi}) \cup Def(\xi')), i)$ $(* \textit{ Strategie } \xi \textit{ vervollständigt } *)$
27. if $(i = 0)$ then return $\langle \xi, \rho \rangle$ $(* \textit{ Reihenfolge der Ausgabe geordnet } *)$
28. else return $\langle \rho, \xi \rangle$

Abbildung 4.8: Diese Funktion berechnet eine Gewinnstrategie ξ bzw. ρ, deren Definitionsbereich eine Gewinnmenge ist, wobei H ein Spielgraph ohne Senke ist.

synthesize(H) wird die Knotenmenge K mit $K := V_H$ festgelegt, deren Größe sukzessiv verkleinert wird. Die Knoten mit höchster Priorität und demselben Modulo-Wert werden mit $\widehat{K}$ bezeichnet. Die restlichen Knoten werden mit K' bezeichnet.

Die Strategien der beiden Spieler werden mit ξ, $\widehat{\xi}$, ξ', ρ, $\widehat{\rho}$, ρ' bezeichnet, wobei ξ, $\widehat{\xi}$ bzw. ξ' eine Strategie für den Spieler i mit $i := (m \bmod 2)$ ist, während ρ, $\widehat{\rho}$ bzw. ρ' eine Strategie für den Spieler $(1-i)$ ist. Mit $\widehat{\xi}$ wird eine Forcestrategie nach $\widehat{K}$, mit ξ' bzw. ρ' eine Gewinnstrategie für den Spieler i bzw. $(1-i)$ in $H|_{K'}$ und mit $\widehat{\rho}$ eine Forcestrategie nach $Def(\rho')$ bezeichnet.

In der Funktion synthesize(H) in Abb. 4.8 wird eine zusätzliche Bedingung zu den Vorbedingungen des Satzes 4.2.9 angenommen, dass der Spielgraph H keine Senke enthält. Dies ist eine wichtige invariante Eigenschaft zum Korrektheitsbeweis der synthesize-Funktion, die wir in Satz 4.3.2 zeigen. Die Ausführung der init-Funktion sorgt dafür, dass diese Bedingung beim ersten Aufruf der synthesize-Funktion erfüllt wird.

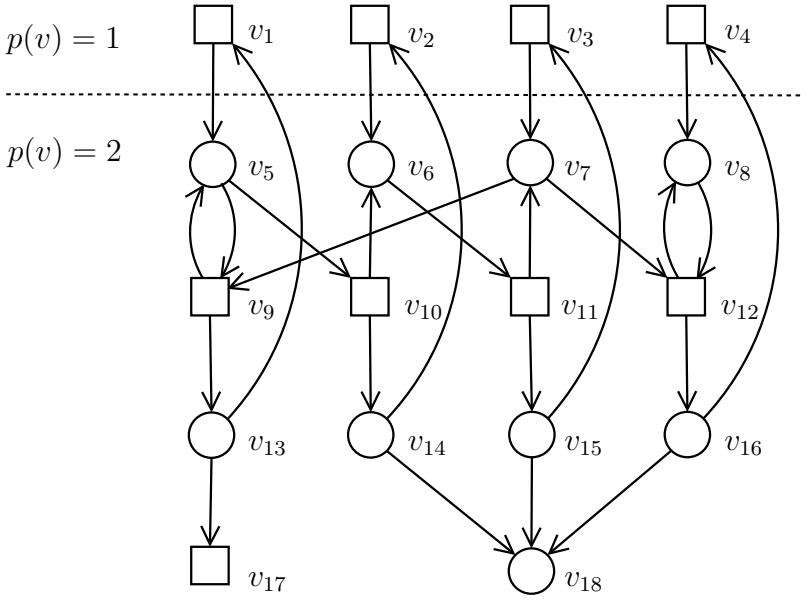

Abbildung 4.9: Spielgraph

Bevor wir den Korrektheitsbeweis für die synthesize- sowie main-Funktion zeigen, illustrieren wir die Berechnungsschritte des Algorithmus main anhand eines Beispiels 4.2.12. Die Knoten $v \in V_0$ des Spielers 0 werden dabei als Kreise dargestellt und die Knoten $v \in V_1$ des Spielers 1 als Quadrate.

Beispiel 4.2.12 *Ein Spielgraph $G = (V_0, V_1, E, p)$ sei wie in Abb. 4.9 gegeben, wobei $V_0 = \{v_5, v_6, v_7, v_8, v_{13}, v_{14}, v_{15}, v_{16}, v_{18}\}$, $V_1 = \{v_1, v_2, v_3, v_4, v_9, v_{10}, v_{11}, v_{12}, v_{17}\}$ und*

$$p(v_k) := \begin{cases} 1 & f\ddot{u}r\ 1 \leq k \leq 4 \\ 2 & sonst \end{cases}$$

Nach der Ausführung der init*-Funktion erhält man eine Gewinnstrategie $\sigma(v_{13}) = v_{17}$ und eine Gewinnmenge $W_0 = \{v_{13}, v_{17}\}$ sowie $W_1 = \{v_{18}\}$. Die Knoten der Gewinnmengen werden aussortiert und die Funktion* synthesize *wird von der* main*-Funktion mit dem Teilgraphen aufgerufen, dessen Knoten die restlichen Knoten sind.*

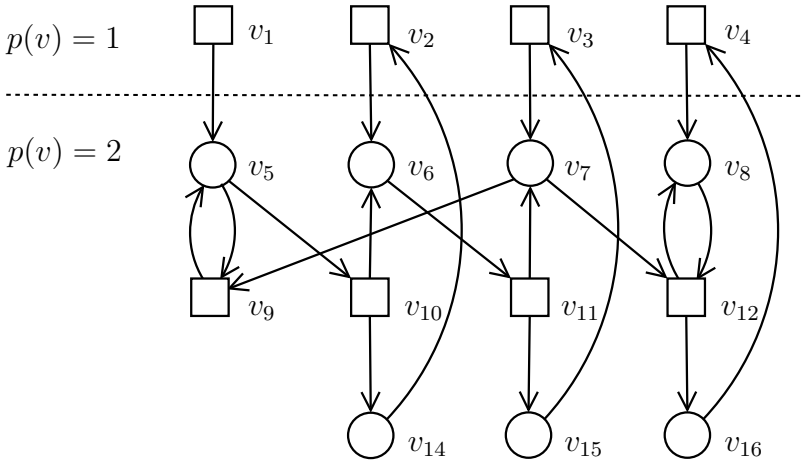

Abbildung 4.10: Die Funktion synthesize wird von der Hauptfunktion main mit diesem Graphen ausgerufen.

In Abb. 4.10 wird dargestellt, welche Knoten und Kanten in der synthesize*-Funktion betrachtet werden, die von der Hauptfunktion* main *aufgerufen wird. Die Kanten, deren Quell- bzw. Zielknoten zu einer Gewinnmenge W_0 bzw. W_1 gehören, werden von dem Spielgraphen ausgeschlossen und bei der Ausführung der* synthesize*-Funktion nicht betrachtet.*

Da der Spielgraph nicht leer ist, wird der else*-Teil in Zeile 3 ausgeführt. Die höchste Priorität m sowie ihre Modulo-Wert i ist gleich mit 1. Da die Knoten mit einer niedrigeren Priorität existieren, wird der* else*-Teil in Zeile 9 ausgeführt. Es gelten $m = 1$, $i = 1$, $\widehat{K} = \{v_1, v_2, v_3, v_4\}$, $K' = \{v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{14}, v_{15}, v_{16}\}$ und $K = (\widehat{K} \cup K')$.*

In Zeile 14 der synthesize*-Funktion wird die Funktion* force *aufgerufen, die in Abb. 4.2 vorgestellt wurde. Dabei wird ein Zähler count benutzt, so dass die Forcestrategie sowie die Forcemenge effizient bestimmt werden kann. In Abb. 4.11 wird die Initialisierung sowie Veränderung des Zählers count tabellarisch dargestellt, wobei die Position der Felder genau den Knoten des Spielgraphen in Abb. 4.10 entspricht.*

0	0	0	0
2	1	2	1
1	1	1	1
	1	1	1

0	0	0	0
1	1	2	1
1	0	1	1
	0	1	1

0	0	0	0
1	0	2	1
1	0	0	1
	0	0	1

0	0	0	0
1	0	1	0
1	0	0	0
	0	0	0

Abbildung 4.11: Diese Tabellen beinhalten die Veränderung des Zählers *count* bei Ausführung der force-Funktion.

Es wird eine Forcestrategie $\widehat{\xi}$ *nach* $\{v_1, v_2, v_3, v_4\}$ *für den Spieler* 1 *gesucht. Die berechnete Forcestrategie* $\widehat{\xi}$ *sieht wie folgt aus:* $\widehat{\xi}(v_{14}) = v_2$, $\widehat{\xi}(v_{10}) = v_{14}$, $\widehat{\xi}(v_{15}) = v_3$, $\widehat{\xi}(v_{11}) = v_{15}$, $\widehat{\xi}(v_6) = v_{11}$, $\widehat{\xi}(v_{16}) = v_4$, $\widehat{\xi}(v_{12}) = v_{16}$, *und* $\widehat{\xi}(v_8) = v_{12}$. *Davon sind nur die Strategien für die Knoten* v_{10}, v_{11}, *und* v_{12} *relevant, da der Spieler 1 von diesen Knoten aus einen Nachfolgerknoten wählen kann. Jedoch dient der diese Berechnung auch dazu, die maximale Forcemenge zu bestimmen. Für die Knoten* $v \in (Def(\widehat{\xi}) \cap V_0)$ *gilt, dass der Spieler* 0 *von den Knoten aus nicht vermeiden kann, nach* $\{v_1, v_2, v_3, v_4\}$ *zu spielen.*

Nach dem Ergebnis wird dann ein Teilgraph der Abb. 4.12 selektiert, dessen Knoten nicht zu dem Definitionsbereich $Def(\widehat{\xi})$ *gehören. Da alle Knoten* v_5, v_7 *und* v_9 *dieselbe Priorität haben, wird die* comp2*-Funktion in der Zeile 7 der rekursiv aufgerufen* synthesize*-Funktion ausgeführt. In Zeile 17 erhält man dann eine Strategie* ρ' *mit* $\rho'(v_5) = v_9$, $\rho'(v_9) = v_5$ *und* $\rho'(v_7) = v_9$.

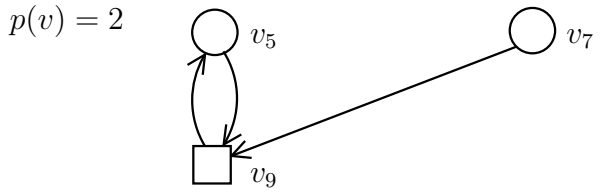

Abbildung 4.12: Die Funktion synthesize wird für diesen Graphen rekursiv ausgeführt.

Dann wird die force*-Funktion in der Zeile 21 der* synthesize *aufgerufen, so dass eine maximale Forcemenge sowie Forcestrategie nach* $\{v_5, v_7, v_9\}$ *berechnet wird. Der Zähler count wird dafür neu initialisiert. In Abb. 4.13 wird die Initialisierung sowie Veränderung des Zählers count tabellarisch dargestellt, wenn die* force*-Funktion in der Zeile 21 ausgeführt wird. Die berechnete Forcestrategie für den Spieler 0 lautet:* $\widehat{\rho}(v_1) = v_5$, $\widehat{\rho}(v_3) = v_7$, $\widehat{\rho}(v_{15}) = v_3$, $\widehat{\rho}(v_{11}) = v_{15}$, $\widehat{\rho}(v_6) = v_{11}$, $\widehat{\rho}(v_2) = v_6$, $\widehat{\rho}(v_{14}) = v_2$, *und* $\widehat{\rho}(v_{10}) = v_{14}$.

1	1	1	1
0	1	0	1
0	1	1	2
	1	1	1

0	1	1	1
0	1	0	1
0	1	1	2
	1	1	1

0	0	0	1
0	0	1	1
0	0	0	2
	0	0	1

0	0	0	1
0	0	0	1
0	0	0	2
	0	0	1

Abbildung 4.13: Diese Tabellen beinhalten die Veränderung des Zählers *count* bei Ausführung der force-Funktion.

Die restlichen Knoten sind v_4, v_8, v_{12} *und* v_{16}. *Nach Ausführung der Zeile 23 gelten* $\widehat{K} = \{v_4\}$ *und* $K' = \{v_8, v_{12}, v_{16}\}$. *Da die Knotenmenge* $Def(\widehat{\rho})$ *nicht leer ist, ist die Abbruchbedingung der* repeat*-Schleife in der Zeile 25 der* synthesize*-Funktion nicht erfüllt. Deshalb wird der Rumpf der Schleife erneut durchgeführt.*

Dann wird die force*-Funktion in Zeile 14 ausgeführt, so dass man eine Forcestrategie* $\widehat{\xi}$ *nach* $\{v_4\}$ *erhält, wobei* $\widehat{\xi}(v_{16}) = v_4$, $\widehat{\xi}(v_{12}) = v_{16}$, *und* $\widehat{\xi}(v_8) = v_{12}$. *Der Spielgraph in Abb. 4.14 wird als erster Parameter benutzt.*

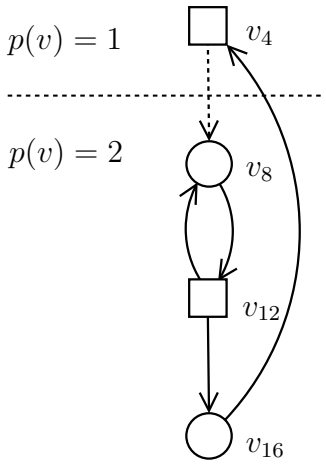

Abbildung 4.14: Für diesen Spielgraphen wird die Funktion force ausgeführt.

Die Knotenmenge $(K' \backslash Def(\widehat{\xi}))$ *ist nun leer, da* $K' = \{v_8, v_{12}, v_{16}\} = Def(\widehat{\xi})$ *gilt. Die zu berechnende Strategie* ρ' *in Zeile 17 ist deshalb die leere Funktion. In Zeile 24 erhält die Strategie* $\widehat{\rho}$ *dann 0. Die Abbruchbedingung der* repeat*-Schleife ist erfüllt, und keine weitere* synthesize*-Funktion wird rekursiv aufgerufen. Zuletzt wird die Funktion* comp *für den Knoten* v_4 *in Zeile 26 ausgeführt, so dass die Strategie* $\xi(v_4) = v_8$ *bestimmt wird.*

Die berechnete Strategie ξ *bzw.* ρ *sieht insgesamt wie folgt aus:* $\xi(v_1) = v_5$, $\xi(v_2) = v_6$, $\xi(v_3) = v_7$, $\xi(v_5) = v_9$, $\xi(v_6) = v_{11}$, $\xi(v_7) = v_9$, $\xi(v_9) = v_5$, $\xi(v_{10}) = v_{14}$, $\xi(v_{11}) = v_{15}$, $\xi(v_{14}) = v_2$, $\xi(v_{15}) = v_3$, $\rho(v_4) = v_8$, $\rho(v_8) = v_{12}$, $\rho(v_{12}) = v_{16}$, *und* $\rho(v_{16}) = v_4$. *Da die höchste Priorität* 1 *ist, wird* ρ *bzw.* ξ *als die Strategie für den Spieler* 0 *bzw.* 1 *zurückgegeben.*

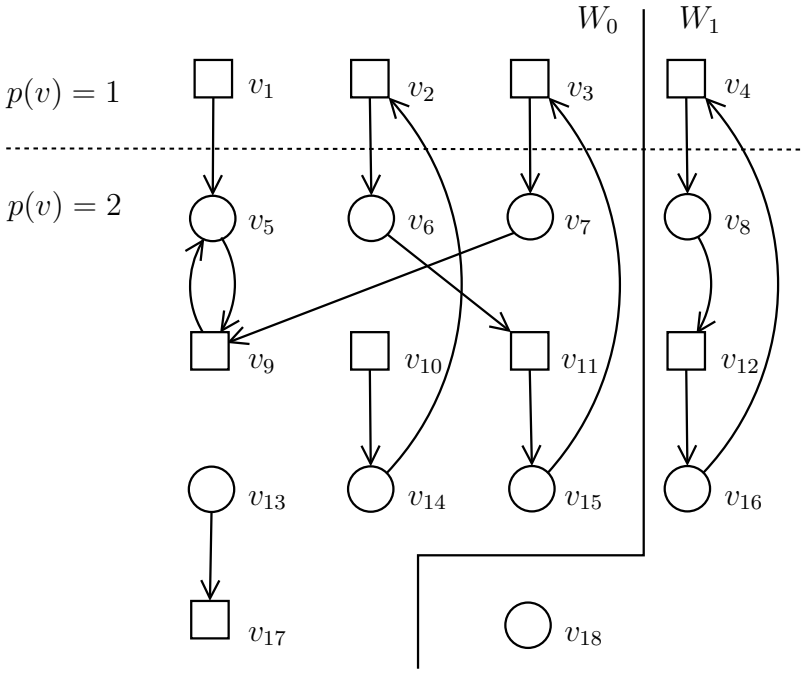

Abbildung 4.15: Berechnete Strategie und Aufteilung der Gewinnmengen

Die gesamte Ausführung der synthesize-Funktion wird somit abgeschlossen und wir kommen zurück zur main-Funktion. Nach der Vereinigung der Gewinnmengen in Zeile 4 erhält man schließlich die Gewinnmenge W_0 bzw. W_1 sowie Gewinnstrategie σ bzw. τ für den Spieler 0 bzw. 1, wie sie in Abb. 4.15 dargestellt werden.

Die berechnete Gewinnstrategie σ bzw. τ für die Knoten $v \in ((V_0 \cap W_0) \cup (V_1 \cap W_1))$ sieht wie folgt aus: $\sigma(v_5) = v_9$, $\sigma(v_6) = v_{11}$, $\sigma(v_7) = v_9$, $\sigma(v_{13}) = v_{17}$, $\sigma(v_{14}) = v_2$, $\sigma(v_{15}) = v_3$, $\tau(v_4) = v_8$, $\tau(v_{12}) = v_{16}$.

Wir haben ein Beispiel ausführlich betrachtet, in dem die Gewinnmenge sowie Gewinnstrategie nach unserem Algorithmus bestimmt wird. Die Berechnung erfolgt hauptsächlich durch rekursive Ausführung der synthesize-Funktion in der repeat-Schleife. Die Knoten des Spielgraphen werden in jeder Iteration so aufgeteilt, dass die Knoten, deren Strategie bei weiteren Iterationen beibehalten werden darf, aussortiert werden.

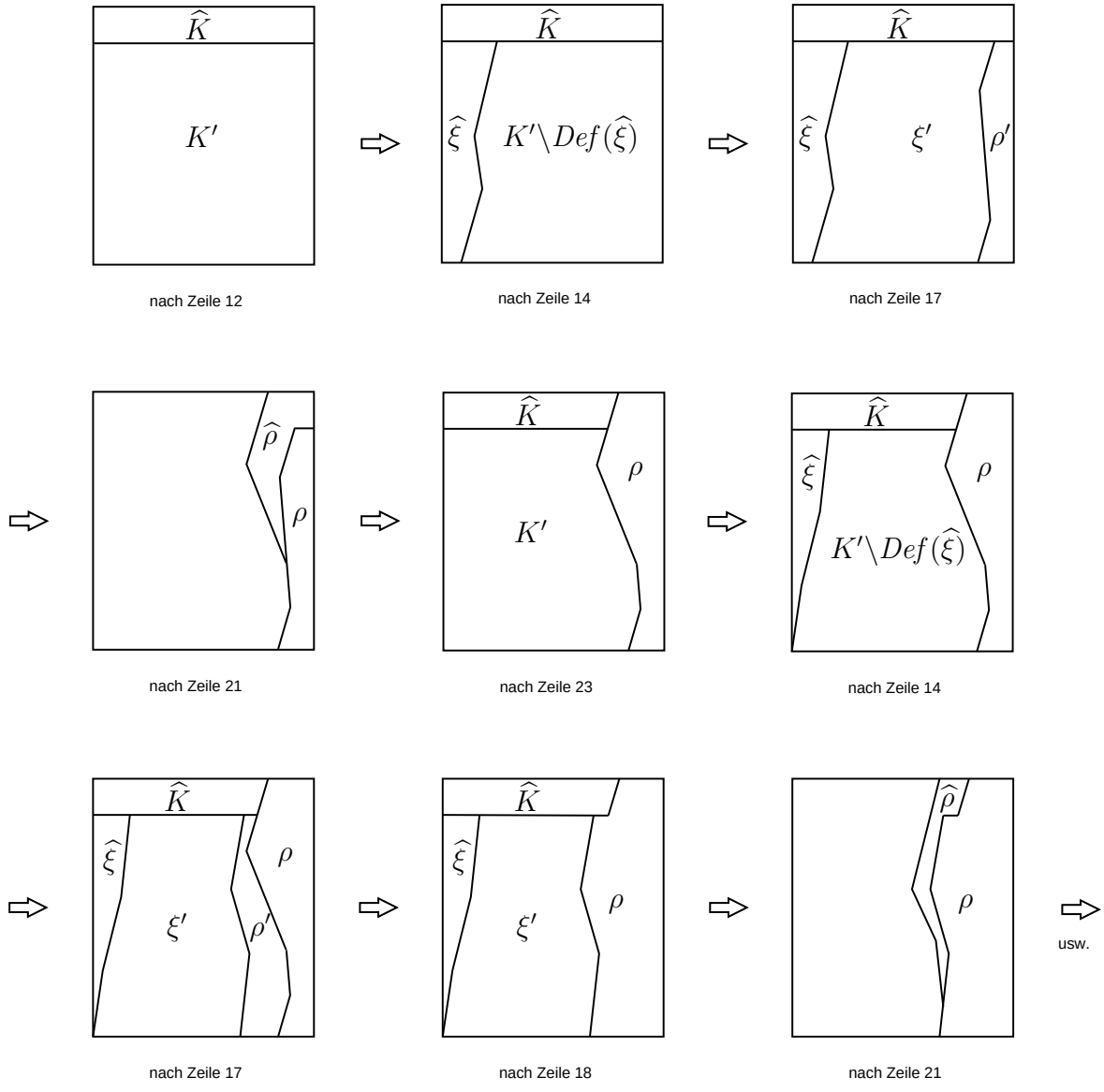

Abbildung 4.16: Die Aufteilung der Knoten bei der repeat-Schleife der synthesize-Funktion wird anschaulich dargestellt.

In Abb. 4.16 wird die Aufteilung der Knoten bei der repeat-Schleife anschaulich

dargestellt, wobei die Bezeichnungen der Variablen der synthesize-Funktion entsprechen.

4.3 Korrektheit

Wir betrachten nun unseren Algorithmus näher, so dass wir die Korrektheit beweisen und die Komplexität berechnen können. Der Kern des Algorithmus ist die Funktion synthesize. Wir zeigen zunächst, welche Vorbedingung beim Aufruf dieser Funktion erfüllt wird.

Lemma 4.3.1
Wird die Funktion synthesize *in Zeile 3 der* main*-Funktion in Abb. 4.7 aufgerufen, dann hat jeder Knoten $v \in K$ mindestens eine ausgehende Kante $e = (v, w)$ mit $w \in K$.*

Beweis: (durch Widerspruch) Würde ein Knoten $v \in K$ keine ausgehende Kante $e = (v, w)$ mit $w \in K$ besitzen, dann würde der Knoten entweder eine Senke in H sein oder ausschließlich die Kanten $e = (v, w)$ mit $w \in (V_H \backslash K)$ besitzen. In beiden Fällen würde der Knoten jedoch zur Gewinnmenge W_0 bzw. W_1 gehören, nachdem die init-Funktion in Zeile 1 ausgeführt wurde. Da die Knotenmenge K mit $K := V_H \backslash (W_0 \cup W_1)$ in Zeile 2 definiert wird, führt dies zum Widerspruch. □

Beim erstmaligen Aufruf der synthesize-Funktion enthält der Spielgraph $H|_K$ also keine Senke. Diese Eigenschaft ist eine relevante Vorbedingung für die synthesize-Funktion, die bei jedem rekursiven Aufruf erfüllt wird. Die Beweisidee ist die gleiche wie beim Lemma 4.3.1.

Satz 4.3.2 (*Invarianz*)
Wird die Funktion synthesize(H) *in Abb. 4.8 mit einem Spielgraphen H ohne Senke ausgeführt, dann enthalten die Spielgraphen, die bei rekursiven Aufrufe der* synthesize*-Funktion als Funktionsparameter benutzt werden, auch keine Senke.*

Beweis: (durch Widerspruch) Es sei angenommen, dass synthesize$(\widetilde{H}_i)$ die erste rekursiv aufgerufene Funktion, so dass der Spielgraph $\widetilde{H}_i$ eine Senke v enthält, wobei

der Index i der Schleifenzähler ist. Dann würde der Knoten v nur die ausgehenden Kanten nach $(V_{\widetilde{H}_{(i-1)}} \backslash V_{\widetilde{H}_i})$ in $\widetilde{H}_{(i-1)}$ besitzen.

Der Knoten v würde dann aber zur Forcemenge nach $(V_{\widetilde{H}_{(i-1)}} \backslash V_{\widetilde{H}_i})$ gehören und wäre bereits vor dem i-ten Aufruf der synthesize-Funktion aussortiert. Dies führt zum Widerspruch. □

Es sind drei Arten von Strategien zu unterscheiden, die in der Funktion synthesize berechnet werden, je nachdem, von welcher Funktion die Strategie bestimmt wird, und zwar von der Funktion force, von der rekursiv ausgeführten synthesize-Funktion oder von der Funktion comp bzw. comp2.

Als Vorbereitung zum Korrektheitsbeweis der Funktion synthesize betrachten wir die Funktion comp bzw. comp2, unter welcher Bedingung sie aufgerufen wird. Wir wollen die Strategie, die durch diese Funktion erstellt wird, bzgl. der synthesize-Funktion charakterisieren.

Wir haben in dieser Arbeit angenommen, dass die Priorität lückenlos definiert ist. Es gilt also, dass $m' = m+1$ in Zeile 11 der synthesize-Funktion in Abb. 4.8. Dies hat zwar auch eine Bedeutung zur Optimierung der Implementierung des Algorithmus, jedoch an dieser Stelle mehr zur Vereinfachung des Korrektheitsbeweises.

Satz 4.3.3
Es sei H ein Spielgraph, der keine Senke enthält. Wenn die comp*-Funktion in Zeile 26 der Funktion* synthesize(H) *ausgeführt wird, dann sind die Vorbedingungen des Satzes 4.2.9 bereits erfüllt.*

Beweis: Die Funktion wird mit den Parametern comp $(\widehat{K}, (Def(\widehat{\xi}) \cup Def(\xi')), i)$ aufgerufen. Die Knotenmenge $\widehat{K}$ wird in Zeile 23 stets verkleinert. Wir verwenden deshalb eine neue Bezeichnung

- $C := (\{v \in V_H \mid p(v) = m\} \backslash Def(\rho))$

für sie, damit wir sie unabhängig von der Veränderung ihrer Größe bezeichnen können. Mit den weiteren Bezeichnungen

- $W_i := Def(\widehat{\xi}) \cup Def(\xi')$ und
- $W_{(1-i)} := Def(\rho)$

gilt die Gleichung $C = (V_H \backslash (W_i \cup W_{(1-i)}))$, da $W_i = (\{v \in V_H \mid p(v) > m\} \backslash Def(\rho))$. In Abb. 4.16 wird die Aufteilung der Knotenmenge V_H in C, W_i und $W_{(1-i)}$ bzgl. $Def(\widehat{\xi})$, $Def(\xi')$ sowie $Def(\rho)$ anschaulich dargestellt.

Es bleibt zu zeigen, dass

(i) jeder Knoten $v \in C$ die höchste Priorität hat,

(ii) jeder Knoten $v \in (C \cap V_i)$ mindestens einen Nachfolgerknoten besitzt, und

(iii) kein Knoten aus C zur Forcemenge nach $Def(\rho)$ für den Spieler $(1-i)$ gehört.

Offenbar gilt $C \subseteq \{v \in V_H \mid p(v) = m\}$ nach der Definition von C. Daraus folgt (i) trivial.

Nach dem Satz 4.3.2 enthält der Spielgraph H keine Senke. Also gilt (ii).

Die Knoten, die zur Forcemenge nach $Def(\rho)$ für den Spieler $(1-i)$ gehören, werden in jedem Durchlauf der Schleife in Zeile 21-23 aus der Menge C herausgenommen. Daraus folgt unmittelbar (iii).

Insgesamt werden alle Vorbedingungen des Satzes 4.2.9 beim Aufruf der comp-Funktion erfüllt. □

Die Vorbedingungen des Satzes 4.2.9 werden auch beim Aufruf der comp2-Funktion erfüllt, wie der folgende Lemma zeigt. Es ist zu beachten, dass die Funktion comp2 im Prinzip dieselbe Funktionalität wie die comp-Funktion hat und lediglich im speziellen Fall mit $W_i = \emptyset$ angewandt wird.

Lemma 4.3.4
Wenn die comp2*-Funktion in Zeile 7 bzw. 8 der Funktion* synthesize(H) *ausgeführt wird, wobei H ein Spielgraph ohne Senke ist, dann sind die Vorbedingungen des Satzes 4.2.9 bereits erfüllt.*

Beweis: Wegen der if-Abfrage in Zeile 6 sind die Modulo-Werte der Priorität aller Knoten gleich. Mit den Bezeichnungen $C := V_H$, $W_i := \emptyset$ und $W_{(1-i)} := \emptyset$ werden alle Vorbedingungen des Satzes 4.2.9 erfüllt, da der Spielgraph H keine Senke enthält. □

Wir zeigen nun die Korrektheit der synthesize-Funktion, wobei die Bezeichnung der Variablen von der Funktion synthesize weiter verwendet wird. Der Beweis erfolgt

durch verallgemeinerte Induktion nach Rekursionstiefe der Aufrufe von synthesize-Funktion.

Satz 4.3.5 *(Korrektheit)*
Es sei ein Spielgraph H gegeben, der keine Senke enthält. Mit $\langle \sigma, \tau \rangle :=$ synthesize(H) wird eine Gewinnstrategie σ bzw. τ in H berechnet, so dass $Def(\sigma)$ bzw. $Def(\tau)$ eine Gewinnmenge für den Spieler 0 bzw. 1 ist, wobei $(Def(\sigma) \cup Def(\tau)) = V_H$ gilt.

Beweis: (durch verallgemeinerte Induktion nach Rekursionstiefe der Aufrufe von synthesize-Funktion)

Induktionsanfang:
Wenn der Graph H leer ist, wird die Zeile 2 ausgeführt und die Behauptung gilt trivial. Wenn der Spielgraph H mindestens einen Knoten enthält und keine Rekursion zur Berechnung stattfindet, bedeutet dies wegen der if-Abfrage in Zeile 6, dass alle Knoten denselben Modulo-Wert der Priorität haben. Alle Spielläufe sind unendlich lang und der Gewinner bleibt gleich. Mit der Funktion comp2 in Zeile 7 bzw. 8 wird eine Gewinnstrategie bestimmt. Nach dem Lemma 4.3.4 und Satz 4.2.9 gilt die Behauptung.

Induktionsannahme:
Die Aussage sei für alle synthesize-Funktionen bewiesen, deren Rekursionstiefe kleiner als n ist. $(*)$

Induktionsschritt:
O.B.d.A. sei n die Rekursionstiefe der Aufrufe von synthesize-Funktion bei synthesize(H). Das heißt, dass die Behauptung für jede rekursiv ausgeführte synthesize-Funktion in synthesize(H) nach der Induktionsannahme richtig sei.

Der Index i wird für den Modulo-Wert der höchsten Priorität aller Knoten verwendet, wie er in Zeile 5 definiert ist. Zum Beweis der Behauptung zeigen wir dann die folgenden Eigenschaften:

1. Die Funktion synthesize(H) terminiert. (*Termination*)
2. Bei Ausführung der repeat-Schleife ist die Strategie ρ stets eine Gewinnstrategie in H und die Knotenmenge $Def(\rho)$ eine Gewinnmenge für den Spieler $(1-i)$. Nachdem ein Knoten $v \in Def(\rho)$ besucht wird, bleibt der weitere Spiellauf innerhalb von $Def(\rho)$, wenn der Spieler $(1-i)$ gemäß der Strategie ρ spielt. (*Invariant*)

3. Nach Ausführung der repeat-Schleife in Zeile 13-25 gibt es keinen Knoten $v \in (V_H \backslash Def(\rho))$, der zur Gewinnmenge für den Spieler $(1-i)$ gehört. (*Maximalität*)

4. In Zeile 26 wird eine Gewinnstrategie ξ für den Spieler i in H konstruiert, so dass $(Def(\xi) \cup Def(\rho)) = V_H$ gilt. (*Dualität*)

Im Folgenden werden die Variablen in Zeile 14-25 so indiziert, dass wir sie bzgl. dem Schleifendurchlauf unterscheiden können. Mit $\widehat{K}_j$ wird beispielsweise die Knotenmenge $\widehat{K}$ in dem j-ten Durchlauf der repeat-Schleife bezeichnet.

1. (*Termination*): Wegen der if-Abfrage in Zeile 20 wird die Abbruchbedingung in Zeile 25 der repeat-Schleife erfüllt, wenn entweder die Knotenmenge $\widehat{K}$ oder $Def(\widehat{\rho})$ leer ist. Da die Größe der Knotenmenge $\widehat{K}$ in Zeile 23 verkleinert wird, bricht die repeat-Schleife spätestens nach dem $|\widehat{K}|$-ten Durchlauf ab. Die rekursiv aufgerufenen synthesize-Funktionen terminieren nach Induktionsannahme. Nach Ausführung der anderen Zeilen, die keine Rekursion enthalten, terminiert die Funktion synthesize(H).

2. (*Invariant*): Die Strategie ρ wird in Zeile 18 und 22 sukzessiv aufgebaut, so dass $\rho = \sum_{j=0}^{n} (\rho'_j + \widehat{\rho}_j)$ für eine bestimmte Zahl $n \leq |\widehat{K}|$ gilt. Wir beweisen die Behauptung durch vollständige Induktion über dem j-ten Schleifendurchlauf.

Induktionsanfang:
Für den Fall $j = 0$ gilt die Behauptung trivial, da $(\rho)_0 = (\rho')_0 = (\widehat{\rho})_0 = 0$ gilt.

Induktionsannahme:
Die Aussage sei richtig für $(j-1)$ mit $j > 0$.

Induktionsschritt:
Die Knotenmenge $Def(\widehat{\xi}_j)$ der Zeile 14 ist die maximale Forcemenge nach $\widehat{K}_j$ für den Spieler i, weshalb keine Kante (v, w) mit $v \in (V_i \cap (K'_j \backslash Def(\widehat{\xi}_j))$ und $w \in (\widehat{K}_j \cup Def(\widehat{\xi}_j))$ existiert. Die in Zeile 16 bzw. 17 berechnete Strategie ρ'_j ist außerdem eine Gewinnstrategie in $H|_{(K'_j \backslash Def(\widehat{\xi}_j))}$ für den Spieler $(1-i)$ wegen if-Abfrage der Zeile 16 und nach $(*)$.

Wenn ein Knoten $v \in Def(\rho'_j)$ in einem Spiellauf besucht wird und der Spieler $(1-i)$ gemäß der Strategie ρ'_j spielt, dann bleibt der weitere Spiellauf in dem Bereich $Def(\rho'_j)$, bis der Spieler i eine Kante (v, w) mit $v \in (V_i \cap Def(\rho'_j))$ und $w \in Def(\rho_j)$ spielt. Die Strategie ρ_j der Zeile 18 ist somit eine Gewinnstrategie

und die Knotenmenge $Def(\rho_j)$ eine Gewinnmenge für den Spieler $(1-i)$. Die Spielläufe, in denen ein Knoten $v \in Def(\rho_j)$ besucht wird, bleiben außerdem in dem Bereich $Def(\rho_j)$.

Die in Zeile 21 berechnete Strategie $\widehat{\rho}_j$ ist eine Forcestrategie nach $Def(\rho'_j)$ für den Spieler $(1-i)$. Die Spielläufe, in denen ein Knoten $v \in Def(\widehat{\rho}_j)$ besucht wird, bleiben somit irgendwann nur in dem Bereich $Def(\rho_j)$. Die Strategie ρ_j der Zeile 22 ist also eine Gewinnstrategie und die Knotenmenge $Def(\rho_j)$ eine Gewinnmenge für den Spieler $(1-i)$.

2. (*Maximalität*): (durch Widerspruch) Würde ein Knoten $v \in (V_H \backslash Def(\rho))$ existieren, der zur Gewinnmenge für den Spieler $(1-i)$ gehört, dann gäbe es einen Knoten $u \in (V_H \backslash Def(\rho))$, der entweder zur Forcemenge nach $Def(\rho)$ oder zur Gewinnmenge in K' für den Spieler $(1-i)$ gehören würde. Wegen der Abbruchbedingung $(Def(\widehat{\rho}) = \emptyset)$ in Zeile 25 sowie der if-Abfrage der Zeile 20 ist der Fall jedoch ausgeschlossen. Dies führt zum Widerspruch.

3. (*Dualität*): Die Strategie ξ der Zeile 26 besteht aus den drei Strategien, die durch die Funktion force in Zeile 14, synthesize in Zeile 16 bzw. 17 sowie comp in Zeile 26 bestimmt werden.

Wegen der Maximalität von $Def(\rho)$ ist die Knotenmenge $(\widehat{K} \cup Def(\widehat{\xi}) \cup Def(\xi'))$ eine Gewinnmenge für den Spieler i. Die in Zeile 16 bzw. 17 berechnete Strategie ξ' ist außerdem eine Gewinnstrategie für den Spieler i in $H|_{(K' \backslash Def(\widehat{\xi}))}$ nach Induktionsannahme $(*)$.

Nach dem Satz 4.3.3 und 4.2.9 ist die durch $\mathsf{comp}(\widehat{K}, (Def(\widehat{\xi}) \cup Def(\xi')), i)$ berechnete Strategie eine Gewinnstrategie, die unabhängig von der Gewinnstrategie in $(Def(\widehat{\xi}) \cup Def(\xi'))$ bestimmt werden kann. Demzufolge ist die Forcestrategie $\widehat{\xi}$ sowie ξ' eine Gewinnstrategie für den Spieler i in H. Insgesamt ist die Strategie ξ eine Gewinnstrategie für den Spieler i, wobei $(Def(\xi) \cup Def(\rho)) = V_H$ gilt. □

Satz 4.3.6 *Der Algorithmus* main *in Abb. 4.7 berechnet eine Gewinnstrategie σ bzw. τ in einem Spielgraphen H für den Spieler* 0 *bzw.* 1*, wobei $Def(\sigma)$ bzw. $Def(\tau)$ eine Gewinnmenge mit $(Def(\sigma) \cup Def(\tau)) = V_H$ ist.*

Beweis: Nach Lemma 4.2.7 ist die Knotenmenge W_0 bzw. W_1, die in Zeile 1 bestimmt wird, eine triviale Gewinnmenge für den Spieler 0 bzw. 1. Die Strategie σ

bzw. τ ist außerdem eine Gewinnstrategie. Da der Spielgraph $H|_K$ nach Lemma 4.3.1 keine Senke enthält, berechnet die Funktion $\mathsf{synthesize}(H|_K)$ nach Satz 4.3.5 eine Strategie σ' bzw. τ', die eine Gewinnstrategie für den Spieler 0 bzw. 1 in $H|_K$ ist. Die Knotenmenge $Def(\sigma')$ bzw. $Def(\tau')$ ist dabei die Gewinnmenge für den Spieler 0 bzw. 1, so dass $(Def(\sigma') \cup Def(\tau')) = K$ gilt.

Spielt der Spieler 0 gemäß der Strategie σ' der Zeile 3 in $Def(\sigma')$, dann bleibt der Spiellauf in $Def(\sigma')$, bis der Spieler 1 eine Kante (v, w) mit $v \in Def(\sigma')$ und $w \in Def(\sigma)$ spielt. Da die Knotenmenge $Def(\sigma)$ in Zeile 1 die maximale Forcemenge nach Senken für den Spieler 0 ist, gewinnt der Spieler 0 das Spiel dadurch, dass er gemäß der Strategie σ weiter spielt.

Wegen Zeile 3 sind die Knotenmengen $Def(\sigma)$ und $Def(\sigma')$ sowie $Def(\tau)$ und $Def(\tau')$ miteinander disjunkt, so dass die Vereinigung der Strategien $\sigma + \sigma'$ bzw. $\tau + \tau'$ in Zeile 5 problemlos durchgeführt werden kann. Also ist die Knotenmenge $(Def(\sigma) \cup Def(\sigma'))$ eine Gewinnmenge und die Strategie $(\sigma + \sigma')$ eine Gewinnstrategie für den Spieler 0. Die Behauptung gilt analog für die Knotenmenge $(Def(\tau) \cup Def(\tau'))$ und die Strategie $(\tau + \tau')$. □

In diesem Abschnitt wurde gezeigt, dass unser Algorithmus eine Gewinnstrategie sowie die Gewinnmenge korrekt berechnet. In dem nächsten Abschnitt wollen wir die Laufzeitkomplexität des Spiel-Algorithmus berechnen.

4.4 Komplexität

Bei den globalen Model-Checking-Algorithmen werden die Knoten nach Alternierungstiefe der Fixpunktformel partitioniert. Die Knoten in einer Partition werden mit 0 bzw. 1 initialisiert, je nachdem, ob der höchste Operator der Fixpunktformel μ oder ν ist. Diese Gruppierung der Knoten ist statisch und bleibt während Ausführung eines Model-Checking-Algorithmus unverändert. Für jeden Knoten steht die Anzahl der Nachfolgerknoten innerhalb einer Partition fest. Der Zähler wird deshalb am Anfang initialisiert und separat gespeichert. Bei der erneuten Initialisierung eines Zählers werden die gespeicherten Informationen wieder verwendet, d.h. zurückkopiert.

Bei unserem Algorithmus wird diese Art der Aufteilung der Knoten beibehalten. Die Alternierungstiefe des Model-Checkings lässt sich in Spielgraphen auf Priorität

sowie Modulo-Wert übertragen. Neben dieser Aufteilung der Knoten werden Partitionen definiert, die eine Einteilung der Gewinnmenge darstellen. Eine Partition kann die Knoten mit unterschiedlichen Prioritäten enthalten, je nachdem, wie die Gewinnmengen eingeteilt werden. Die Größe der Partitionen ändert sich während der Berechnung einer Gewinnstrategie. Das bedeutet, dass die schnelle Initialisierung von Zählern im Sinne der Model-Checking Algorithmen nicht möglich ist. Die Zähler werden jedes Mal neu berechnet, da die Anzahl der Nachfolgerknoten innerhalb einer Partition nicht gleich bleibt.

In diesem Abschnitt wird die asymptotische Laufzeitkomplexität unseres Algorithmus abgeschätzt. Der Hauptteil unseres Algorithmus ist die Funktion synthesize, die Gewinnmenge sowie Gewinnstrategie für die beiden Spieler berechnet. In dem Algorithmus main wird lediglich triviale Gewinnmenge sowie Gewinnstrategie durch die Funktion init berechnet und aussortiert. Dann wird der restliche Spielgraph in die synthesize-Funktion übergeben.

Die Funktion init benötigt eine lineare Laufzeit von $\mathcal{O}(|V_H| + |E_H|)$ nach Lemma lemma-Init-Komplex. In diesem Abschnitt wird deshalb die Laufzeitkomplexität der Funktion synthesize abgeschätzt. Es wird dabei angenommen, dass die Knoten des Spielgraphen beim Einlesen nach ihrer Priorität aufgeteilt werden.

Der Laufzeitaufwand der synthesize-Funktion wird in mehreren Stufen abgeschätzt. Zunächst wird eine Formel erstellt, die stark von der rekursiven Ausführung der synthesize-Funktion abhängt. Dann wird die Formel so umgestellt, dass einige Terme zusammengesetzt werden können, und schließlich dass die komplexe Formel vereinfacht werden kann. Danach wird eine Formel erstellt, die lediglich von der Anzahl der Knoten sowie der Kanten abhängt. In jeder Stufe wird eine Abschätzung gemacht. Jeder Schritt wird bei der Abschätzung genau beobachtet, so dass man herausfinden kann, welcher Schritt einen hohen Laufzeitaufwand verursacht.

Im Folgenden sei H ein Spielgraph, der keine Senke enthält. Es wird angenommen, dass die Knoten des Spielgraphen bereits beim Einlesen in der main-Funktion nach ihrer Priorität sowie Modulo-Wert partitioniert werden.

Die Variablen, die in der Funktion synthesize verwendet werden, werden nach dem Ablauf der Iteration indiziert. Die Variablen in der j-ten Iteration werden mit K_j, $\widehat{K}_j$, K'_j, $\widehat{\xi}_j$, ξ'_j, $\widehat{\rho}_j$, ρ'_j bezeichnet. Es gelten also $V_H = K_0 = \widehat{K}_0 \cup K'_0$, $\widehat{K}_{j+1} \subseteq \widehat{K}_j$ und $K'_{j+1} \subseteq K'_j$. Nach Satz 4.3.5 ist die maximale Durchläufe der repeat-Schleife kleiner als $|\widehat{K}_0|$. Wir bezeichnen mit r die tatsächliche Anzahl der Durchläufe.

Satz 4.4.1 *Die Funktion $t(H)$ sei bzgl.* synthesize(H) *wie folgt rekursiv definiert:*

$$t(H) := \begin{cases} |in(V_H)|, \textit{ falls die Rekursionstiefe der } \textsf{synthesize}\textit{-Funktion 1 ist,} \\ \sum_{j=0}^{r}(P_j + t(H|_{K'_j \setminus Def(\widehat{\xi}_j)}) + Q_j) + |K_0| + |out(\widehat{K}_0)| \textit{ sonst,} \end{cases}$$

wobei $r := \min\{j \mid Def(\widehat{\rho}_j) = \emptyset\}$,

$P_j := |out(K'_j)| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|$,

$Q_j := |out(K_j \setminus Def(\rho'_j))| + |in(Def(\rho'_j))) + |in(Def(\widehat{\rho}_j))|$ *für jedes* $0 \leq j \leq r$

$P_{|\widehat{K}_0|} := Q_{|\widehat{K}_0|} := 0$.

Dann ist $t(H)$ eine asymptotische Laufzeit der Funktion synthesize(H).

Beweis: (durch verallgemeinerte Induktion nach Rekursionstiefe)
Mit $T(syn(H))$ bezeichnen wir die Laufzeit der Funktion synthesize(H). Dann ist zu zeigen, dass $T(syn(H)) \leq t(H)$ für jeden Spielgraphen H gilt, der keine Senke enthält. Wir beweisen dies durch verallgemeinerte Induktion nach Rekursionstiefe der Aufrufe von synthesize-Funktion.

Für die Erkennung der Leerheit von Spielgraphen in Zeile 1 sowie der höchste Priorität m der Knoten in Zeile 4, die Bestimmung des Modulo-Wertes i in Zeile 5, die Kennzeichnung der Knoten K in Zeile 10, die Bestimmung der nächst höchsten Priorität m' mit anderem Modulo-Wert in Zeile 11, die Erkennung der Knotenmenge $\widehat{K}$ sowie K' in Zeile 12 wird eine konstante Laufzeitaufwand benötigt.

Induktionsanfang:
Es wird keine weitere Rekursion ausgeführt, wenn die Prioritäten aller Knoten denselben Modulo-Wert haben. Dann wird die if-Bedingung der Zeile 6 erfüllt und die comp2-Funktion der Zeile 7 bzw. 8 ausgeführt. Nach Lemma 4.2.10 wird eine Laufzeit von $|in(V_H)|$ benötigt. Wegen $T(syn(H)) = |in(V_H)| = t(H)$ gilt also die Aussage für den Induktionsanfang.

Induktionsannahme:
Die Aussage sei für alle synthesize-Funktionen bewiesen, deren Rekursionstiefe kleiner als ein n ist. O.B.d.A. sei n die Rekursionstiefe von synthesize(H). Dann ist die Aussage für alle rekursiv aufgerufene synthesize-Funktionen bei synthesize(H) richtig.

Induktionsschritt:
Die if-Bedingung der Zeile 6 wird nicht erfüllt, so dass die Zeilen 10-28 ausgeführt werden. Zur Ausführung der force-Funktion in Zeile 14 wird eine Laufzeit von

$(|out(K'_j)| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|) =: P_j$ benötigt, da die ausgehenden Kanten von K'_j zum Aufbau der Zähler in einer Laufzeit von $|out(K'_j)|$ gezählt werden und dann eine rückwärts gerichtete Tiefensuche von den Knoten $v \in \widehat{K}_j$ aus durchgeführt wird, wobei eine Laufzeit von $|in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|$ benötigt wird.

Die force-Funktion in Zeile 21 kann analog in einer Laufzeit von $(|out(K_j \backslash Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|) =: Q_j$ ausgeführt werden. Es ist dabei zu beachten, dass die Knotenmenge $\widehat{K}_{|\widehat{K}_0|}$ leer ist, da mindestens ein Knoten aus $\widehat{K}_0$ in jedem Schleifendurchlauf herausgenommen wird. In dem $|\widehat{K}_0|$-ten Schleifendurchlauf werden deshalb die Zeilen 14-15 sowie 21-23 nicht ausgeführt. Daraus folgt $P_{|\widehat{K}_0|} := Q_{|\widehat{K}_0|} := 0$.

Die Strategie ξ'_j sowie ρ'_j wird in Zeile 16 bzw. 17 innerhalb des Bereichs $(K'_j \backslash Def(\widehat{\xi}_j))$ bestimmt. Da die Knoten von $Def(\rho'_j)$ bzw. $Def(\widehat{\rho}_j)$ in Zeilen 19 und 23 aus der Knotenmenge K_j, K'_j sowie $\widehat{K}_j$ nacheinander herausgenommen werden, sind die Knotenmengen $|Def(\rho'_j)|$ und $|Def(\widehat{\rho}_j)|$ miteinander disjunkt. Die Strategie ρ wird in Zeile 18 und 22 sukzessiv aufgebaut. Dafür wird eine Laufzeit von $|Def(\rho'_j)|$ bzw. $|Def(\widehat{\rho}_j)|$ benötigt. Es gilt deshalb

$$|K_0| = |V_H| = \sum_{j=0}^{r}(|Def(\rho'_j)| + |Def(\widehat{\rho}_j)|) + |\widehat{K}_r| + |Def(\widehat{\xi}_r)| + |Def(\xi'_r)|. \qquad (*)$$

In Zeile 26 wird die comp-Funktion ausgeführt, wobei eine Laufzeit von $|out(\widehat{K}_r)|$ benötigt wird. Zum Aufbau der Strategie ξ der Zeile 26 wird also eine Laufzeit von $(|Def(\widehat{\xi}_r)| + |Def(\xi'_r)| + |out(\widehat{K}_r)|)$ benötigt. Insgesamt gilt:

$$\begin{aligned}
T(syn(H)) &= \sum_{j=0}^{r}(P_j + T(syn(H|_{K'_j \backslash Def(\widehat{\xi}_j)})) + Q_j) \\
&\quad + \sum_{j=0}^{r}(|Def(\rho'_j)| + |Def(\widehat{\rho}_j)|) \\
&\quad + |Def(\widehat{\xi}_r)| + |Def(\xi'_r)| + |out(\widehat{K}_r)| \\
\text{(wegen } *) \quad &\leq \sum_{j=0}^{r}(P_j + T(syn(H|_{K'_j \backslash Def(\widehat{\xi}_j)})) + Q_j) + |K_0| + |out(\widehat{K}_r)| \\
\text{(nach I.A.)} \quad &\leq \sum_{j=0}^{r}(P_j + t(H|_{K'_j \backslash Def(\widehat{\xi}_j)}) + Q_j) + |K_0| + |out(\widehat{K}_r)| \\
&\leq \sum_{j=0}^{r}(P_j + t(H|_{K'_j \backslash Def(\widehat{\xi}_j)}) + Q_j) + |K_0| + |out(\widehat{K}_0)| \\
&= t(H).
\end{aligned}$$

Also ist $t(H)$ eine asymptotische Laufzeit der Funktion synthesize(H). □

Unser Ziel ist den Laufzeitaufwand der synthesize-Funktion explizit abzuschätzen. Die obige Formel von $t(H)$ genügt dafür nicht, da sie rekursiv bzgl. der synthesize-Funktion definiert ist. Wir definieren deshalb eine Funktion $\widetilde{t}(H)$, die keine Rekursion enthält, und die auch eine asymptotische Laufzeit der Funktion synthesize(H) ermittelt. Die Laufzeitkomplexität hängt dabei von der Anzahl der Knoten sowie Kanten im Spielgraphen H und von der Anzahl der verschiedenen Prioritäten sowie Modulo-Werten ab.

Bei jedem rekursiven Aufruf der synthesize-Funktion werden eine Menge der Knoten ausgelassen, deren Priorität höchst ist, und die nicht zur Gewinnmenge für den Gegenspieler gehört. Die Rekursionstiefe kann deshalb durch die Alternierung des Modulo-Werte der Prioritäten ausgedrückt werden. Wir führen einige Bezeichnungen ein, damit wir dieses Verhältnis leicht beschreiben können. Zunächst werden die Knoten $v \in V_H$ je nach Priorität $p(v)$ und ihrem Modulo-Wert $(p(v) \bmod 2)$ wie folgt gruppiert:

- $m_1(H) := min\{p(v) \mid v \in V_H\}$ und für jedes $k \geq 1$
- $m_{(k+1)}(H) := min\{p(v) \mid (p(v) > m_k(H)) \wedge ((p(v) \bmod 2) \neq (m_k(H) \bmod 2))\}$
- $L_k(H) := \{v \in V_H \mid (m_k(H) \leq p(v)) \wedge (\exists m_{(k+1)}(H) \Rightarrow (p(v) < m_{(k+1)}(H)))\}$

Wir nennen $L_k(H)$ den k-ten Level und bezeichnen mit $d(H)$ die Anzahl der Levels im Spielgraphen H.

- $d(H) := |\{k \mid L_k(H) \neq \emptyset\}|$

In Zeile 12 der synthesize-Funktion in Abb. 4.8 werden die Knoten $v \in \widehat{K}$ mit höchster Priorität und demselben Modulo-Wert herausgenommen, so dass sie beim rekursiven Aufruf der synthesize-Funktion nicht in Betracht gezogen werden. Die Knotenmenge $\widehat{K}$ entspricht dabei einem Level, dessen Index am kleinsten ist. Die Anzahl der Levels wird deshalb bei jeden rekursiven Aufruf der synthesize-Funktion um 1 verkleinert. Sind Verwechslungen über den Spielgraphen H ausgeschlossen, dann schreiben wir auch abkürzend m_k, M_k und d ohne H.

Die Funktion $\widetilde{t}(H)$, die wir im Folgenden einführen, wird mit Hilfe von $d(H)$ definiert. Für die Knotenmenge sowie Kantenmenge führen wir noch die folgenden zusätzlichen Bezeichnungen zur Verbesserung der Lesbarkeit ein:

- $U_k := \bigcup_{j \geq k} L_j$

- $x_k := |L_k|$
- $y_k := |U_k|$
- $\alpha_k := |out(U_k)|$
- $\beta_k := |in(U_k)|$
- $\gamma_k := |out(L_k)|$

Es ist zu beachten, dass die Gleichungen $\alpha_1 = |E_H| = \beta_1$ und $\alpha_k = \gamma_k + \alpha_{k+1}$ offenbar gelten.

Satz 4.4.2 *Die Funktion $\widetilde{t}(H)$ sei bzgl. der Anzahl der Levels d wie folgt definiert:*

$$\widetilde{t}(H) := \begin{cases} \beta_1, \textit{ falls } d = 1 \textit{ ist} \\ 2 \cdot \sum_{j=1}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1), \textit{ falls } d \geq 2 \textit{ ist} \end{cases}$$

Dann ist $\widetilde{t}(H)$ eine asymptotische Laufzeit der Funktion synthesize(H).

Beweis: (durch verallgemeinerte Induktion nach $d(H)$)
Wir zeigen, dass $t(H) \leq \widetilde{t}(H)$ für jeden Spielgraphen H gilt, der keine Senke enthält.

Induktionsanfang:
Falls $d(H) = 1$ ist, gilt die Behauptung, da $t(H) = |in(V_H)| = \beta_1 = \widetilde{t}(H)$ gilt.

Induktionsannahme:
O.B.d.A. sei $d(H) = n$. Die Aussage sei für die Funktionen synthesize(H') bewiesen, wobei $d(H') < n$.

Induktionsschritt:
Die Formel $t(H)$ in Satz 4.4.1 wird zunächst umgeformt, so dass ein Vergleich zwischen $t(H)$ und $\tilde{t}(H)$ ermöglicht wird. Es gilt

$$\begin{aligned} t(H) &= \sum_{j=0}^{r} (P_j + t(H|_{K'_j \setminus Def(\widehat{\xi}_j)}) + Q_j) + |K_0| + |out(\widehat{K}_0)| \\ &= \sum_{j=0}^{r} P_j + \sum_{j=0}^{r} Q_j + \sum_{j=0}^{r} t(H|_{K'_j \setminus Def(\widehat{\xi}_j)}) + |K_0| + |out(\widehat{K}_0)| \end{aligned}$$

Nach Definition von P_j und Q_j gilt

$$= \sum_{j=0}^{r} (|out(K'_j)| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|)$$

$$+ \sum_{j=0}^{r}(|out(K_j \backslash Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|)$$
$$+ \sum_{j=0}^{r} t(H|_{K'_j \backslash Def(\widehat{\xi}_j)}) + |K_0| + |out(\widehat{K}_0)|$$

Es seien $K'_j := \widehat{K}_j := \emptyset$, $\widehat{\xi}_j := \rho'_j := \widehat{\rho}_j = 0$ und $t(H|_{K'_j \backslash Def(\widehat{\xi}_j)}) := 0$ für alle $r < j$.

Dann gilt wegen $P_{|\widehat{K}_0|} = Q_{|\widehat{K}_0|} = 0$

$$= \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K'_j)| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|)$$
$$+ \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K_j \backslash Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|)$$
$$+ \sum_{j=0}^{|\widehat{K}_0|} t(H|_{K'_j \backslash Def(\widehat{\xi}_j)}) + |K_0| + |out(\widehat{K}_0)|$$

Wir wählen ein $0 \leq s \leq r$ mit $t(H|_{K'_j \backslash Def(\widehat{\xi}_j)}) \leq t(H|_{K'_s \backslash Def(\widehat{\xi}_s)})$ für jedes $0 \leq j \leq r$.

Dann gilt

$$\leq \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K'_j)| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|)$$
$$+ \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K_j \backslash Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|)$$
$$+ \sum_{j=0}^{|\widehat{K}_0|} t(H|_{K'_s \backslash Def(\widehat{\xi}_s)}) + |K_0| + |out(\widehat{K}_0)|$$

Wegen $K'_j \subseteq K'_0$, $(\widehat{K}_j \cap Def(\widehat{\xi}_j)) = \emptyset$, $(\widehat{K}_j \cup Def(\widehat{\xi}_j)) \subseteq K_0$ für jedes $0 \leq j \leq r$ gilt

$$\leq \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K'_0)| + |in(K_0)|)$$
$$+ \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K_j \backslash Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|)$$
$$+ \sum_{j=0}^{|\widehat{K}_0|} t(H|_{K'_s \backslash Def(\widehat{\xi}_s)}) + |K_0| + |out(\widehat{K}_0)|$$

Aus $(K_j \backslash Def(\rho'_j)) \subset K_0$ folgt unmittelbar

$$< \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K'_0)| + |in(K_0)| + |out(K_0)|) + \sum_{j=0}^{(|\widehat{K}_0|-1)}(|in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|)$$
$$+ \sum_{j=0}^{|\widehat{K}_0|} t(H|_{K'_s \backslash Def(\widehat{\xi}_s)}) + |K_0| + |out(\widehat{K}_0)|$$

Wegen $(Def(\rho'_j) \cap Def(\widehat{\rho}_j)) = \emptyset$ und $\bigcup_{j=0}^{r}(Def(\rho'_j) \cup Def(\widehat{\rho}_j)) \subseteq K_0$ gilt weiter

$$\leq \sum_{j=0}^{(|\widehat{K}_0|-1)}(|out(K'_0)| + |in(K_0)| + |out(K_0)|) + |in(K_0)|$$
$$+ \sum_{j=0}^{|\widehat{K}_0|} t(H|_{K'_s \backslash Def(\widehat{\xi}_s)}) + |K_0| + |out(\widehat{K}_0)|$$

$$\leq |\widehat{K}_0| \cdot (|out(K'_0)| + |in(K_0)| + |out(K_0)|) + |in(K_0)|$$
$$+ (|\widehat{K}_0| + 1) \cdot t(H|_{K'_s \setminus Def(\widehat{\xi}_s)}) + |K_0| + |out(\widehat{K}_0)|$$

Aus $|\widehat{K}_0| = |L_1| =: x_1$, $|out(K'_0)| = |out(U_2)| =: \alpha_2$, $|in(K_0)| = |in(U_1)| =: \beta_1$, $|K_0| = |U_1| =: y_1$, $|out(K_0)| = |out(U_1)| =: \alpha_1$ sowie $|out(\widehat{K}_0)| = |out(L_1)| =: \gamma_1$ folgt

$$= x_1 \cdot (\alpha_2 + \beta_1 + \alpha_1) + \alpha_1 + (x_1 + 1) \cdot t(H|_{K'_s \setminus Def(\widehat{\xi}_s)}) + y_1 + \gamma_1$$

Falls $(d = 2)$ ist, gilt nach Induktionsannahme

$$\leq x_1 \cdot (\alpha_2 + \beta_1 + \alpha_1) + \alpha_1 + (x_1 + 1) \cdot \beta_2 + y_1 + \gamma_1$$
$$= x_1\alpha_2 + x_1\beta_1 + x_1\alpha_1 + \alpha_1 + x_1\beta_2 + \beta_2 + y_1 + \gamma_1$$

Wegen $\alpha_2 < \alpha_1$, $\beta_2 < \beta_1$, $y_1 \leq \beta_1$ und $\gamma_1 \leq \alpha_1$ gilt

$$< x_1\alpha_1 + x_1\beta_1 + x_1\alpha_1 + \alpha_1 + x_1\beta_1 + \beta_1 + \beta_1 + \alpha_1$$
$$= 2 \cdot (x_1\alpha_1 + x_1\beta_1 + \alpha_1 + \beta_1)$$
$$= 2 \cdot (\alpha_1 + \beta_1) \cdot (x_1 + 1)$$
$$= 2 \cdot \sum_{j=1}^{1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1)$$
$$= \widetilde{t}(H)$$

Falls $(d > 2)$ ist, gilt nach Induktionsannahme

$$\leq x_1 \cdot (\alpha_2 + \beta_1 + \alpha_1) + \alpha_1 + (x_1 + 1) \cdot 2 \cdot \sum_{j=2}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=2}^{j} (x_k + 1) + y_1 + \gamma_1$$
$$= x_1\alpha_2 + x_1\beta_1 + x_1\alpha_1 + \alpha_1 + y_1 + \gamma_1 + 2 \cdot \sum_{j=2}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1)$$

Wegen $\alpha_2 < \alpha_1$, $y_1 \leq \beta_1$ und $\gamma_1 \leq \alpha_1$ gilt

$$< x_1\alpha_1 + x_1\beta_1 + x_1\alpha_1 + \alpha_1 + x_1\beta_1 + \alpha_1 + 2 \cdot \sum_{j=2}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1)$$
$$= 2 \cdot (x_1\alpha_1 + x_1\beta_1 + \alpha_1) + 2 \cdot \sum_{j=2}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1)$$
$$< 2 \cdot (x_1 + 1)(\alpha_1 + \beta_1) + 2 \cdot \sum_{j=2}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1)$$
$$= 2 \cdot \sum_{j=1}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1)$$
$$= \widetilde{t}(H)$$

Insgesamt gilt die Behauptung. □

Zur Abschätzung der Laufzeit der synthesize-Funktion werden die maximale Anzahl der Durchläufe der repeat-Schleife und die maximale Größe der Knotenmenge K'_j sowie $\widehat{K}_j$ angenommen. Es wird z.B. ignoriert, dass die Größe der Knotenmenge

$(\widehat{K}_j \cup Def(\widehat{\xi}_j))$ sowie $(K_j \backslash Def(\rho'_j))$ in jedem Durchlauf der repeat-Schleife wie folgt immer kleiner wird:

- $(\widehat{K}_0 \cup Def(\widehat{\xi}_0)) \supset (\widehat{K}_1 \cup Def(\widehat{\xi}_1)) \supset \cdots (\widehat{K}_j \cup Def(\widehat{\xi}_j)) \supset \cdots$
- $(K_0 \backslash Def(\rho'_0)) \supset (K_1 \backslash Def(\rho'_1)) \supset \cdots \supset (K_j \backslash Def(\rho'_j)) \supset \cdots$

Außerdem wird die Laufzeit aller rekursiv aufgerufenen synthesize-Funktionen durch die Maximale ersetzt, so dass die Rekursion abgelöst werden kann.

Es ist eine sehr grobe Abschätzung und die synthesize-Funktion benötigt wahrscheinlich einen viel geringeren Laufzeitaufwand, da die Größe der Knotenmenge K'_j in jedem Durchlauf der repeat-Schleife immer kleiner wird und der Faktor $\prod_{k=1}^{j}(x_k+1)$ davon beeinflusst wird. Nach dem ersten Durchlauf der repeat-Schleife wird z.B. mindesten ein Knoten $v \in K'_0$ zum $Def(\rho'_0)$ gehören und aus K'_0 herausgenommen. Ist der Knoten v aus $L_{k'}$, dann wird der Faktor $(x_{k'}+1)$ um 1 kleiner.

Wenn man noch jedes α_j bzw. β_j durch α_1 bzw. β_1 grob abschätzt und dazu mit dem größten Faktor $\prod_{k=1}^{d-1}(x_k+1)$ multipliziert, erhält man auch eine asymptotische Laufzeit von $\mathcal{O}\left(d \cdot |E_H| \cdot \left(\frac{|V_H|+d-1}{d-1}\right)^{d-1}\right)$ für die Funktion synthesize(H), wie der folgende Lemma zeigt:

Lemma 4.4.3 *Es sei d die Anzahl der Levels im Spielgraphen H, der keine Senke enthält, und c eine Konstante. Die Funktion $\overline{t}(H)$ sei wie folgt definiert:*

$$\overline{t}(H) := c \cdot d \cdot |E_H| \cdot \left(\frac{|V_H|+d-1}{d-1}\right)^{d-1}.$$

Dann ist $\overline{t}(H)$ eine asymptotische Laufzeit der Funktion synthesize(H).

Beweis: Wir zeigen, dass $\widetilde{t}(H) \leq \overline{t}(H)$ für jeden Spielgraphen H gilt.

Für den Fall $d(H) = 1$ gilt die Behauptung offensichtlich, da $\widetilde{t}(H) = \beta_1 = |E_H|$.

Für den Fall $d(H) > 1$ gilt

$$\widetilde{t}(H) = 2 \cdot \sum_{j=1}^{d-1}(\alpha_j + \beta_j) \cdot \prod_{k=1}^{j}(x_k+1)$$

Wegen $\alpha_j > \alpha_{j+1}$ und $\beta_j > \beta_{j+1}$ gilt

$$< 2 \cdot \sum_{j=1}^{d-1}(\alpha_1 + \beta_1) \cdot \prod_{k=1}^{d-1}(x_k+1)$$

$$= 2 \cdot (d-1) \cdot (\alpha_1 + \beta_1) \cdot \prod_{k=1}^{d-1} (x_k + 1)$$
$$< 2 \cdot d \cdot (\alpha_1 + \beta_1) \cdot \prod_{k=1}^{d-1} (x_k + 1)$$

Aus Ungleichung fur das arithmetische und das geometrische Mittel folgt

$$\leq 2 \cdot d \cdot (\alpha_1 + \beta_1) \cdot \left(\frac{\sum_{k=1}^{d-1} (x_k+1)}{d-1} \right)^{d-1}$$

Aus $\sum_{k=1}^{d} x_k = |V_H|$ folgt

$$< 2 \cdot d \cdot (\alpha_1 + \beta_1) \cdot \left(\frac{|V_H|+d-1}{d-1} \right)^{d-1}$$

Aus $\alpha_1 = \beta_1 = |E_H|$ ergibt sich unmittelbar

$$= 4 \cdot d \cdot |E_H| \cdot \left(\frac{|V_H|+d-1}{d-1} \right)^{d-1}$$

□

Es muss betont werden, dass die Laufzeit insgesamt sehr grob abgeschätzt wurde. Die Abschätzung wurde in mehreren Stufen $T(syn(H)) \leq t(H) \leq \widetilde{t}(H) \leq \overline{t}(H)$ durchgeführt. In jeder Stufe wird die berechnete Abschätzung immer grober.

Mit der letzten Abschätzung $\overline{t}(H)$ ist es möglich, sie mit der Laufzeitkomplexität von anderen Model-Checking-Algorithmen zu vergleichen, da sie von der Anzahl von Knoten, Kanten sowie Levels abhängt. Es ist jedoch nicht sinnvoll, die Laufzeitkomplexität unseres Algorithmus auf diese Weise abzuschätzen. Die Stärke unseres Algorithmus liegt darin, dass die Spielgraphen bei rekursiv aufgerufenen synthesize-Funktionen immer kleiner werden.

Die Aufteilung der Knoten in Levels ist deshalb wichtig und muss bei der Abschätzung der Laufzeit berücksichtigt werden. Um dies zu verdeutlichen, betrachten wir den Fall näher, dass die Knoten des Spielgraphen beispielsweise in zwei Levels aufgeteilt sind.

Korollar 4.4.4 *Es sei H ein Spielgraph, der keine Senke enthält, und dessen Knoten in zwei Levels L_1 bzw. L_2 aufgeteilt sind. Dann benötigt die Funktion* synthesize(H) *eine Laufzeit von $\mathcal{O}(|E_H| \cdot |L_1|)$.*

Beweis: Nach Satz 4.4.2 gilt die Behauptung offensichtlich. □

Wird die Formel $\overline{t}(H)$ in Lemma 4.4.3 zum Abschätzen der Laufzeit für den Fall mit zwei Levels angewandt, dann erhält man die Laufzeit
$\overline{t}(H) = 4 \cdot 2 \cdot |E_H| \cdot \left(\frac{|V_H|+2-1}{2-1}\right)^{2-1} = 8 \cdot |E_H| \cdot (|V_H| + 1)$.

Wenn man aber die Formel $\widetilde{t}(H)$ von Satz 4.4.2 dafür verwendet, beträgt die Laufzeit

$\widetilde{t}(H) = 2 \cdot (\alpha_1 + \beta_1) \cdot (x_1 + 1) = 2 \cdot (|E_H| + |E_H|) \cdot (|L_1| + 1) = 4 \cdot |E_H| \cdot (|L_1| + 1)$.

Der Unterschied von den beiden Abschätzungen der Laufzeit ist recht groß. Wir untersuchen diesen Fall mit zwei Levels im nächsten Kapitel noch genauer, so dass wir unseren Algorithmus verbessern können.

Wenn man die Aufteilung der Knoten in Levels zum Abschätzung der Laufzeitkomplexität berücksichtigt, erhält man eine Formel, die zum Verglich mit der Laufzeit von anderen Model-Checking-Algorithmen nicht gut geeignet ist. Andererseits wird die Abschätzung zu grob und deshalb unbrauchbar.

Um einen Eindruck zu bekommen, betrachten wir einen Fall, in dem die Knoten sowie Kanten des Spielgraphen gleichmäßig in Levels aufgeteilt sind.

Lemma 4.4.5 *Für jedes $1 \leq k < d$ sei*

- $|L_k| = |L_{k+1}|$
- $|out(L_k)| = |out(L_{k+1})|$
- $|in(L_k)| = |in(L_{k+1})|$

Dann benötigt die Funktion synthesize *eine Laufzeit von $\mathcal{O}(e \cdot z^{d-1})$, wobei $e := |out(L_1)| + |in(L_1)|$ und $z := |L_1| + 1$ gelten.*

Beweis: Nach Voraussetzung gelten $x_k = x_{k+1}$, $\alpha_k = (d - k + 1) \cdot |out(L_1)|$ und $\beta_k = (d - k + 1) \cdot |in(L_1)|$ für jedes $1 \leq k < d$.
Wir zeigen, dass $\widetilde{t}(H) < c \cdot (|out(L_1) + in(L_1)|) \cdot (|L_1| + 1)^{d-1}$ für eine Konstante c gilt. Die Formel $\widetilde{t}(H)$ von Satz 4.4.2 lässt sich wie folgt vereinfachen:

$$\begin{aligned}\widetilde{t}(H) &= 2 \cdot \sum_{j=1}^{d-1} (\alpha_j + \beta_j) \cdot \prod_{k=1}^{j} (x_k + 1) \\ &= 2 \cdot \sum_{j=1}^{d-1} ((d-j+1) \cdot |out(L_1)| + (d-j+1) \cdot |in(L_1)|) \cdot \prod_{k=1}^{j} (|L_1| + 1)\end{aligned}$$

Mit der Bezeichnung $e := |out(L_1)| + |in(L_1)|$ sowie $z := |L_1| + 1$ gilt

$$= 2 \cdot \sum_{j=1}^{d-1}((d-j+1)\cdot e)\cdot z^j$$
$$= 2 \cdot e \cdot \sum_{j=1}^{d-1}(d-j+1)\cdot z^j$$

Wir vereinfachen zunächst den Term $f(z) := \sum_{j=1}^{d-1}(d-j+1)\cdot z^j$ wie folgt:

$$\begin{aligned}(z\cdot f(z) - f(z)) &= -dz + z^2 + z^3 + \cdots + z^{d-1} + 2z^d \\ &= z\cdot(-d + z + z^2 + \cdots + z^{d-2} + 2z^{d-1}) \\ &= z\cdot(z^{d-1} - d + z + z^2 + \cdots + z^{d-2} + z^{d-1}) \\ &= z\cdot(z^{d-1} - d + z\cdot(1 + z + \cdots + z^{d-2})) \\ &= z\cdot(z^{d-1} - d + z\cdot\tfrac{z^{d-1}-1}{z-1}) \\ &= z\cdot(z^{d-1} - d + \tfrac{z}{z-1}\cdot(z^{d-1}-1))\end{aligned}$$

Daraus folgt $f(z) = \frac{z}{z-1}\cdot(z^{d-1} - d + \frac{z}{z-1}\cdot(z^{d-1}-1))$.

Wegen $|L_1| \geq 1$ gilt $z = |L_1| + 1 \geq 2$ und somit auch $\frac{z}{z-1} \leq 2$.
Also gilt $f(z) \leq 2\cdot(z^{d-1} - d + 2\cdot(z^{d-1}-1)) < 6z^{d-1}$

Insgesamt gilt $\widetilde{t}(H) = 2\cdot e\cdot\sum_{j=1}^{d-1}(d-j+1)\cdot z^j \leq 2\cdot e\cdot 6z^{d-1} < 12ez^{d-1}$ □

Der Lemma 4.4.5 zeigt, dass unser Algorithmus mindestens in dem Fall einen wenigeren Laufzeitaufwand als

$$\mathcal{O}(|E_H|\cdot\left(\frac{|V_H|}{d}\right)^{d-1}) = \mathcal{O}(d\cdot e\cdot\left(\frac{d\cdot(z-1)}{d}\right)^{d-1}) = \mathcal{O}(d\cdot e\cdot(z-1)^{d-1})$$

des Model-Checking-Algorithmus in [CKS92] sowie

$$\begin{aligned}\mathcal{O}(d\cdot|E_H|\cdot\left(\frac{|V_H|+d}{d}\right)^{\lceil d/2\rceil}) &= \mathcal{O}(d\cdot d\cdot e\cdot\left(\frac{d\cdot(z-1)+d}{d}\right)^{\lceil d/2\rceil}) \\ &= \mathcal{O}(d^2\cdot e\cdot\left(\frac{z}{d}\right)^{\lceil d/2\rceil})\end{aligned}$$

des Algorithmus von Vöge und Jurdziński [VoJu00] für $d \leq 3$.

Wir haben in diesem Kapitel die wichtigen Definitionen sowie Begriffe über Zwei-Personen-Spiele eingeführt und einen neuen Spiel-Algorithmus entwickelt. Dann wurde die Korrektheit des Algorithmus gezeigt und die Laufzeitkomplexität in der unterschiedlichen Form abgeschätzt.

Eine Abschätzung, die man mit der Laufzeit der anderen Model-Checking- bzw. Spiel-Algorithmen vergleichen kann, war zu grob. Deshalb wurde die Laufzeit für

zwei Fälle betrachtet, um einen Eindruck zu bekommen, wie effizient unser Algorithmus ist. In dem nächsten Kapitel wird diese Betrachtung fortgesetzt und vertieft, so dass wir unseren Algorithmus verbessern können.

Kapitel 5

Optimierungen und Verbesserungen

In diesem Kapitel werden einige Verbesserungen unseres Algorithmus der Abb. 4.8 vorgestellt. In der Funktion synthesize wird der Spielgraph in Teilgraphen zerlegt und dieselbe Funktion für den Teilgraphen rekursiv aufgerufen. Dieser Konzept ist anschaulich und leicht zu verstehen, so dass er relativ einfach in unserem Algorithmus umgesetzt werden konnte.

Im Folgenden wird die Laufzeit der synthesize-Funktion genauer betrachtet, so dass wir die Stellen finden und verbessern, wo einen großen Laufzeitaufwand benötigt wird. Wir setzen zunächst die Berechnung der Laufzeit für den Fall fort, dass die Alternierungstiefe der Formel 2 ist. Es ist zu beachten, dass die Alternierungstiefe als Rekursionstiefe bzw. Level in unserem Spiel-Algorithmus übertragen wurde.

Danach stellen wir auch einige Verbesserungen für höhere Alternierungstiefe vor. Alle Verbesserungen bzw. Optimierungen unseres Algorithmus werden in unserem Tool MetaGame bereits integriert, und zwar in zwei Versionen, je nach Optimierungsgrad, implementiert.

5.1 Optimierung für Alternierungstiefe 2

Im letztem Kapitel wurde die Laufzeitkomplexität unseres Spiel-Algorithmus abgeschätzt. Es wurde dabei festgestellt, dass die Aufteilung der Knoten in Levels

als eine sehr wichtige Komponente zur Abschätzung der Laufzeit auftritt. Im Folgenden wird der Fall behandelt, dass die Knoten eines Spielgraphen in zwei Levels aufgeteilt sind. Da die synthesize-Funktion der Hauptteil unseres Spiel-Algorithmus ist, betrachten wir nur die Funktion synthesize, d.h. der Spielgraph enthält keine Senke, so dass die Funktion init nicht ausgeführt wird. Die Bezeichnungen, die wir im vorherigen Abschnitt eingeführt sind, werden weiter verwendet.

Wir betrachten den Fall genauer, so dass wir feststellen können, an welchen Stellen wie viele Laufzeit die synthesize-Funktion benötigt. Dann werden einige Verbesserungen unseres Spiel-Algorithmus vorgeschlagen, die bereits in die Implementierung integriert sind. Sie wurden in unserem Spiel-Algorithmus nicht vorgestellt, da die Laufzeitkomplexität gleich bleibt.

Wir beginnen mit der Formel von Satz 4.4.1, da sie eine relativ genaue Abschätzung der Laufzeit liefert. Die rekursive Formel ist dabei kein Hindernis mehr, da sie durch eine explizite Formel ersetzt werden kann. Es gilt

$$t(H) = \sum_{j=0}^{r}(P_j + t(H|_{K'_j \setminus Def(\widehat{\xi}_j)}) + Q_j) + |K_0| + |out(\widehat{K}_0)|$$

Nach Definition von P_j und Q_j gilt

$$\begin{aligned}
&= \sum_{j=0}^{r}(|out(K'_j)| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|) + \sum_{j=0}^{r} t(H|_{K'_j \setminus Def(\widehat{\xi}_j)}) \\
&\quad + \sum_{j=0}^{r}(|out(K_j \setminus Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|) \\
&\quad + |K_0| + |out(\widehat{K}_0)|
\end{aligned}$$

Wegen $t(H|_{K'_j \setminus Def(\widehat{\xi}_j)}) = |in(H|_{K'_j \setminus Def(\widehat{\xi}_j)})|$ und $K'_j \setminus Def(\widehat{\xi}_j) = Def(\rho')$ gilt

$$\begin{aligned}
&= \sum_{j=0}^{r}(|out(K'_j)| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|) \\
&\quad + \sum_{j=0}^{r} |in(Def(\rho'_j))| \\
&\quad + \sum_{j=0}^{r}(|out(K_j \setminus Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|) \\
&\quad + |K_0| + |out(\widehat{K}_0)|
\end{aligned}$$

Es ist zu beachten, dass $\widehat{K}_0 = L_1$, $K'_0 = L_2$ und $V_H = K = (L_1 \cup L_2)$ gelten, da der die Knoten des Spielgraphen in zwei Levels L_1 und L_2 aufgeteilt sind.

Der dritte Term $\sum_{j=0}^{r}(|out(K_j \setminus Def(\rho'_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|)$ zeigt den Laufzeitaufwand der force-Funktion der Zeile 21 in Abb. 4.8. Zur Erstellung des Zählers wird die Laufzeit $|out(K_j \setminus Def(\rho'_j))|$ benötigt. Wenn man den Zähler nach

Bedarf nur für die besuchten Knoten erstellt, kann man den unnötigen Laufzeitaufwand sparen, wie die Abb. 5.1 zeigt.

```
force2 (H : Teilgraph, A : Knotenmenge, i : Spieler) : Strategie
 var  ϱ : Strategie
      v.count : Int (* Zähler *)
      worklist : Stack
 1.  for each v ∈ A do (* Init. *)
 2.     v.count := 0
 3.     worklist.push(v) (* Startknoten für rückwärtsgerichtete Tiefensuche *)
 4.  while worklist ≠ ∅ do
 5.     v := worklist.pop ()
 6.     for each u ∈ (in(v) ∩ V_H) do (* Relevante Vorgängerknoten *)
 7.        if (u.count = undef.) then
 8.           if (u ∈ V_i) then (* Der Spieler i wählt den nächsten Knoten *)
 9.              u.count := 1 (* Mindestens eine Kante zur Forcemenge ist zu finden *)
10.           else (* Der Gegenspieler wählt den nächsten Knoten *)
11.              u.count := |succ(u) ∩ (V_H ∪ A)|
                 (* Alle Nachfolgerknoten müssen zur Forcemenge gehören *)
12.        if (u.count > 0) then
13.           u.count := u.count − 1 (* Zählerstand um 1 verringert *)
14.           if (u.count = 0) then (* Der Knoten u gehört zu Forcemenge *)
15.              ϱ(u) := v (* Forcestrategie erweitert *)
16.              worklist.push(u)
17.  return ϱ
```

Abbildung 5.1: Diese Funktion berechnet eine Forcestrategie ϱ sowie die maximale Forcemenge $Def(\varrho)$ nach der Knotenmenge A für den Spieler i, wobei der Zähler nach Bedarf aufgebaut wird.

Die Laufzeit der force2-Funktion beträgt $(|out(Def(\varrho))|+|in(A)|+|in(Def(\varrho))|)$, da der Zähler nur für die Knoten $v \in Def(\varrho)$ initialisiert wird. Der Definitionsbereich $Def(\varrho)$ kann zwar maximal wie V_H groß sein, jedoch wird der Vorteil der Funktion force2 gegenüber force deutlich, wenn man die Funktion bzgl. der synthesize-Funktion betrachtet.

Ersetzt man die force-Funktion in Zeile 21 der synthesize-Funktion durch die force2-

Funktion, beträgt die Laufzeit des dritte Terms

$$\sum_{j=0}^{r}(|out(Def(\widehat{\rho}_j))| + |in(Def(\rho'_j))| + |in(Def(\widehat{\rho}_j))|).$$

Wegen $\sum_{j=0}^{r}(|Def(\rho'_j)| + |Def(\widehat{\rho}_j)|) + |\widehat{K}_r| + |Def(\widehat{\xi}_r)| + |Def(\xi'_r)| = |V_H| = |K_0|$ benötigt man dann eine lineare Laufzeit von $\mathcal{O}(|out(K_0)| + |in(K_0)|)$, d.h. $\mathcal{O}(|E_H|)$.

Der erste Term $\sum_{j=0}^{r}(|out(K'_j)|+|in(\widehat{K}_j)|+|in(Def(\widehat{\xi}_j))|)$ zeigt den Laufzeitaufwand der force-Funktion der Zeile 14 in Abb. 4.8. Ersetzt man diese force-Funktion durch die force2-Funktion, dann beträgt die Laufzeit des ersten Term

$$\sum_{j=0}^{r}(|out(Def(\widehat{\xi}_j))| + |in(\widehat{K}_j)| + |in(Def(\widehat{\xi}_j))|).$$

Die force2-Funktion kann $|L_1|$-Mal aufgerufen werden, da mindesten ein Knoten $v \in L_1$ in jedem Durchlauf der repeat-Schleife herausgenommen wird. Der Definitionsbereich $Def(\widehat{\xi}_0)$ kann maximal so groß wie K'_0 und die Kantenmenge $in(\widehat{K}_0 \cup Def(\widehat{\xi}_0))$ wie E_H sein. Die Laufzeit des ersten Terms beträgt somit $\mathcal{O}(|L_1| \cdot |E_H|)$. Die Laufzeitkomplexität ändert sich also nicht.

Es gelten jedoch $\widehat{K}_0 \supset \widehat{K}_1 \supset \cdots \supset \widehat{K}_r$ und $Def(\widehat{\xi}_0) \supseteq Def(\widehat{\xi}_1) \supseteq \cdots \supset Def(\widehat{\xi}_r)$. Das bedeutet, dass die Anzahl der untersuchenden Kanten bei der force2-Funktion in jedem Durchlauf der repeat-Schleife immer weniger wird.

Der zweite Term $\sum_{j=0}^{r}(|in(Def(\rho'_j))|)$ zeigt den Laufzeitaufwand der rekursiv aufgerufenen synthesize-Funktion in Zeile 16 bzw. 17. Da die Kanten (u, v) mit $v \in Def(\rho'_j)$ ohnehin an dieser Stelle untersucht werden, kann man die Kanten mit $u \in K_j \backslash Def(\rho'_j)$ hierbei selektieren, die dann als Auslöser zur Forcemenge nach $Def(\rho'_j)$ dienen können. Somit kann der Laufzeitaufwand zur Selektierung solcher Kanten in der force2-Funktion, die in Zeile 21 aufgerufen wird, erspart bleiben.

Es ist zu beachten, dass die Laufzeit von $\mathcal{O}(|L_1| \cdot |E_H|)$ besser als $\mathcal{O}(|V_H| \cdot |E_H|)$ von anderen Model-Checking- sowie Spiel-Algorithmen ist, da $|V_H| = |L_1| + |L_2|$ ist. Wenn die Anzahl der Knoten $|L_2|$ gross ist, wird der Unterschied deutlich.

Während die Funktion force durch force2 problemlos ersetzt werden kann, ist die Realisierung der Idee, dass die auslösenden Kanten zur Forcemenge bereits bei Ausführung der synthesize-Funktion zu selektieren, nicht so einfach, wenn die Rekursionsaufrufe tiefer geht. Man braucht eine zusätzliche Verwaltung solcher Kanten. In dem nächsten Abschnitt wird die Integration der Verbesserungen eingeführt, für die weitere Datenstrukturen sowie Funktionen benötigt werden.

5.2 Verbesserungen für höhere Alternierungstiefe

In dem letzten Abschnitt wurde festgestellt, dass der Zähler der für viele Knoten $v \in K_j$ bzw. $v \in K'_j$ wiederholt initialisiert wird. Die Funktion force benötigt also insgesamt einen großen Laufzeitaufwand. Wir wollen in diesem Abschnitt die Redundanzen bei Bestimmung einer Forcestrategie in force-Funktion erkennen und eliminieren. Dafür werden wir zusätzliche Datenstruktur einführen und einige Änderungen bei der synthesize-Funktion vornehmen. Die modifizierte Version der synthesize-Funktion werden wir mit synthesize2 bezeichnen.

Das Konzept sowie die Struktur der Funktion synthesize in Abb. 4.8 bleibt beibehalten. Es werden jedoch die folgenden Punkte verbessert:

1. Die Anzahl der Knoten, für die ein Zähler eingebaut wird, wird reduziert.
2. Erneute Initialisierung des Zählers wird möglichst vermieden.
3. Die Knoten sowie Kanten, die als Auslöser zur Forcemenge dienen, werden selektiert und bei den rekursiv ausgeführten synthesize-Funktionen bewertet, so dass die Suche nach solchen Knoten und Kanten erspart bleibt.
4. Die Zerlegung sowie Verschmelzung der Teilgraphen, für deren Knoten eine Gewinnstrategie bereits berechnet wurde bzw. noch zu berechnen ist, wird durch Partitionierung der Knoten sowie ihre Charakterisierung vereinfacht.

In der Funktion synthesize wird der Spielgraph in Teilgraphen zerlegt, je nachdem, ob ein Knoten zur Gewinnmenge gehört oder nicht. Die Funktion synthesize wird dann für den Teilgraphen rekursiv ausgeführt, der aus den Knoten besteht, die nicht zur Gewinnmenge gehören. Diese Zerlegung wurde in der synthesize-Funktion lokal behandelt, so dass die Knotenmenge mit $\widehat{K}$, $Def(\xi)$, $Def(\rho)$, K' bezeichnet wurde. Wir wollen solche Zerlegung des Spielgraphen global verwalten, so dass sie von jeder rekursiv ausgeführten synthesize-Funktion erkennbar wird.

Die statische Einteilung der Knoten, die in dem Abschnitt 4.4 eingeführt und Level genannt wurde, verwenden wir weiter, so dass die Knoten des Spielgraphen in Levels $L_1, L_2, \ldots, L_r$ verteilt sind. Es ist zu beachten, dass $(p(v) \bmod 2) \neq (p(w) \bmod 2)$ für jedes $1 \leq k < r$ mit $v \in L_k$ und $w \in L_{k+1}$ gilt. Der Einfachheit halber nehmen wir an, dass $(p(v) \bmod 2) = (k \bmod 2)$ für alle $v \in L_k$ gilt.

In diesem Abschnitt wird eine zusätzliche dynamische Einteilung der Knoten erstellt, die mit B_k^i bzw. B_k^{1-i} bezeichnet und Partition genannt wird. Vergleicht man B_k^i bzw. B_k^{1-i} mit den Bezeichnungen der synthesize-Funktion, dann entspricht B_k^i bzw. B_k^{1-i} dem Knotenmenge $(\widehat{K} \cup \mathit{Def}(\xi))$ bzw. $\mathit{Def}(\rho)$. Außerdem wird die Bezeichnung A für die Menge der Knoten verwendet, deren Gewinnstrategie sowie Partitionszugehörigkeit noch zu bestimmen ist.

Die Informationen der Knoten über ihre Level- sowie Partitionszugehörigkeit und Gewinnstrategie werden in dem Knoten gespeichert, so dass sie global zugegriffen werden können. Die Partitionen werden außerdem als Menge definiert, so dass die Knoten einer Partition aufgelistet und die Mengenoperationen leicht durchgeführt werden kann.

Um die Erklärung des Algorithmus zu vereinfachen, werden die folgenden Datenstrukturen sowie Vereinbarungen für einen Spielgraphen H eingeführt:

- $V_H = (V_0 \cup V_1) = L_1 \cup \cdots \cup L_r$
- $B_k := B_k^0 \cup B_k^1$ für jedes $1 \leq k \leq r$
- $V_H = B_1 \cup \cdots \cup B_r \cup A$
- $\bigcup_{j<k} B_j := B_1 \cup \cdots \cup B_{k-1}$
- $\bigcup_{j>k} B_j := B_{k+1} \cup \cdots \cup B_r$
- $v.\mathit{level} = k$ für jedes $v \in L_k$
- $v.\mathit{player} = i$ für jedes $v \in V_i$
- $v.\mathit{partnr} = k$ für jedes $v \in (B_k^0 \cup B_k^1)$
- $v.\mathit{winner} = i$ für jedes $v \in B_k^i$ mit einem $1 \leq k \leq r$
- Die Bezeichnung $\mathit{disproving\_infl}_k$ wird für die Menge der Kanten (u, v) mit $u \in B_k^i$ und $v \in \bigcup_{j \geq k} B_j^{1-i}$ für $i := (k \bmod 2)$ verwendet, die bei Bestimmung der Gewinnstrategie des Knotens u berücksichtigt werden.
- Die Bezeichnung $\mathit{force\_trigger}_k$ wird für die Menge der Knoten $u \in (V_i \cap \bigcup_{j>k} B_j)$ verwendet, die mindestens eine ausgehende Kante (u, v) mit $v \in B_k^i$ haben, wobei $i := (k \bmod 2)$ ist.
- Mit $v(B_k^i \rightarrow B_{k'}^{i'})$ bezeichnen wir, dass der Knoten von B_k^i herausgenommen und in $B_{k'}^{i'}$ hinzugefügt wird.

Für jeden Knoten $v \in V_H$ wird $v.level$ sowie $v.player$ beim Einlesen des Spielgraphen festgelegt und bleibt während Ausführung der Funktion synthesize unverändert. Die Information $v.partnr$ bzw. $v.winner$ über den Spiellauf wird am Anfang nach dem Wert $v.level$ bzw. seinem Modulo-Wert initialisiert und dann nach dem Zwischenergebnis des Ablauf von unserem Algorithmus temporär geändert.

Ist die Operation $v(B_k^i \rightarrow B_{k'}^{i'})$ durchgeführt, dann wird die Information $v.partnr$ sowie $v.winner$ dementsprechend geändert. Die Menge der Kanten $disproving\_infl_k$ bzw. der Knoten $force\_trigger_k$ dient dazu, die Auslöser zur Forcemenge zu speichern. Somit erreichen wir den Laufzeitaufwandes der Berechnung von Forcemengen zu reduzieren.

Die Gewinnstrategie ξ für den Spieler 0 bzw. 1 wird direkt in dem Knoten gespeichert. Zur Unterscheidung, ob und für welchen Spieler $\xi(v)$ eine Gewinnstrategie ist, wird die Einteilung der Gewinnmengen bestimmt.

```
add_disproving_infl (e : Kante)
 var  i, i', k : Int
      u, v : Knoten
 1.  u := e.source_node, v := e.target_node, k := u.level (* u ∈ L_k *)
 2.  if (u.partnr = k) then (* u ∈ B_k *)
 3.     i := (k mod 2), i' := v.winner (* v ∈ B^i' *)
 4.     if (u ∈ B_k^i) then  (* u hat noch die initiale Knotenbewertung *)
 5.        if (i' ≠ i) then  (* v hat eine andere Knotenbewertung *)
 6.           disproving_infl_k := disproving_infl_k ∪ {e}
 7.        else (* v hat auch dieselbe Knotenbewertung *)
 8.           disproving_infl_k := disproving_infl_k \ {e}
```

Abbildung 5.2: Diese Prozedur fügt die Kante $e = (u, v)$ in die Kantenmenge $disproving\_infl_k$ hinzu bzw. entfernt sie aus $disproving\_infl_k$, je nachdem, ob die Knotenbewertung von v bei Bestimmung der Bewertung des Knotens u berücksichtigt werden soll.

Es werden drei Prozeduren bzw. Funktionen eingeführt, die zur Verbesserung der Laufzeit dienen. Wir werden die Einsatzstellen von neuen Prozeduren bzw. Funktionen bzgl. synthesize-Funktion in Abb. 4.8 erläutern, da die die Grundstruktur der synthesize-Funktion erhalten bleibt.

Die erste Prozedur ist add_disproving_infl in Abb. 5.2. Bei Ausführung der force-

Funktion in Zeile 21 der Abb. 4.8 werden die auslösenden Knoten $u \in \widehat{K}$ zur Forcemenge nach $Def(\rho')$ berechnet. Dieser Vorgang kann so gekürzt werden, dass die Kanten (u, v) mit $u \in \widehat{K}$ und $v \in Def(\rho')$ bei der Iteration bzw. rekursiv aufgerufenen Funktionen gesammelt werden.

Die zweite Prozedur ist force3 der Abb. 5.3. Die force-Funktion in Zeile 14 der Abb. 4.8 wird durch diese Prozedur ersetzt. Der Unterschied besteht darin, dass kein Zähler innerhalb der force3-Prozedur erstellt wird. Der Zähler wird lediglich für die Knoten $v \in B_k^i$ und ihre direkte Vorgängerknoten u mit $u \in (V_i \cap \bigcup_{j>k} B_j)$ erstellt.

```
force3 (v : Knoten, k : Int)
 var  i : Int
 1.  i := (k mod 2)
 2.  v(A → B_k^i) (* Ändere die Partitionszugehörigkeit von v *)
 3.  for each e = (u, v) ∈ E do
 4.      if (u.partnr < k) then
 5.          add_disproving_infl(e)
 6.      else (* u ∈ ⋃_{j≥k} B_j *)
 7.          if ((u.player = i) ∧ (u ∈ A)) then
 8.              ξ(u) := v (* Eine Forcestrategie gefunden *)
 9.              force3(u, k)  (* ξ wird erweitert *)
```

Abbildung 5.3: Diese Prozedur berechnet eine Forcestrategie nach dem Knoten v für den Spieler i und addiert sie zu ξ.

Es sei betont, dass B_k^i sowie A global definiert ist und in Zeile 2 der force3-Prozedur geändert wird. Die Funktion force3 wird von den auslösenden Knoten $v \in force\_trigger_k$ zur Forcemenge nach B_k^i aus aufgerufen, wobei $force\_trigger_k \subseteq (V_i \cap \bigcup_{j>k} B_j)$ gilt.

Es ist zu beachten, dass die force3-Funktion keine maximale Forcemenge berechnet. Sobald ein Knoten $u \in (V_{1-i} \cap \bigcup_{j>k} B_j)$ besucht wird, wird die Berechnung nicht fortgesetzt. Um sofort zu entscheiden, ob ein solcher Knoten $u \in (V_{1-i} \cap \bigcup_{j>k} B_j)$ zur Forcemenge gehört oder nicht, braucht man entweder einen Zähler zu erstellen oder alle ausgehende Kanten zu betrachten, was wir vermeiden wollen. Die Entscheidung wird erst in einer rekursiv aufgerufenen Funktion synthesize2(k') getroffen, wenn $u \in L_{k'}$ gilt.

Die dritte Funktion ist adjust_infl der Abb. 5.4. Die force-Funktion in Zeile 21 der

Abb. 4.8 wird durch diese Funktion ersetzt. Die Funktion adjust_infl ist etwas komplizierter als force3 der Abb. 5.3, da sie eine Aktualisierung des Zählers sowie der Knotenmenge *force_trigger* enthält.

```
adjust_infl (e : Kante, k : Int) : Bool
 var  u, v : Knoten, i : Int
 1.  u := e.source_node, v := e.target_node, i := (k mod 2)
 2.  if (u.partnr < k) then
 3.     add_disproving_infl(e)
 4.  else  (* u ∈ (A ∪ ⋃_{j≥k} B_j) *)
 5.     if (u ∈ B_k^i) then
 6.        if (u.player = i) then  (* u ∈ V_i *)
 7.           if (u ∈ L_k) then
 8.              u.count := u.count − 1
 9.              if (u.count = 0) then  (* u hat keinen Nachfolgerknoten v ∈ (A ∪ ⋃_{j≥0} B_j^i) *)
10.                 u(B_k^i → B_k^{1−i})  (* Forcemenge nach B_k^{1−i} erweitert *)
11.                 return true
12.           else  (* u ∉ L_k *)
13.              if (u ∈ force_trigger_k ∧ v ∈ L_k) then
14.                 u.count := u.count − 1
15.                 if (u.count = 0) then  (* Kein Auslöser zur Forcemenge nach B_k^i *)
16.                    force_trigger_k := force_trigger_k \ {u}  (* Korrektur *)
17.        else  (* u.player ≠ i *)
18.           u(B_k^i → B_k^{1−i}), ξ(u) := v  (* Eine Forcestrategie nach B_k^{1−i} gefunden *)
19.           return true
20.     else  (* u ∉ B_k^i *)
21.        if ((u ∈ (A ∪ ⋃_{j>k} B_j^i)) ∧ (u.player ≠ i)) then
22.           u(B_{k+1}^i → B_k^{1−i}), ξ(u) := v  (* Eine Forcestrategie nach B_k^{1−i} gefunden *)
23.           return true
24.  return false
```

Abbildung 5.4: Diese Funktion integriert den Einfluss der Kante $e = (u, v)$ in dem Zählerstand u.count und gib eine Antwort aus, ob die Partitionszugehörigkeit des Knotens u geändert wurde.

Die Funktion adjust_infl besteht aus vielen if-Bedingungen mit einer Kombination der Abfragen über *u.partnr*, *u.level*, *u.winner*, *u.player* sowie *v.level*. Es wird untersucht, ob der Quellknoten u zur Forcemenge nach B_k^{1-i} gehört. Wenn ja, wird

der Knoten in die Partition B_k^{1-i} hinzugefügt und gibt die Antwort true aus.

Im Folgenden werden zwei zusätzliche Prozeduren eingeführt, mit deren Hilfe der eigentliche Vorgang der Berechnung einer Gewinnstrategie vereinfacht dargestellt werden kann.

Die erste Prozedur init_counter der Abb. 5.5 selektiert die Knoten aus A, die die höchste Priorität besitzen, und initialisiert den Zähler für diesen Knoten sowie ihre direkte Vorgängerknoten.

```
init_counter (k : Int)
  var  i : Int
 1.  i := (k mod 2), force_trigger_k := ∅, disproving_infl_k := ∅
 2.  for each v ∈ (A ∩ L_k) do
 3.      v(A → B_k^i)  (* B_k^i := (A ∩ L_k) und A := A\L_k *)
 4.      v.count := 0   (* Zähler wird mit 0 initialisiert *)
 5.  for each (v, w) ∈ E with v ∈ B_k^i do
 6.      if ((v.player = i) ∧ (w ∈ (A ∪ B_k^i)) then
 7.          v.count := v.count + 1  (* Optimistische Annahme *)
 8.  for each e = (u, v) ∈ E with v ∈ B_k^i do
 9.      if (u ∈ A ∧ u.player = i) then
10.          if (u ∉ force_trigger_k) then
11.              u.count := 1  (* Als Auslöser zur Forcemenge nach B_k^i hinzugefügt *)
12.              force_trigger := force_trigger ∪ {u}
13.          else
14.              u.count := u.count + 1  (* Zählerstand erhöht *)
```

Abbildung 5.5: Diese Prozedur partitioniert B_k^i aus A und initialisiert den Zählerstand von $v \in (B_k^i \cup force\_trigger_k)$.

Die zweite Prozedur complete_ws der Abb. 5.6 bestimmt eine Strategie für die Knoten $v \in ((force\_trigger_k \cup (B_k^i \cap L_k)) \cap V_i)$, so dass die Strategie der Knoten $v \in (B_k^i \cap V_i)$ vervollständigt wird. Sie ist mit der Funktion comp der Abb. 4.5 vergleichbar. In complete_ws wird zusätzlich eine Strategie für die Knoten $v \in (force\_trigger_k \cap V_i)$ bestimmt, da diese Knoten nicht durch Festlegung einer Strategie sondern unter Anwendung eines Zählers bewertet werden.

Wir stellen eine Prozedur synthesize2 in Abb. 5.7 vor, in der alle bisher eingeführte Hilfsfunktionen bzw. Prozeduren angewandt werden. Wir werden in dem nächsten

```
complete_ws (k : Int )
 var  e : Kante,  i : Int,  u, v, w : Knoten
 1.  i := (k mod 2)
 2.  for each v ∈ (force_trigger_k ∩ V_i) do
 3.      select e = (v, w) ∈ E with w ∈ (B_k^i ∩ L_k)
 4.      ξ(v) := w
 5.  for each v ∈ (B_k^i ∩ L_k ∩ V_i) do
 6.      select e = (v, w) ∈ E with w ∈ (B_k^i ∪ B_{k+1}^i)
 7.      ξ(v) := w
 8.  for each e = (u, v) ∈ E with v ∈ (B_k ∩ L_k) do
 9.      if (u.partnr < k) then
10.          add_disproving_infl(e)
11.  B_k^i := B_k^i ∪ B_{k+1}^i, B_{k+1}^i := ∅
```

Abbildung 5.6: Diese Prozedur bestimmt eine Gewinnstrategie $\xi(v)$ für die Knoten $v \in ((force\_trigger_k \cup (B_k^i \cap L_k)) \cap V_i)$.

Abschnitt zeigen, dass die Prozedur synthesize2(k) der Abb. 5.7 eine Gewinnstrategie ξ sowie die Gewinnmenge B_k^i bzw. B_k^{1-i} auf den Spielgraphen $H|_{(A \cup \bigcup_{j \geq k} B_j)}$ für den Spieler i bzw. $(1-i)$ bestimmt, wobei die Strategie $\xi|_{(B_k^i \cap V_i)}$ bzw. $\xi|_{(B_k^{1-i} \cap V_{1-i})}$ eine Gewinnstrategie für den Spieler i bzw. $(1-i)$ ist.

Die Einschränkung des Satzes 4.3.2, dass der Spielgraph keine Senken enthält, trifft für die Prozedur synthesize2 nicht zu. Der Grund dafür ist, dass keine maximale Forcemenge hierbei bestimmt wird. Die Senken werden in Zeilen 5-6 sowie 8-12 extra behandelt.

In Abb. 5.8 wird schließlich die Hauptfunktion main2 vorgestellt, in der die Prozedur synthesize2 aufgerufen wird. Es ist zu beachten, dass die Informationen über die Partitionszugehörigkeit von B_k^0, B_k^1, A sowie die Gewinnstrategie ξ in den Knoten gespeichert werden, so dass sie von einem Knoten aus geändert werden können. Die Partitionen sind Mengen, die als globale Variablen definiert werden.

5.3 Korrektheit

In diesem Abschnitt wird die Korrektheit der Prozedur synthesize2 in Abb. 5.7 gezeigt. Der Beweis verläuft ähnlich zu dem Beweis von Satz 4.3.5. Der Unterschied

```
synthesize2 (k : Int)  (* A = V_H \ ⋃_{j<k} B_j *)
 var  worklist: Knotenliste (* Stack *)
      e : Kante,  i : Int,  u, v : Knoten
 1.  if (A = ∅) then break  (* Spielgraph ist leer *)
 2.  else
 3.     init_counter(k), i := (k mod 2) (* B_k^i selektiert und Zähler initialisiert *)
 4.     for each v ∈ B_k^i do
 5.        if ((v.count = 0) ∧ (v.player = i)) then
           (* v ∈ (B_k^i ∩ V_i) hat keinen Nachfolgerknoten w ∈ (A ∪ ⋃_{j≥k} B_j^i) *)
 6.           v(B_k^i → B_k^{1−i}), worklist.push(v)  (* B_k^{1−i} initialisiert *)
 7.     repeat
 8.        while (worklist ≠ ∅) do
 9.           v = worklist.pop()
10.           for each e = (u, v) ∈ E do
11.              if (adjust_infl(e, k) = true) then  (* B_k^{1−i} erweitert *)
12.                 worklist.push(u)
13.        for each v ∈ force_trigger_k do
14.           force3(u, k)  (* B_k^i erweitert *)
15.        synthesize2 (k + 1)  (* A = V_H \ ⋃_{j≤k} B_j *)
16.        B_k^{1−i} := B_k^{1−i} ∪ B_{k+1}^{1−i}, B_{k+1}^{1−i} := ∅
17.        for each e ∈ disproving_infl_k do
18.           if (adjust_infl(e, k) = true) then
19.              worklist.push(e.source_node)
20.        if (worklist ≠ ∅) then
21.           A := (B_{k+1}^i ∪ (B_k^i \ L_k)), B_{k+1}^i := ∅  (* Reset von A und B_{k+1}^i *)
22.           B_k^i := (B_k^i ∩ L_k)  (* Reset von B_k^i *)
23.     until (worklist = ∅)
24.     complete_ws(k)
```

Abbildung 5.7: Diese Prozedur berechnet eine Gewinnstrategie ξ sowie die Gewinnmenge B_k^i bzw. B_k^{1-1} auf den Spielgraphen $H|_{(A \cup \bigcup_{j \geq k} B_j)}$ für den Spieler i bzw. $(1-i)$.

main2(H : *Spielgraph*) : $\langle$*Strategie, Strategie, Knotenmenge, Knotenmenge*$\rangle$
var k : *Int*

1. if $(V_H = \emptyset)$ then $(*$ *Spielgraph ist leer* $*)$
2. return $\langle 0, 0, \emptyset, \emptyset \rangle$
3. else $(*$ *Spielgraph ist nicht leer* $*)$
4. $A := V_H$ $(*$ *Erste neue Partition* $*)$
5. $k := min\{p(v) \mid v \in A\}$ $(*$ *Höchste Priorität* 0 *bzw.* 1 *bestimmt* $*)$
6. synthesize2(k)
7. if $(k = 0)$ then
8. return$\langle \xi|_{(B_0^0 \cap V_0)}, \xi|_{(B_1^1 \cap V_1)}, B_0^0, B_1^1 \rangle$
9. else $(* \; k = 1 \; *)$
10. return$\langle \xi|_{(B_2^0 \cap V_0)}, \xi|_{(B_1^1 \cap V_1)}, B_2^0, B_1^1 \rangle$

Abbildung 5.8: Diese Funktion berechnet eine Gewinnstrategie ξ sowie die Gewinnmengen W_0 bzw. W_1 für den Spieler 0 *bzw.* 1 *auf dem Spielgraphen H.*

liegt jedoch darin, dass die Spielgraphen nicht mehr senkenfrei sind, da wir den Aufbau des Zählers zur Verbesserung der Laufzeit reduziert haben.

Wir zeigen zunächst, welche Eigenschaften bei Ausführung von synthesize2 stets erfüllt sind. Der Einfachheit halber sei i im Folgenden der Modulo-Wert von k, so dass $i := (k \bmod 2)$ gilt. Es ist zu beachten, dass $(1 - i) = ((k + 1) \bmod 2)$ gilt.

Satz 5.3.1 (*Invarianzen*)
Bei Ausführung der Prozedur synthesize2(k) *werden die folgenden Eigenschaften erfüllt:*

1. *Für jedes $u \in B_k$ gilt $u \in \bigcup_{j \geq k} L_j$.*
2. *Für jede Kante $(u, v) \in$ disproving_infl$_k$ gelten $u \in (L_k \cap B_k^i)$ und $v \in \bigcup_{j \geq k} B_j^{1-i}$.*
3. *Es gibt keine Kante (u, v) mit $u \in (V_i \setminus \bigcup_{j<k} B_j)$ und $v \in \bigcup_{j<k} B_j^i$.*
4. *Es gibt keine Kante (u, v) mit $u \in (V_{(1-i)} \setminus \bigcup_{j<k} B_j)$ und $v \in \bigcup_{j<k} B_j^{1-i}$.*
5. *Nach Initialisierung des Zählers mit* init_counter(k) *in Zeile 3 der* synthesize2(k) *der Abb. 5.7 gilt u.count $= |\{(u, v) \in E \mid v \in (A \cup \bigcup_{j \geq k} B_j^i)\}|$ für jeden Knoten $u \in (B_k^i \cap L_k \cap V_i)$.*
6. *Für jedes $u \in$ force_trigger$_k$ gilt u.count $= |\{(u, v) \in E \mid v \in B_k^i\}|$.*

Beweis:

1. Die Partition B_k^i besteht aus den Knoten $v \in (A \cap L_k)$ in Zeilen 2-3 der init_counter-Prozedur der Abb. 5.5, wobei $A = (V_H \setminus \bigcup_{j<k} B_j)$ gilt. Die Partition B_k^{1-k} ist am Anfang leer. Eine Änderung der Partitionszugehörigkeit von den Knoten $v \in (A \cup \bigcup_{j\geq k} B_j)$ gibt es dann

   - $v(A \rightarrow B_k^i)$ in Zeile 2 der force3-Prozedur,
   - $v(B_k^i \rightarrow B_k^{1-i})$ in Zeile 10, 18, 22 der adjust_infl-Funktion,
   - $v(B_{k+1}^i \rightarrow B_k^i)$ in Zeile 11 der complete_ws-Prozedur,
   - $v(B_k^i \rightarrow B_k^{1-i})$ in Zeile 6, $v(B_{k+1}^{1-i} \rightarrow B_k^{1-i})$ in Zeile 16 sowie $v(B_{k+1}^i \rightarrow A)$ in Zeile 21 der synthesize2-Prozedur.

   In allen Fällen bleibt die Teilmengenrelation $B_k = (B_k^0 \cup B_k^1) \subseteq \bigcup_{j\geq k} L_j$ richtig und somit gilt die Behauptung.

2. Für jede Kante $e = (u, v)$ mit $u \in \bigcup_{j<k} B_j$ und $v \in (A \cup \bigcup_{j\geq k} B_j)$ wird die Prozedur add_disproving_infl(e) aufgerufen (in Zeile 5 von force3, in Zeile 3 von adjust_infl und in Zeile 10 von complete_ws). In Zeile 2, 4 sowie 5 der Prozedur add_disproving_infl der Abb. 5.2 werden die Bedingungen überprüft, die mit denen der Behauptung übereinstimmen. Da die Kante e je nach Ergebnis ins *disproving_infl*$_k$ hinzugefügt bzw. aus *disproving_infl*$_k$ entfernt wird, gilt die Behauptung.

3. (Widerspruchbeweis)
   Gäbe es eine solche Kante (u, v), dann würde der Knoten zur Forcemenge nach $B_{k'}^i$ mit $k' < k$ gehören, so dass der Knoten u bereits bei Ausführung der synthesize2(k')-Prozedur aus $(A \cup \bigcup_{j\geq k} B_j)$ entfernt und in $B_{k'}^i$ hinzugefügt wird. Also kann eine solche Kante bei Ausführung der Prozedur synthesize2(k) nicht existieren.

4. Der Beweis verläuft analog zum Beweis der dritten Behauptung.

5. Jedesmal wenn ein Nachfolgerknoten v eines Knotens $u \in (B_k^i \cap L_k)$ seine Partitionszugehörigkeit von $(A \cup \bigcup_{j\geq k} B_j^i)$ zu B_k^{1-i} wechselt, wird der Zählerstand in Zeile 8 bzw. 14 von adjust_infl um 1 vermindert. Die Ausführung der for-Schleife in Zeilen 17-20 sowie der while-Schleife in Zeilen 8-12 von synthesize2 sorgt dafür, dass alle Zählerstände aktualisiert werden. Also gilt die Behauptung.

6. Der Beweis verläuft analog zum Beweis der fünften Behauptung, mit dem Unterschied, dass keine Schleife ausgeführt wird.

□

Falls ein Knoten u aus der Knotenmenge $force\_trigger_k$ in Zeile 16 von adjust_infl entfernt wird, werden die Nachfolgerknoten von u nicht weiter untersucht, da sie nicht zur $force\_trigger_k$ gehören können.

Satz 5.3.2 (*Korrektheit*)
Die Prozedur synthesize2(k) *der Abb. 5.7 bestimmt eine Gewinnstrategie* $\xi|_{(B_k^0 \cap V_0)}$ *bzw.* $\xi|_{(B_k^1 \cap V_1)}$ *auf* $H|_{(B_k^0 \cup B_k^1)}$ *sowie die Gewinnmenge* B_k^0 *bzw.* B_k^1 *für den Spieler* 0 *bzw.* 1, *wobei* $(B_k^0 \cup B_k^1) = V_H \setminus \bigcup_{j<k}(B_j^0 \cup B_j^1)$ *gilt.*

Beweis: (durch verallgemeinerte Induktion nach Rekursionstiefe der Aufrufe von synthesize2-Prozedur)

Induktionsanfang:
Falls der Spielgraph $H|_{(V_H \setminus \bigcup_{j<k} B_j)}$ leer ist, gilt die Behauptung trivial.

Induktionsannahme:
Die Behauptung sei für die Aufrufe der Prozedur synthesize2$(k+1)$ in Zeile 15 der Abb. 5.7 richtig bewiesen.

Induktionsschritt:
Wir zeigen zunächst, dass die Strategie $\xi|_{(B_k^{1-i} \cap V_{(1-i)})}$ bzw. die Partition B_k^{1-i} stets bei Ausführung der synthesize2(k)-Prozedur eine Gewinnstrategie bzw. eine Gewinnmenge für den Spieler $(1-i)$ auf den Spielgraphen $H|_{(A \cup \bigcup_{j \geq k} B_j)}$ ist. Wir zeigen dies durch vollständige Induktion nach Anzahl der Durchläufe der repeat-Schleife der Zeilen 7-23 der Abb. 5.7.

Als Induktionsanfang zeigen wir, dass die Behauptung vor dem Durchlauf der repeat-Schleife gilt. In Zeilen 4-5 werden die Knoten $v \in (B_k^i \cap V_i)$ selektiert, deren Zählerstand 0 ist. Nach Satz 5.3.1.5 haben sie keinen Nachfolgerknoten $w \in (A \cup \bigcup_{j \geq k} B_j^i)$. Sie sind Senken und gehören zur Gewinnmenge für den Spieler $(1-i)$ auf $H|_{(A \cup \bigcup_{j \geq k} B_j)}$. Da die Strategie $\xi|_{(B_k^{1-i} \cap V_{(1-i)})}$ leer ist, gilt die Behauptung offensichtlich.

Als Induktionsschritt betrachten wir, welche Knoten in B_k^{1-i} hinzugefügt werden, und welche Strategie für die Knoten $(B_k^{1-i} \cap V_{(1-i)})$ bestimmt wird, während die repeat-Schleife ausgeführt wird.

Zunächst betrachten wir die durch synthesize2$(k+1)$ bestimmte Knotenmenge B_{k+1}^{1-i} sowie die Strategie $\xi|_{(B_{k+1}^{1-i} \cap V_{(1-i)})}$. Nach Induktionsannahme der verallgemeinerten Induktion ist B_{k+1}^{1-i} eine Gewinnmenge und $\xi|_{(B_{k+1}^{1-i} \cap V_{(1-i)})}$ eine Gewinnstrategie für den Spieler $(1-i)$ auf dem Spielgraphen $H|_{(B_{k+1}^0 \cup B_{k+1}^1)}$.

Die Spielläufe, die mit einem Knoten $u \in B_{k+1}^{1-i}$ beginnen und nach Strategie $\xi|_{(B_{k+1}^{1-i} \cap V_{(1-i)})}$ konstruiert werden, sind entweder endlos lang und bleiben innerhalb B_{k+1}^{1-i}, oder enden nach Satz 5.3.1.3 bzw. 5.3.1.4 mit einem Knoten $v \in V_i$, dessen Nachfolgerknoten nur in $\bigcup_{j \leq k} B_j^{1-i}$ enthalten sind. Da die Knotenmenge B_k^{1-i} bzw. die Strategie $\xi|_{(B_k^{1-i} \cap V_{(1-i)})}$ nach Induktionsannahme der vollständigen Induktion eine Gewinnmenge bzw. Gewinnstrategie für den Spieler $(1-i)$ auf $H|_{(B_k^{1-i} \cap V_{(1-i)})}$ ist, wird die Vereinigung der Partitionen von B_k^{1-i} und B_{k+1}^{1-i} in Zeile 16 als Gewinnmenge gerechtfertigt.

Für die Kanten (u, v) mit $u \in (L_k \cap B_k^i)$ und $v \in \bigcup_{j \geq k} B_j^{1-i}$ wird die Funktion adjust_infl in Zeile 18 der Abb. 5.7 zur Aktualisierung des Zählerstandes $u.count$ aufgerufen. Für die Knoten $u \in V_{(1-i)}$ in Zeile 17 der Abb. 5.4 kann der Spieler $(1-i)$ die Kante (u, v) als Gewinnstrategie wählen, weshalb der Knoten u zur Gewinnmenge B_k^{1-i} gehört. In Zeile 18 der Abb. 5.4 wird der Knoten u in B_k^{1-i} hinzugefügt.

Für die Knoten $u \in V_i$ wird der Zählerstand $u.count$ in Zeile 8 bzw. 14 der Abb. 5.4 um 1 vermindert. Erreicht der Zählerstand eines Knotens $u \in L_k$ Null in Zeile 9, dann bedeutet dies, dass der Knoten u nach Satz 5.3.1.5 keinen Nachfolgerknoten in $(A \cup \bigcup_{j \geq k} B_j^i)$ hat. Wegen 5.3.1.3 sind also alle Nachfolgerknoten von u in $\bigcup_{j \leq k} B_j^{1-i}$ enthalten. Der Knoten gehört deshalb zur Gewinnmenge B_k^{1-i}. In Zeile 10 der Abb. 5.4 wird der Knoten u in B_k^{1-i} hinzugefügt.

Für den Fall $u \in force\_trigger_k$ und $v \in L_k$ in Zeile 13 wird der Zählerstand $u.count$ um 1 vermindert, da die Kante (u, v) keine Begründung mehr bietet, dass der Knoten u zur Forcemenge nach B_k^i gehört. Erreicht der Zählerstand Null in Zeile 15, dann wird der Knoten u aus $force\_trigger_k$ entfernt.

In Zeilen 8-12 der Prozedur synthesize2 der Abb. 5.7 wird die Funktion adjust_infl zur Erweiterung der Gewinnmenge B_k^{1-i} aufgerufen. In Zeilen 21-22 der Funktion adjust_infl der Abb. 5.4 wird der Fall $u \notin (B_k^i \cup force\_trigger_k)$ berücksichtigt. Der

Spieler $(1-i)$ kann die Kante (u,v) in Zeile 22 als Gewinnstrategie nach B_k^{1-i} wählen, weshalb der Knoten zur Gewinnmenge B_k^{1-i} gehört.

Wir zeigen nun, dass die Gewinnmenge B_k^{1-i} nach Ausführung der repeat-Schleife der Zeilen 7-23 in Abb. 5.7 maximal ist. Wäre die Gewinnmenge B_k^{1-i} nicht maximal, dann würde ein Knoten $u \in B_k^i$ existieren, der entweder einen Nachfolgerknoten in B_k^{1-i} hat, falls $u \in V_{(1-i)}$ ist, oder dessen alle Nachfolgerknoten in B_k^{1-i} enthalten sind, falls $u \in V_i$ ist. In beiden Fällen würde der Knoten u bei Ausführung der Funktion adjust_infl in B_k^{1-i} hinzugefügt worden. Dies führt zum Widerspruch.

Aus Maximalität der Gewinnmenge B_k^{1-i} folgt unmittelbar, dass B_k^i eine Gewinnmenge für den Spieler i ist.

Es bleibt noch zu zeigen, dass die Strategie $\xi|_{(B_k^i \cap V_i)}$ eine Gewinnstrategie für den Spieler i auf $H|_{(B_k^0 \cup B_k^1)}$ ist. Wir betrachten, welche Spielläufe konstruiert werden können, wenn der Spieler i nach der Strategie $\xi|_{(B_k^i \cap V_i)}$ spielt.

Für den Fall, dass ein Spiellauf endlich lang ist, gilt die Behauptung nach der ersten Gewinnbedingung in Definition 4.1.4. Wir betrachten also die Spielläufe, die endlos lang sind.

Nach Ausführung der repeat-Schleife der Zeilen 7-23 erhält man zwei Partitionen B_k^i und B_{k+1}^i. Wenn die Spielläufe innerhalb B_{k+1}^i bleiben, gewinnt der Spieler i nach der Induktionsannahme. Dies gilt auch bei $(B_k^i \cap L_k)$, da $i = (k \bmod 2)$. Die Strategie $\xi|_{(B_k^i \setminus L_k)}$ ist eine Forcestrategie nach $(B_k^i \cap L_k)$. Deshalb ist sie auch eine Gewinnstrategie.

Es bleibt die Spielläufe zu betrachten, in denen die Kanten benutzt werden, die durch die Prozedur complete_ws der Abb. 5.6 als Strategie gewählt sind. Werden solche Kanten endlich oft benutzt, bleibt die Behauptung richtig. Falls eine Kante unendlich oft verwendet wird, dann gewinnt der Spieler i nach der zweiten Gewinnbedingung von Definition 4.1.4. Insgesamt gilt somit die Behauptung □

5.4 Diskussion

Bei der Prozedur synthesize2 der Abb. 5.7 werden die folgenden Änderungen als Verbesserung zur synthesize-Funktion der Abb. 4.8 aufgenommen:

1. Der Zähler wird lediglich für die Knoten $v \in (B_k^i \cap L_k) \cup \mathit{force\_trigger}_k$ initialisiert.
2. Der Zähler wird nur einmal vor Ausführung der repeat-Schleife initialisiert und dann beim Durchlauf der Schleife aktualisiert, so dass er weiter verwendet werden kann.
3. Bei Bestimmung einer Forcemenge nach $(B_k^i \cap L_k)$ werden die auslösenden Knoten $v \in \mathit{force\_trigger}_k$ nicht jedesmal erneut berechnet. Die Knoten, die nicht zur $\mathit{force\_trigger}_k$ gehören, werden beim Durchlauf der Schleife mit Hilfe des Zählers herausgefunden und aus $\mathit{force\_trigger}_k$ entfernt.
4. Zur Bestimmung einer Forcemenge nach $\mathit{force\_trigger}_k$ wird kein Zähler verwendet.
5. Zur Bestimmung einer Forcemenge nach B_k^{1-i} werden die Kanten $e \in \mathit{disproving\_infl}_k$ gesammelt, die bei Aktualisierung des Zählers sowie Bewertung des Quellknotens berücksichtigt werden.
6. Zur Bestimmung einer Forcemenge nach B_k^{1-i} wird kein Zähler verwendet.

Ein Hauptbestandteil unseres Algorithmus ist die force-Funktion. Den Laufzeitaufwand zur Bestimmung einer Forcemenge zu reduzieren, wird der Zähler nur an den Knoten eingebaut, wo der Zähler beim Durchlauf der Schleife weiter verwendet werden kann. An den sonstigen Stellen wird die Forcemenge ohne Zähler bestimmt.

Vermutlich kann der Faktor $\bigcup_{j \geq k} L_j$ von α_k sowie β_k in Satz 4.4.2 durch Verbesserungen der letzten Abschnitt auf L_k reduziert werden Für den Fall, dass die Alternierungstiefe 2 ist, kompensiert der Vorteil jedoch mit der Initialisierung sowie Verwaltung der zusätzlichen Datenstrukturen.

In der Forschung wurden Algorithmen für Paritätsspiel entworfen, deren Laufzeitkomplexität subexponentiell beträgt. In [BSV03] wurde z.B. ein randomisierter Algorithmus von Björklund, Sandberg und Vorobyov vorgestellt, dessen Komplexität in $n^{\mathcal{O}(\sqrt{n/\log n})}$ liegt, wobei n die Anzahl der Knoten und d die Alternierungstiefe ist. Es ist zu beachten, dass der Algorithmus erst dann einen Vorteil gegenüber bekannten Model-Checking-Algorithmen sowie Paritätsspielen bzgl. der Komplexität hat, wenn die Alternierungstiefe größer als $\Omega(n^{1/2})$ ist.

Der Algorithmus, der von Jurdziński, Paterson und Zwick in [JPZ06] entwickelt wurde, hat auch eine subexponentielle Komplexität von $n^{\mathcal{O}(\sqrt{n})}$. Für den Fall, dass

jeder Knoten maximal zwei ausgehende Kanten besitzt, beträgt die Komplexität $n^{\mathcal{O}(\sqrt{n/\log n})}$.

Das Spiel-Problem zum Lösen des Paritätsspiels ist für den Bereich der Komplexitätstheorie auch sehr interessant, da das Problem in NP ∩ co-NP [EJS93] und sogar in UP ∩ co-UP [Ju98] liegt. In [EJS93, Ju00, VoJu00, GTW02, BSV03, Ob03] wurde also geforscht, ob und in wie fern die Komplexität dieses Problems verbessert werden kann. Bisher ist kein Algorithmus für Paritätsspiel bekannt, der eine polynomiale Laufzeitkomplexität besitzt.

Die Versuche, einen Algorithmus mit einer besseren Komplexität zu entwickeln, waren bedingt erfolgreich, z.B. für große Alternierungstiefe, oder mit gewisser Einschränkung der Spielgraphen.

In den praktischen Anwendungen des Model-Checkings kommt es selten vor, dass die Alternierungstiefe mehr als 3 beträgt. Die μ-Kalkül-Formeln mit einer hohen Alternierungstiefe sind intuitiv schwer nachvollzuziehen und werden meistens künstlich erzeugt. Die Stärke unseres Algorithmus liegt genau in diesem Bereich. Unser Algorithmus hat zwar einen exponentiellen Laufzeitaufwand, der jedoch genauso gut wie der von anderen bisher bekannten Algorithmen für Paritätsspiel ist.

Kapitel 6

Implementierung und Benutzung des Tools

In diesem Kapitel wird die Implementierung sowie die Benutzung unseres Tools beschrieben, mit dem eine Gewinnstrategie für das Model-Checking-Spiel nach dem Algorithmus vom letzten Kapitel berechnet wird. Das Tool wird zum Zweck entwickelt, die Erfüllbarkeit einer Eigenschaft in einem relativ kleinen System zu testen. Ein globaler Model-Checking-Algorithmus ist zwar eine dafür gut geeignete Technik, jedoch liefert nicht ausreichende Informationen, um eine Fehlerdiagnose über die Erfüllbarkeit der Eigenschaft zu stellen. In unserem Tool wird der Spiel-Algorithmus von dem letzten Kapitel realisiert, so dass man nicht nur die Erfüllbarkeit von Systemeigenschaften als Ergebnis erhält, sondern auch ihre Begründung. Der Anwender kann dem Beweis bzw. der Widerlegung der Erfüllbarkeit einer Systemeigenschaft Schritt für Schritt folgen, nachdem die Begründung der Knotenbewertung als Gewinnstrategie berechnet wurde.

Ein Schwerpunkt unser Tools liegt noch in einer leichten Bedienbarkeit. Die Ein- und Ausgaben werden sowohl textuell als auch graphisch angezeigt und der Anwender erhält Möglichkeit, einen oder mehrere für ihn interessanten Pfade von Spielgraphen auszuwählen, um die Erfüllbarkeit von Systemeigenschaften besser zu verstehen.

Die Eigenschaften des Systems werden mit einer modalen μ-Kalkül formuliert, während das System selbst in einem Kripke-Transitionssystem modelliert wird. Es wird zunächst die Grammatik der Eingabeformel vorgestellt und dann über den Aufbau des Modells sowie Spielgraphen beschrieben.

6.1 Eingabeformel

Die Eingabeformeln werden von einem Parser gelesen, dessen Grammatik der Syntax des μ-Kalküls nach der Definition 2.3.2 entspricht. Der Parser nimmt sowohl die Präfix- als auch Infix-Schreibweise an. Zusätzlich werden die CTL-Formeln akzeptiert und unmittelbar beim Einlesen in eine μ-Kalkül-Formel umgewandelt.

Im Folgenden wird die Grammatik der Eingabeformel in BNF dargestellt, wobei die atomaren Propositionen, die Aktionen und die Variablen mit *AProp*, *Action* und *Var* bezeichnet werden. Der Name einer atomaren Proposition fängt immer mit dem Apostroph an, während der Name einer Aktion bzw. einer Variable mit einem Kleinbuchstabe bzw. mit einem Großbuchstabe anfängt. Die Eingaben lassen sich dadurch leicht unterscheiden. Die Aktionen wie a, b, c werden in eine Menge $\{a, b, c, \ldots\}$ zusammengesetzt. Die Menge der Aktionen bezeichnen wir mit *ActSet*. $\{a, b\}$ bedeutet also $\{a\} \cup \{b\}$ für die Aktionen a und b.

$Prop ::=$ $T \mid F$ *für den logischen Wahrheitswert „*true*“ bzw. „*false*“*
$AProp$ *für eine atomare Proposition*
Var *für eine Variable*
$\tilde{}Prop$ *für die Negation*
$(Prop \mid Prop) \mid (Prop \,\&\, Prop)$ *für die Disjunktion bzw. Konjunktion*
$\langle ActSet \rangle Prop \mid [ActSet] Prop$ *für die modallogischen Formeln*
$max\ Var.(Prop) \mid min\ Var.(Prop)$ *für die Fixpunktformeln*

Alle Variablen in den Formeln sind jeweils mit einer Fixpunktoperator verbunden. Als Platzhalter bei den modallogischen Formeln kann ein Punkt benutzt werden. Die Formel $\langle . \rangle \Phi$ bedeutet, dass irgendeine Aktion ausgeführt und dann die Formel Φ erfüllt wird. Die Formel $[.]\Phi$ bedeutet analog, dass entweder keine Aktion existiert oder wenn irgendeine Aktion existiert, dann die Formel Φ nach aller Aktion erfüllt wird. Die Aktionen entsprechen normalerweise der vorwärts gerichteten Kanten im Modell. Sie können jedoch der rückwärts gerichteten Kanten entsprechen. Solche Aktionen werden mit $!\langle . \rangle !$ und $![.]!$ bezeichnet. Demgemäß werden die folgenden Eingaben noch erlaubt.

$Prop ::= \langle . \rangle Prop \mid [.] Prop \mid$
$!\langle . \rangle !Prop \mid ![.]!Prop \mid !\langle ActSet \rangle !Prop \mid ![ActSet]!Prop$

Die CTL-Formeln sind AF, AG, EF und EG. A bedeutet hierbei „für alle Pfad

gilt...", E für „es gibt einen Pfad, in dem gilt...", F „irgendwann gilt...", und G „es gilt immer...". Die CTL-Formeln werden sich unterscheiden, ob es sich um vorwärts oder rückwärts handelt. $\_F$ steht für vorwärts und $\_B$ für rückwärts. Die erlaubten CTL-Formeln sind also wie folgt:

$$\begin{aligned} \mathit{Prop} ::= {} & \mathit{AG\_F\ Prop} \mid \mathit{EG\_F\ Prop} \mid \mathit{AF\_F\ Prop} \mid \mathit{EF\_F\ Prop} \mid \\ & \mathit{AG\_B\ Prop} \mid \mathit{EG\_B\ Prop} \mid \mathit{AF\_B\ Prop} \mid \mathit{EF\_B\ Prop} \end{aligned}$$

Die obigen Formeln werden unmittelbar beim Einlesen jeweils in eine sematisch äquivalente μ-Kalkül-Formel wie folgt umgewandelt:

- $\mathit{AG\_F\ Prop} \Rightarrow \mathit{max}\, X.([.]X \wedge \mathit{Prop})$
- $\mathit{EG\_F\ Prop} \Rightarrow \mathit{max}\, X.((\langle . \rangle X \vee [.]F) \wedge \mathit{Prop})$
- $\mathit{AF\_F\ Prop} \Rightarrow \mathit{min}\, X.((\langle . \rangle T \wedge [.]X) \vee \mathit{Prop})$
- $\mathit{EF\_F\ Prop} \Rightarrow \mathit{min}\, X.(\langle . \rangle X \vee \mathit{Prop})$

Für die Umwandlung der CTL-Formeln mit der Option $\_B$ (rückwärts) in eine μ-Kalkül-Formel wird $!\langle . \rangle !$ bzw. $![\,.\,]!$ an der Stelle von $\langle . \rangle$ bzw. $[\,.\,]$ benutzt. In dieser Arbeit wird ein Kripke-Transitionssystem als Modell benutzt, so dass die gewünschten Eigenschaften im Modell auch durch Einschränkung der Kanten ausgedrückt werden können. Zum Beispiel bedeutet die Formel $\mathit{AG\_F}\,(\mathit{ActSet}, \mathit{Prop})$, dass die Eigenschaft von Prop immer in allen Pfaden gilt, wobei jede Kante auf den Pfaden mindestens eine Aktion aus ActSet erfüllt wird. Die Umformung solcher CTL-Formeln in eine μ-Kalkül-Formel erfolgt, wenn man den Platzhalter obiger μ-Kalkül-Formeln durch ActSet ersetzt.

- $\mathit{AG\_F}\,(\mathit{ActSet}, \mathit{Prop}) \Rightarrow \mathit{max}\, X.([\mathit{ActSet}]X \wedge \mathit{Prop})$
- $\mathit{EG\_F}\,(\mathit{ActSet}, \mathit{Prop}) \Rightarrow \mathit{max}\, X.((\langle \mathit{ActSet} \rangle X \vee [\mathit{ActSet}]F) \wedge \mathit{Prop})$
- $\mathit{AF\_F}\,(\mathit{ActSet}, \mathit{Prop}) \Rightarrow \mathit{min}\, X.((\langle \mathit{ActSet} \rangle T \wedge [\mathit{ActSet}]X) \vee \mathit{Prop})$
- $\mathit{EF\_F}\,(\mathit{ActSet}, \mathit{Prop}) \Rightarrow \mathit{min}\, X.(\langle \mathit{ActSet} \rangle X \vee \mathit{Prop})$

Die CTL-Formeln mit dem *until*- bzw. *weak_until*-Operator werden mit SU bzw. WU eingegeben. Mit der Optionen A, E, $\_F$ und $\_B$ hat man dann die folgenden CTL-Formeln.

$$
\begin{aligned}
Prop ::= \; & ASU\_F\,(Prop_1, Prop_2) \mid ESU\_F\,(Prop_1, Prop_2) \mid \\
& AWU\_F\,(Prop_1, Prop_2) \mid EWU\_F\,(Prop_1, Prop_2) \mid \\
& ASU\_B\,(Prop_1, Prop_2) \mid ESU\_B\,(Prop_1, Prop_2) \mid \\
& AWU\_B\,(Prop_1, Prop_2) \mid EWU\_B\,(Prop_1, Prop_2)
\end{aligned}
$$

Die obige CTL-Formeln werden unmittelbar beim Einlesen vom Parser in eine semantisch äquivalente μ-Kalkül-Formel umgewandelt.

- $ASU\_F\,(Prop_1, Prop_2) \Rightarrow min\; X.((((\langle . \rangle T \wedge [.]X) \wedge Prop_1) \vee Prop_2)$
- $ESU\_F\,(Prop_1, Prop_2) \Rightarrow min\; X.((\langle . \rangle X \wedge Prop_1) \vee Prop_2)$
- $AWU\_F\,(Prop_1, Prop_2) \Rightarrow max\; X.(([.]X \wedge Prop_1) \vee Prop_2)$
- $EWU\_F\,(Prop_1, Prop_2) \Rightarrow max\; X.((((\langle . \rangle X \vee [.]F) \wedge Prop_1) \vee Prop_2)$

Die Eingabe einer Formel kann dadurch erfolgen, dass man „Property“ in dem Hauptmenü wählt und anschließend die Formel entweder textuell angibt („New property“) oder aus einer Datei lädt („Choose property“ bzw. „Load property“).

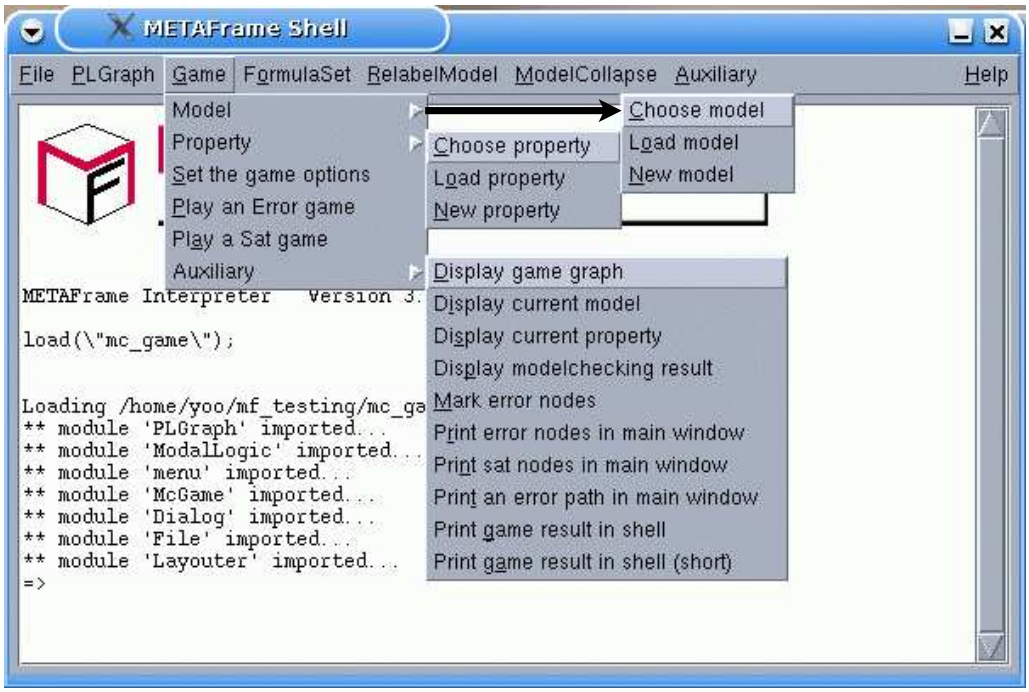

Abbildung 6.1: Hauptmenü im METAFrame-Shell

Will man die gewünschte Formel selbst eingeben, dann wählt man das Untermenü „New property“. Standardmäßig wird die zuletzt geladene Formel in dem

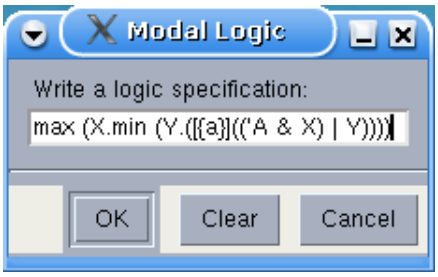

Abbildung 6.2: Eingabefenster

Eingabefenster wie in Abbildung 6.2 gezeigt, so dass man sie leicht editieren kann. Man kann aber auch die in einer Datei gespeicherte Formel laden.

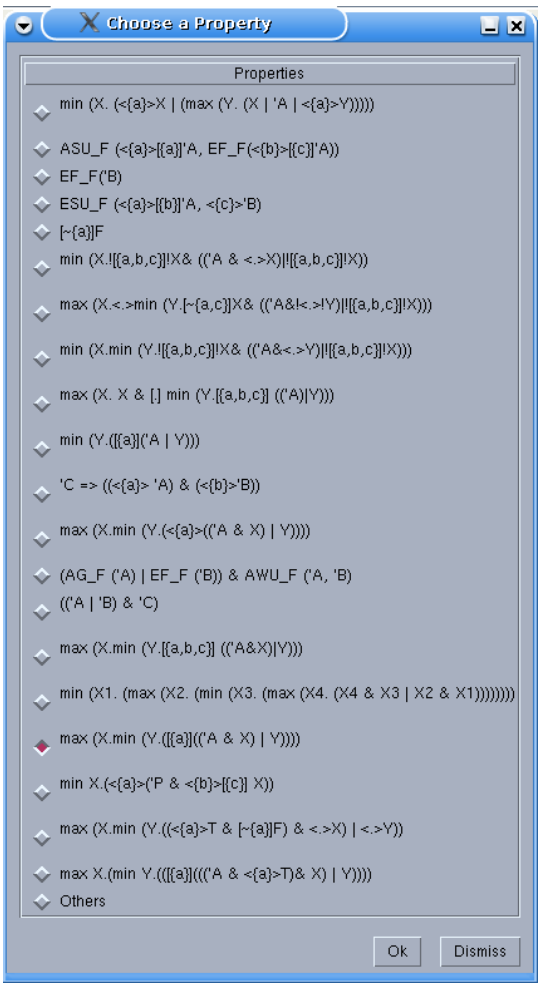

Abbildung 6.3: Auswahl einer Formel

Wählt man das Untermenü „Load property“ dann wird die Liste der Dateien im aktuellen Verzeichnis gezeigt, deren Name mit „.prop“ endet. Normalerweise kann man aber mit den Dateinamen nicht wissen, welche Formel in den Dateien gespeichert sind.

Wenn man die in den Dateien gespeicherten Formeln zunächst sehen und davon eine Formel wählen möchte, kann man das Untermenü „Choose property“ wählen. Man erhält dann eine Liste der Formeln wie in Abbildung 6.3. Wird eine syntaktisch nicht korrekte Formel eingegeben oder gewählt, dann wird die Fehlermeldung vom Parser sowohl im Hauptfenster als auch im Shell ausgegeben, wobei die Fehlerstelle mit einem Hacken textuell markiert gezeigt wird.

6.2 Eingabe des Modells

In dieser Arbeit wird ein Kripke-Transitionssystem als Modell verwendet. Um ein Modell einzugeben, wählt man „Model“ im Hauptmenü. Dann hat man die Auswahl „Choose model“, „Load model“, und „New model“, wie in Abbildung 6.1 zu sehen ist. In Abbildung 6.4 wird das Modell von Abbildung 3.1 gezeigt, das in einem PLGraph-Editor von METAFrame geladen wird.

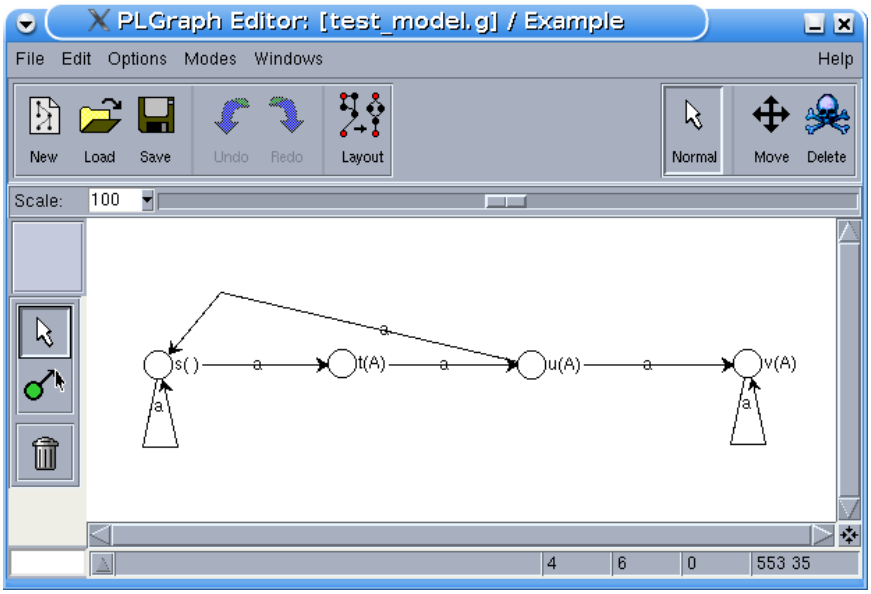

Abbildung 6.4: Modell

Ein Knoten kann mit der Tastenkombination *Ctrl+M1*(linke Mousetaste) hinzugefügt werden. Um zwei Knoten mit einer Kante zu verbinden, braucht man den Quellknoten mit der linken Mousetaste (*M1*) und dann den Zielknoten mit der Tastenkombination *Ctrl+M1* anzuklicken. Um eine Proposition an einem Knoten oder eine Aktion an einer Kante anzuhängen, wird das zugehörige Label geändert, das man durch Auswahl des Menüs „Windows“ und dann „Information Window“ in dem PLGraph-Editor erhält. Das Label von Knoten bzw. Kanten wird GameModelNodeLabel bzw. GameEdgeLabel genannt.

6.3 Aufbau der Spielgraphen

In diesem Abschnitt wird erklärt, wie ein Spielgraph aus einer modalen μ-Kalkül-Formel und einem Modell aufgebaut wird. Nach der Eingabe einer Formel sowie eines Modells wird eine graphische Struktur wie in Abb. 6.6 konstruiert, die zum Aufbau eines Spielgraphen verwendet wird.

Für die Eingabeformel ϕ wird ein Transisitionssystem $T = (S, Act \cup \{\lambda\}, \rightarrow)$ erstellt, wobei S die Menge der Teilformeln von ϕ, Act eine Menge der Aktionen, die in ϕ auftreten, λ keine echte Aktion, und $\rightarrow \subseteq S \times \{Act \cup \lambda\} \times S$ eine Transitionsrelation ist, die nach den folgenden Abwicklungsregeln bestimmt wird:

- $A \vdash T$ und $A \vdash F$ für eine atomare Proposition A
- $(\phi \vee \psi) \vdash \phi$ und $(\phi \vee \psi) \vdash \psi$
- $(\phi \wedge \psi) \vdash \phi$ und $(\phi \wedge \psi) \vdash \psi$
- $\langle a \rangle \phi \vdash_a \phi$ und $[a]\phi \vdash_a \phi$ für eine Aktion $a \in Act$
- $max\, X.\phi(X) \vdash \phi(X)$ und $X \vdash max\, X.\phi(X)$
- $min\, X.\phi(X) \vdash \phi(X)$ und $X \vdash min\, X.\phi(X)$

Die Transition $\phi \xrightarrow{a} \psi$ entspricht der Abwicklung $\phi \vdash_a \psi$. Für die Abwicklung $\phi \vdash \psi$ ohne Beschriftung wird die Transition $\phi \xrightarrow{\lambda} \psi$ benutzt, wobei λ ein Platzhalter ist. Die Transition $\phi \xrightarrow{\lambda} \psi$ bedeutet also, dass keine Aktion durchgeführt wird.

Wir betrachten die Formel $\phi = \nu X.\mu Y.[a]((A \wedge X) \vee Y)$, die im Beispiel 3.3.2 behandelt wurde. Der Fixpunktoperator ν bzw. μ wird dabei mit max bzw. min bezeichnet, wie die Eingabeformel für unseren Parser geschrieben wird. In Abb. 6.5 wird das Transitionssystem für die Formel $\phi = max\, X.min\, Y.[a]((A \wedge X) \vee Y)$ dargestellt, das nach den Abwicklungsregeln erstellt wird.

Die Abwicklung von einer Fixpunktvariable zur Fixpunktformel wird durch Zusammenfalten an die Fixpunktformel dargestellt, so dass die weitere Abwicklungen an Untergraphen erspart bleibt. In Abb. 6.5 wird solche Abwicklung als Kurve angezeigt.

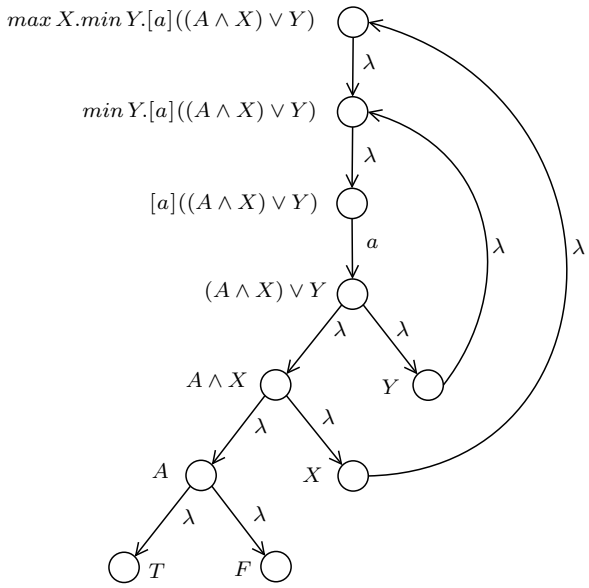

Abbildung 6.5: Transitionssystem für die Formel $max\, X.min\, Y.[a]((A \wedge X) \vee Y)$

Wir bezeichnen das Transitionssystem aus einer Formel ϕ mit $T = (S_\phi, Act_\phi, \rightarrow_\phi)$ und das Kripke-Transitionssystem als Modell mit $M = (S_M, Act_M, AP_M, \rightarrow_M, L)$, wobei $L : S_M \rightarrow 2^{AP_M}$ eine Markierungsfunktion ist, die jedem Zustand eine Menge von atomaren Propositionen zuordnet.

Wir definieren ein Produkt-Transitionssystem $P = M \otimes T$, wobei

$$P = (S_M \times S_\phi, Act_M \cup \{\lambda\}, \rightarrow)$$

und für alle $s_M, s'_M \in S_M$, $s_\phi, s'_\phi \in S_\phi$ und $a \in (Act_M \cup \{\lambda\})$ gilt:

- $(s_M, s_\phi) \xrightarrow{a} (s'_M, s'_\phi) \Leftrightarrow s_M \xrightarrow{a} s'_M \wedge s_\phi \xrightarrow{a} s'_\phi$ für jede Aktion $a \in (Act_M \cap Act_\phi)$
- $(s_M, s_\phi) \xrightarrow{\lambda} (s_M, s'_\phi) \Leftrightarrow s_\phi \xrightarrow{\lambda} s'_\phi$ für $s_\phi \notin AP_M$
- $(s_M, s_\phi) \xrightarrow{\lambda} (s_M, T) \Leftrightarrow s_\phi \xrightarrow{\lambda} T$ für $s_\phi \in L(s_M)$
- $(s_M, s_\phi) \xrightarrow{\lambda} (s_M, F) \Leftrightarrow s_\phi \xrightarrow{\lambda} F$ für $s_\phi \in (AP_M \backslash L(s_M))$

Die Transitionen eines Produkt-Transitionssystems sind entweder ein synchrones Produkt für eine Aktion a, oder ein Produkt, in dem der Zustand des Modells unverändert bleibt, aber die Formel sich durch die Dummy-Aktion λ ändert. Nach der obigen Definition kann ein Produkt-Transitionssystem durch eine gegebene Formel und ein Modell eindeutig bestimmt werden.

In Abbildung 6.6 wird ein Produkt-Transitionssystem dargestellt, das aus dem Transitionssystem in Abbildung 6.5 und dem Modell in Abbildung 6.4 sowie 3.1 konstruiert werden kann. Die Beschriftung λ der Dummy-Kanten wird dabei ausgelassen.

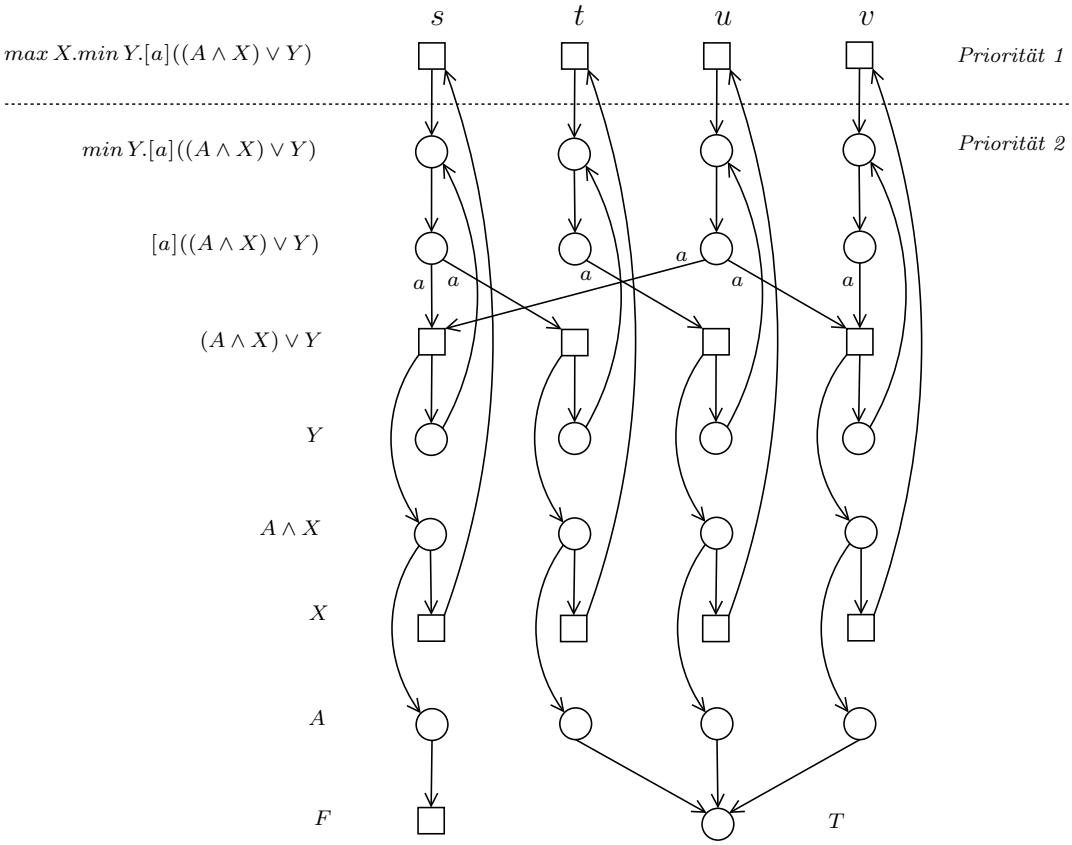

Abbildung 6.6: Produkt-Transitionssystem

Das Produkt-Transitionssystem kann als Spielgraph $G = (V_0, V_1, E, p)$ benutzt werden. Zur Knotenmenge V_0 des Spielers 0 gehören die Knoten der Form $(s_M, min\, X.\psi(X))$, $(s_M, \psi_1 \wedge \psi_2)$, $(s_M, [a]\psi)$ sowie (s_M, T) und zur Knotenmenge V_1 des Spielers 1 die Knoten der Form $(s_M, max\, X.\psi(X))$, $(s_M, \psi_1 \vee \psi_2)$, $(s_M, \langle a\rangle\psi)$ sowie (s_M, F), wobei s_M ein Knoten im Modell ist. Die Knoten des Spielers 0 werden in Abb. 6.6 als Kreise dargestellt und die Knoten des Spielers 1 als Quadrate.

Die Knoten (s_M, T) und (s_M, F) sind jeweils eine Senke. Wenn sie in einem Spiel besucht werden, wird das Spiel beendet und der Spieler, der an dem Knoten spielen soll, verliert das Spiel nach der ersten Gewinnbedingung in Definition 4.1.4.

Die Prioritäten können nach der Struktur der Fixpunktformel bestimmt werden. Die kleinste Fixpunktformel $min\, X.\psi(X)$ erhält eine gerade Zahl und die größte Fixpunktformel $max\, X.\psi(X)$ eine ungerade Zahl als Priorität. Bei der Alternierung

der Fixpunktoperatoren wie in der Formel $max\, X.min\, Y.\psi(X,Y)$ erhält die äußere Formel $max\, X.min\, Y.\psi(X,Y)$ höhere Priorität als die innere Formel $min\, Y.\psi(X,Y)$. Um die Anzahl der Prioritäten möglichst klein zu haben, wird die Formel nach den Vorschlägen in [Kl94] analysiert. Die Anzahl der Prioritäten entspricht somit genau der reduzierte Alternierungstiefe.

Das in Abbildung 6.6 konstruierte Produkt-Transitionssystem kann zwar ohne jegliche Änderung als Spielgraph benutzt werden, enthält jedoch redundante Knoten und Kanten. Lässt man solche Knoten und Kanten im Spielgraphen stehen, dann hat man als Nachteil, dass die graphische Darstellung von Spielgraphen zu groß und unübersichtlicher wird.

In unserem Tool werden deshalb die redundanten Knoten und Kanten der Spielgraphen eliminiert. Die Reduzierung der redundanten Knoten sowie Kanten in den Spielgraphen soll aber vor allem dazu beitragen, dass der Anwender das Spiel-Ergebnis leicht verstehen kann. Werden die Spielgraphen zu kompakt dargestellt, dann kann es schwierig werden, die Spielläufe nachzuvollziehen.

Die Anforderung, wie ausführlich oder kompakt die Spielgraphen dargestellt werden sollen, ist davon abhängig, je nachdem wieviel Kenntnisse über die modale μ-Kalkül und die Konstruktion der Spielgraphen der Anwender besitzt. In unserem Tool wird dies durch die Option der Darstellungsart („detailliert“ bzw. „kompakt“) der Spielgraphen gesteuert. Es werden zunächst drei Arten der Reduzierung vorgestellt, die standardmäßig durchgeführt wird. Danach werden optionale Reduzierungen eingeführt.

Die Kürzungsregeln, die im Folgenden erläutert werden, beziehen sich lediglich auf die graphische Darstellung des Spielgraphen. Zur Berechnung unseres Spiel-Algorithmus wird der Spielgraph standardmäßig möglichst klein aufgebaut. Dies hat jedoch keine Wirkung auf die Verbesserung der Laufzeit.

Die erste Kürzungsregel bezieht sich auf der Fixpunktvariablen. Die Knoten (s,X), (t,X), (u,X), (v,X), (s,Y), (t,Y), (u,Y) und (v,Y) in Abbildung 6.6 werden eliminiert und ihre Vorgängerknoten zeigen direkt auf die Nachfolgerknoten. Solche Knoten haben nur einen Vorgänger- und Nachfolgerknoten und durch Eliminierung der Knoten werden die Abwicklung einer Teilformel zur Fixpunktvariable und die Transformation der Fixpunktvariable zur entsprechenden Fixpunktformel zusammengefasst.

Die zweite Kürzungsregel bezieht sich auf die modalen Operatoren. Die Knoten

mit einer Formel, deren höchste Operator eine modale Operator $\langle . \rangle$ bzw. $[.]$ ist, werden eliminiert, wenn sie einen Vorgängerknoten haben. Beispielsweise wird der Pfad $(s, min\, Y.[a]((A \wedge X) \vee Y)) \rightarrow (s, [a]((A \wedge X) \vee Y)) \xrightarrow{a} (t, A \wedge X) \vee Y)$ durch $(s, min\, Y.[a]((A \wedge X) \vee Y)) \xrightarrow{a} (t, A \wedge X) \vee Y)$ ersetzt. Es ist zu beachten, dass die Knoten mit der Formel eines modalen Operator mehrere Nachfolgerknoten haben können. Jeder Nachfolgerknoten wird mit dem Vorgängerknoten direkt verbunden.

Es ist zu beachten, dass die Kürzungsregeln nicht dafür angewandt werden, um den Spielgraphen zu verkleinern. Die Größe von Spielgraphen wird standardmäßig klein gehalten. Wenn der Anwender eine graphische Darstellung eines Spielgraphen sieht, helfen die Kürzungsregeln, die Beschriftung der Knoten zu verstehen.

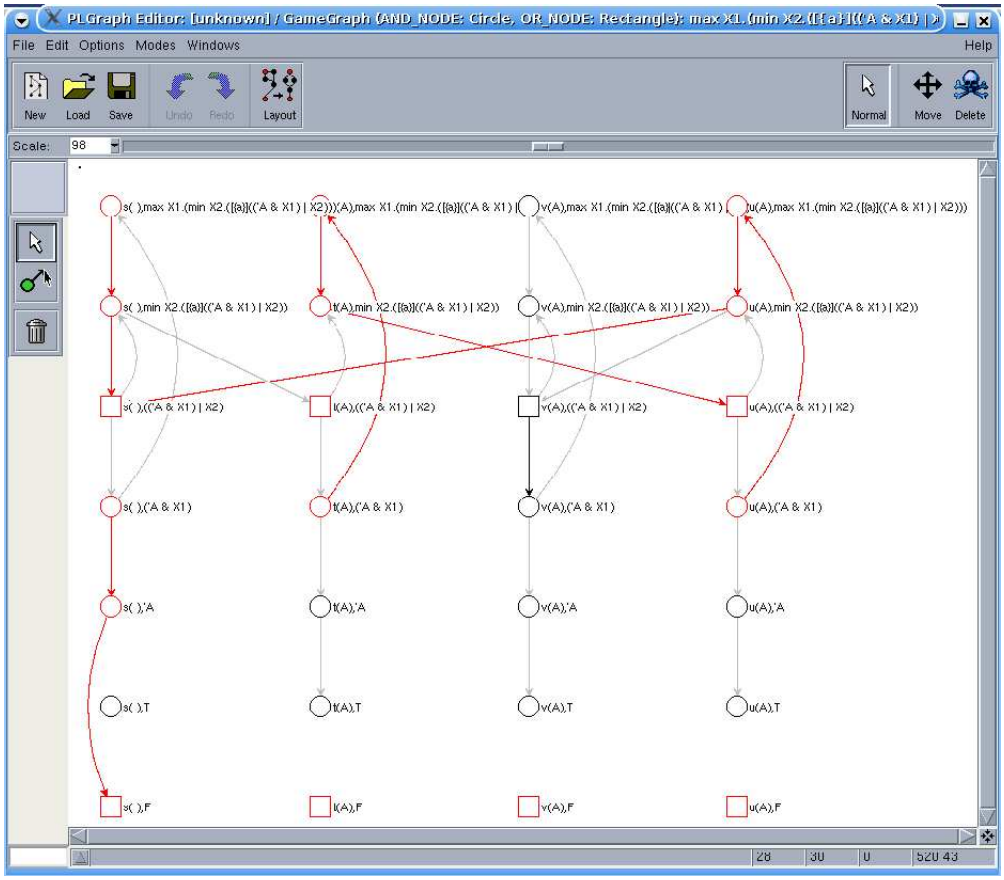

Abbildung 6.7: Detailliert dargestellter Spielgraph

In Abbildung 6.7 wird ein Spielgraph unter Benutzung unseres Tools dargestellt, der identisch mit dem in Abbildung 6.6 vorgestellten Spielgraphen ist, aber zu dessen Aufbau die obigen zwei Kürzungsregeln angewandt werden. Die Knoten der Gewinnmenge W_0 bzw. W_1 sowie die Gewinnstrategie für den Spieler 0 bzw. 1 werden rot bzw. schwarz gefärbt.

Die zwei nächsten Kürzungsregeln werden nur dann angewandt, wenn man einen „kompakten“ Spielgraphen anzeigen lässt. Man wählt das Menü „Set game options“ in dem Hauptmenü und dann markiert die Zeile „Compact“. Die nächste Kürzungsregel bezieht sich auf dem Knoten in Form (s_M, A), wobei s_M ein Knoten des Modells und A eine atomare Proposition ist. Der Knoten hat einen Nachfolgerknoten (s_M, T) bzw. (s_M, F), je nachdem, ob die Proposition A in s_M erfüllt ist. Hat der Knoten (s_M, A) einen Vorgängerknoten, wie z.B. $(s_M, A \wedge \phi)$, und gilt die Proposition A in s_M, dann wird der Pfad $(s_M, A \wedge \phi) \rightarrow (s_M, A) \rightarrow (s_M, T)$ durch $(s_M, A \wedge \phi) \rightarrow (s_M, T)$ ersetzt.

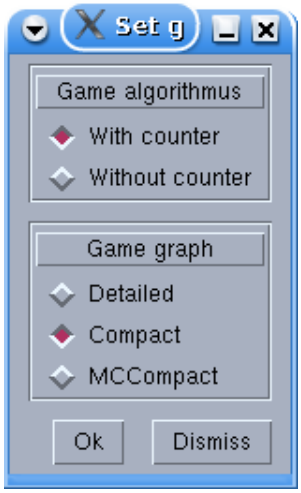

Abbildung 6.8: Optionen

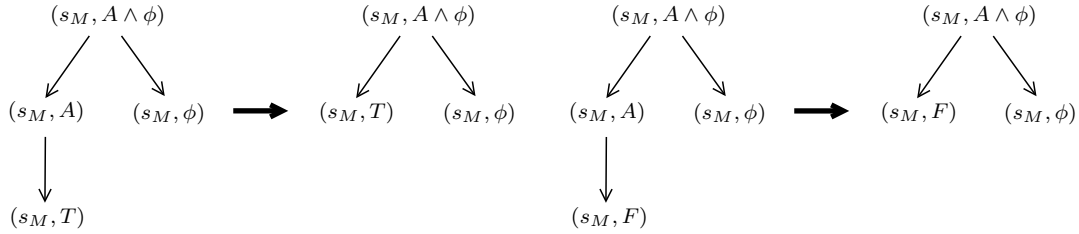

Abbildung 6.9: Elimination der Knoten mit einer atomaren Proposition

In Abbildung 6.9 wird graphisch dargestellt, wie die Knoten mit einer atomaren Proposition eliminiert. Für den Fall, dass der boolesche Operator eine Disjunktion $\vee$ ist, verläuft die Reduzierung des Knotens analog.

Die nächste Kürzungsregel bezieht sich auf die geschachtelten booleschen Operatoren. Werden mehrere Disjunktionen bzw. Konjunktionen direkt nacheinander abgewickelt, dann kann man dies als eine Disjunktion bzw. Konjunktion mit mehrere Operanden darstellen.

In Abbildung 6.10 wird der Fall dargestellt, dass eine Konjunktion mehrere Operanden erhält. Für die Disjunktion verläuft die Kürzung der Abwicklungsschritte analog. In Abbildung 6.11 wird ein Spielgraph dargestellt, der identisch mit dem Spielgraphen in Abbildung 6.7 ist, aber zu dessen Aufbau die obigen zwei Kürzungsregeln angewandt werden.

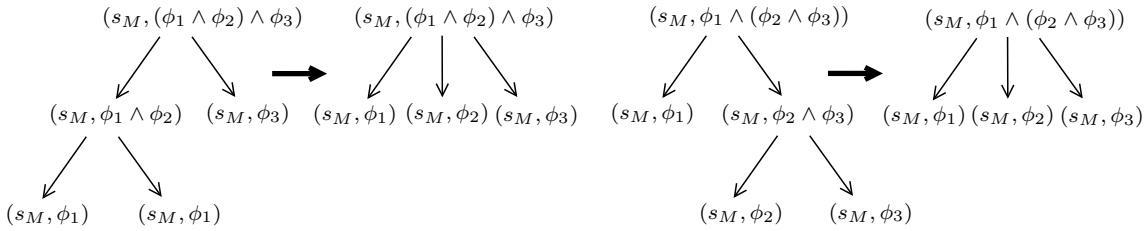

Abbildung 6.10: Kürzung der Abwicklungsschritte bzgl. Konjunktionen

Normalerweise ist es leicht zu erkennnen, welche Knoten sowie Kanten durch welche Kürzungsregel reduziert sind. Es kann jedoch passieren, dass verschiedene Kürzungsregeln parallel an einer Stelle angewandt werden, wodurch der Spielgraph zu kompakt dargestellt wird. In solchem Fall kann der Anwender die Option der Darstellungsart von Spielgraphen auf „detailliert“ setzen.

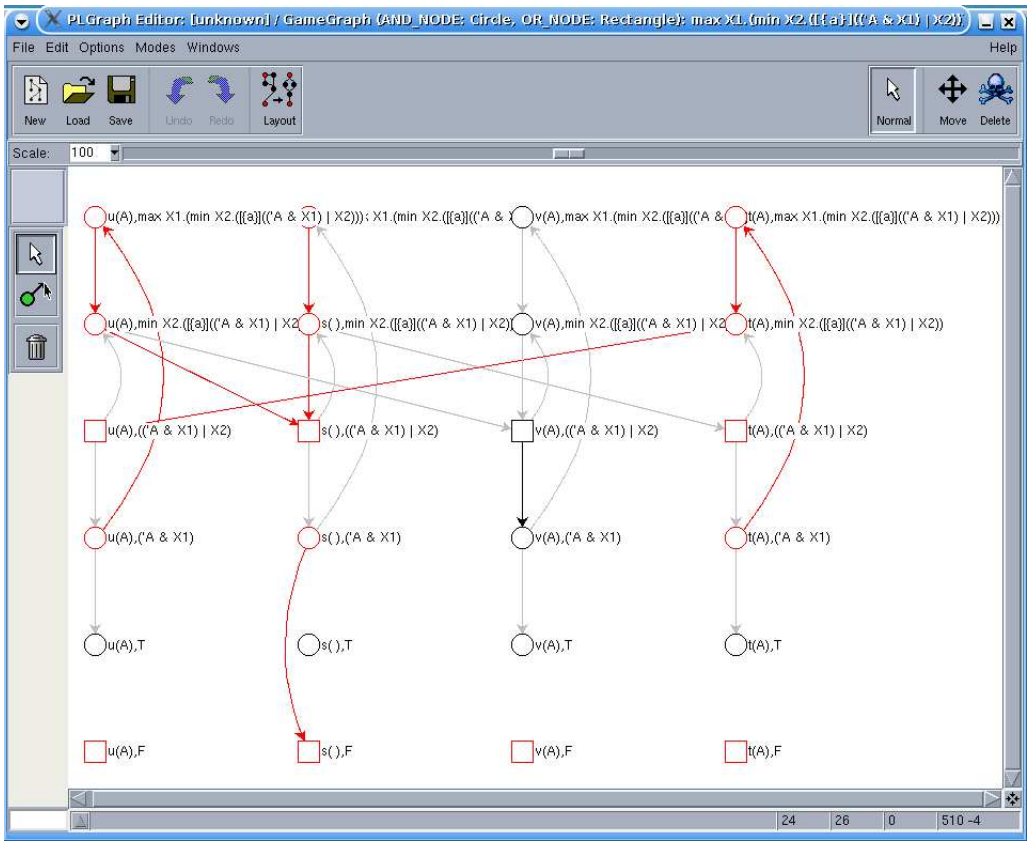

Abbildung 6.11: Kompakt dargestellter Spielgraph

Die *true*- bzw. *false*-Knoten sind Senken und die Bewertung der Knoten kann an ihren Vorgängerknoten verschoben werden. Wird die Knotenreduzierung dieser Art durchgeführt, dann wird die Größe des Spielgraphen zwar kleiner, aber dann

müssen die Gewinnbedingungen angepasst geändert werden, da ein Spiel an einem Knoten beendet werden kann, der einen Nachfolgerknoten hat. Die *true*- und *false*-Knoten werden dann implizit interpretiert und dies kann den Anwender irritieren. In unserem Tool wird diese Möglichkeit in einer Option offen gelassen, so dass der Anwender sie nach Bedarf einschalten kann. Man wählt die Zeile „MCCompact“ in dem Menü „Set game options“.

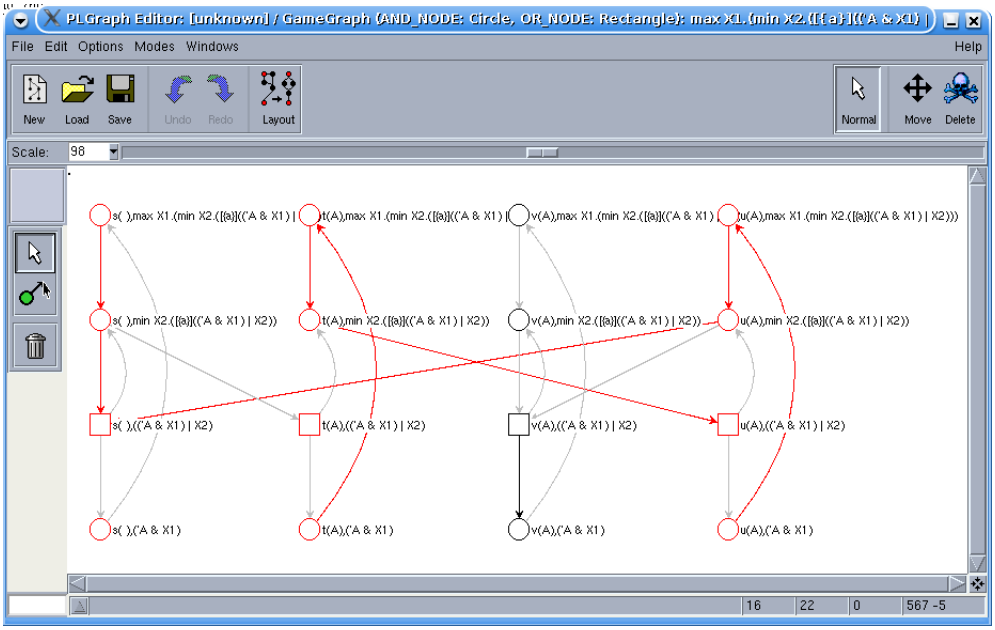

Abbildung 6.12: Kompakt dargestellter Spielgraph ohne *true*- bzw. *false*-Knoten

Unser Tool wird entwickelt, um das Ergebnis von Model-Checking effektiv zu diagnostizieren. Mit dem Tool kann man ein Modell erstellen, eine modale μ-Kalkül-Formel einlesen, das Model-Checking durchführen, und das Ergebnis durch animierte Strategie-Synthese analysieren lassen.

Ist das Modell klein, dann kann man direkt den Spielgraphen, in dem die berechneten Gewinnmengen sowie die Gewinnstrategie dargestellt werden, direkt betrachten, um das Spielergebnis nachzuvollziehen. Man verliert jedoch schnell den Übersicht, sobald das Modell und dementsprechend der Spielgraph groß wird. Um das Spielergebnis gut zu verstehen, braucht man verschiedene mögliche Spielläufe zu verfolgen, was nicht immer einfach ist, wenn man lediglich den gesamten Spielgraphen betrachten kann.

Normalerweise hat man besondere Interesse an bestimmten Spielläufen und deshalb ist es überflüssig, dass der ganze Spielgraph angezeigt wird. Mit unserem Tool kann

der Anwender die Spielläufe, die für ihn von Interesse sind, selektiert konstruieren lassen. Gib der Anwender ein Modell sowie eine modale μ-Kalkül-Formel ein, baut das Tool den Spielgraphen auf und berechnet die Gewinnmengen und eine Gewinnstrategie.

Je nachdem, ob man die Pfade, die die gewünschte Eigenschaft erfüllen bzw. nicht erfüllen, untersuchen möchte, wählt man dann das Menü „Play a Sat. game“ bzw. „Play an Error game“ in dem Hauptfenster wie in Abbildung 6.1. Für den zweiten Fall werden die Fehlerpfade untersucht und das Programm übernimmt die Rolle des Spielers 0. Das Programm wählt den nächsten Knoten stets nach der berechneten Gewinnstrategie, wenn die Knoten in einem Spiellauf auftreten, die zur Knotenmenge $V_0 \cap W_0$ gehören. V_0 ist die Menge der Knoten, von denen aus der Spieler 0 einen Nachfolgerknoten wählt. Dazu gehören die Knoten, deren Formel [.] oder $\wedge$ als höchster Operator hat. W_0 ist die Gewinnmenge des Spielers 0. Der Anwender übernimmt die Rolle des Gegenspielers. Die Spielläufe, die von dem Anwender und dem Programm auf diese Weise interaktiv konstruiert werden, werden in einer baumartigen Struktur dargestellt. In dem nächsten Abschnitt wird anhand eines Beispiels erläutert, wie ein Spiel mit unserem Tool analysiert wird, und welche Optionen man dabei einstellen kann.

6.4 Spielsteurung

Zum Starten eines Spiels wählt der Anwender das Menü entweder „Play a Sat. game“ oder „Play an Error game“. Nach der Berechnung von Gewinnmengen sowie Gewinnstrategie zeigt unser Tool dem Anwender die Knoten, von denen aus das Model-Checking Ergebnis durch animierte Strategie-Synthese analysiert werden kann. In Abbilung 6.13 wird eine Anfangssituation im Tool gezeigt, wobei das Modell von der Abbildung 6.4 und die Formel $max\, X1.min\, X2.[a]((A \wedge X1) \vee X2)$ ist. Der Anwender will die Fehlerpfade untersuchen. Im Hauptfenster werden die Knoten angezeigt, in denen die Formel nicht gilt. Das bedeutet, dass der Spieler 1 von dem Knoten aus das Spiel verliert. Der Anwender kann einen Knoten davon als Anfangsknoten wählen, um zu testen, welche Spielläufe konstruiert werden können.

Da das Programm den nächsten Knoten stets nach der berechneten Gewinnstrategie wählt, werden nur die Knoten aus der Gewinnmenge W_0 auf den Spielläufen auftreten. Das bedeutet, dass der Anwender nur an den Knoten aus $V_1 \cap W_0$ einen

nächsten Knoten wählen kann. Dies geschieht durch ein einfaches Klicken mit dem Maus auf einem Knoten.

In dem rechten kleinen Fenster „Game Menu“ hat man Möglichkeiten, das Spiel zu steuern. Drückt man den Knopf „Play“, dann wird das Spiel fortgesetzt, bis der Anwender einen nächsten Knoten wählen soll. Wenn man den Knopf „Fast Forward“ drückt, wird einen nächsten Knoten, den der Anwender wählen soll, einmal willkürlich gewählt.

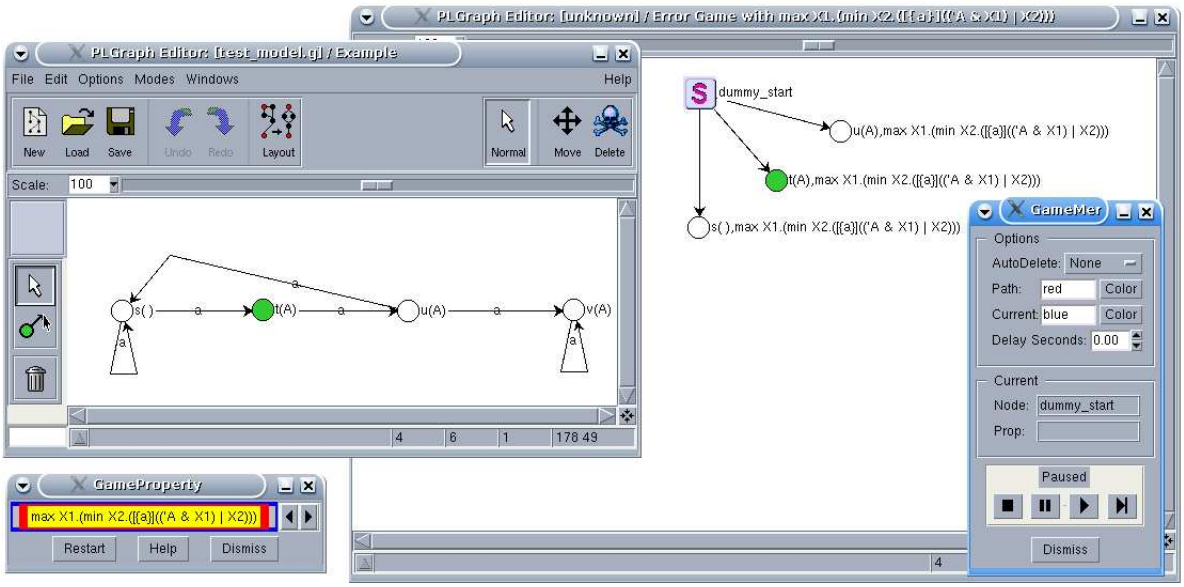

Abbildung 6.13: Starten eines Spiels

Ein Spiel endet nur dann, wenn entweder eine Senke erreicht wird oder eine Schleife als Spiellauf gebildet wird. Falls man einen beliebigen Fehlerpfad vollständig automatisch konstruieren lassen möchte, ohne dass der Anwender interaktiv einen nächsten Knoten wählt, braucht man nur einen Anfangsknoten zu wählen und ihn mit dem Maus doppelt zu drücken.

Standardmäßig wird die Option „AutoDelete“ auf „None“ gesetzt, so dass man mehrere Spielläufe gleichzeitig betrachten kann. Mit „Lastone“ wird der zuletzt aufgebaute Spiellauf automatisch gelöscht, falls er existiert. Wenn man die Option auf „Lastall“stellt, werden alle in dem Fenster dargestellte Spielläufe außer dem aktuellen Spiellauf gelöscht. Man kann auch einen Teilbaum, den man nicht mehr zu betrachten braucht, mit dem Knopf „Cut“ löschen lassen.

Mit der Option „Delay Second“ kann man die Verzögerung der Zeit zum Auf- bzw.

Abbau der Spielläufe bestimmen. Die aktuelle Position auf der baumartigen Darstellung kann auf das Modell und die Formel projiziert werden. Dies wird auf den Felder „Current“ in „Game Menu“ textuell angezeigt. Gleichzeitig wird der aktuelle Knoten im Modell sowie die aktuelle Teilformel im Fenster „Game Property“ graphisch markiert dargestellt.

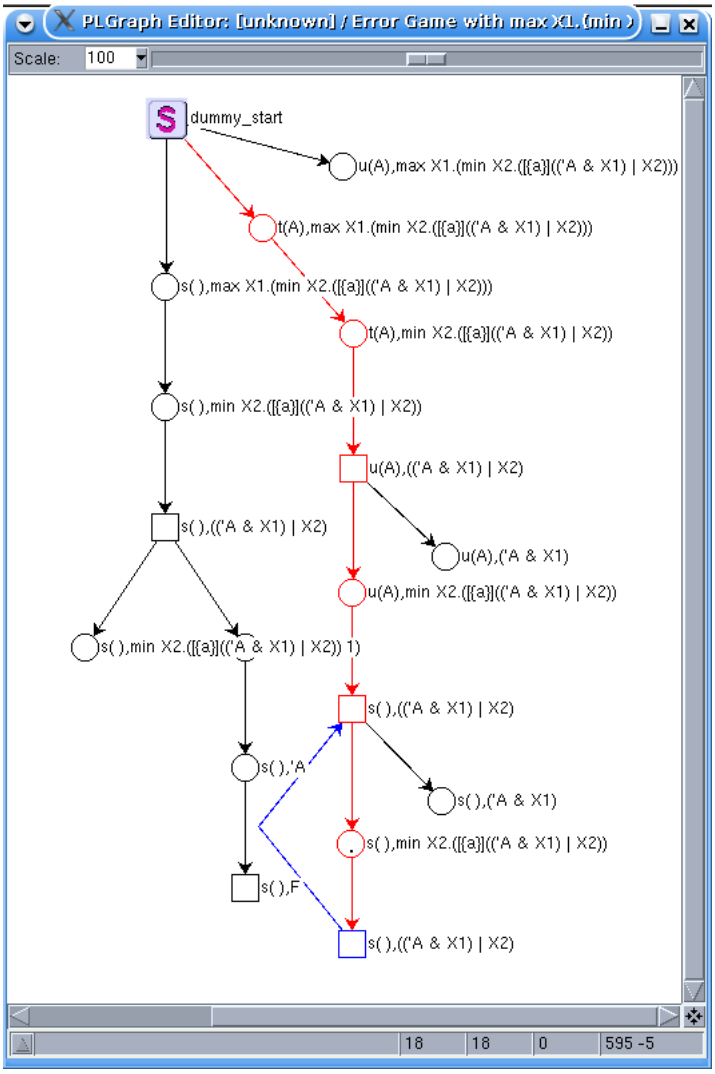

Abbildung 6.14: Baumartig dargestellte Spielläufe

Das Ergebnis des Spiels sowie Model-Checkings kann man auch textuell ausgeben lassen. Solche zusätzliche Möglichkeiten werden im Menü „Auxiliary“aufgelistet. Für den Fall, dass man das Tool zunächst nur zum Model-Checking benutzen und für die bestimmten Problemfälle zur Diagnose des Ergebnisses einsetzen will, kann diese Möglichkeit nützlich sein.

In diesem Kapitel wurde erläutert, wie unser Spiel-Algorithmus als Tool implementiert ist. Es wurde erklärt, wie der Anwender eine μ-Kalkül Formel bzw. ein Modell in das Tool eingeben kann, und wie ein Spielgraph daraus aufgebaut wird. Es wurde dann verschiedene Möglichkeiten der Steuerung vorgestellt, so dass der Anwender verschiedene Spiellauf in einer GUI-basiert Umgebung schrittweise animieren lassen kann.

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein neuer Spiel-Algorithmus zur Fehlerdiagnose beim Model-Checking für die μ-Kalkül entwickelt.

Es wurden die Optimierungsvorschläge von Cleaveland, Steffen [CKS92] und Klein [Kl94] übernommen, die zur Analyse der μ-Kalkül-Formel und des Modells dienen. Dann wurden globale Model-Checking-Probleme in Zwei-Personen-Spiele umgewandelt, wie Stirling [Sti95] und Stevens [StSt98] für lokales Model-Check-Problem vorgestellt haben.

Unter dieser Vorbedingung haben wir einen neuen Spiel-Algorithmus entwickelt, der Model-Checking-Probleme nicht nur löst sondern auch Gewinnstrategie berechnet, die zur Fehlerdiagnose angewandt werden kann.

An erster Stelle lag unsere Interesse, ein GUI-basiertes benutzerfreundliches Tool zu implementieren, mit dem man verschiedene Spielläufe selbst ausprobieren und dadurch mehr Informationen über Model-Checking Ergebnis erhalten kann. Wegen der Erschwernis zum Verstehen der μ-Kalkül bei Anwendern war dies besonders wichtig. Dies wurde mit unserem Tool MetaGame [MuYo] in der METAFrame-Umgebung [BMSY98] erfolgreich umgesetzt.

Bei der Abschätzung der Laufzeitkomplexität unseres Spiel-Algorithmus wurde gezeigt, dass die Einteilung der Knoten nach Prioritäten ein wichtiger Faktor ist. Wir haben den Fall ausführlich behandelt, für den eine Laufzeit relativ genau abgeschätzt werden kann. Darüber hinaus wurde erwiesen, dass man eine sehr gute Laufzeitkomplexität erhält, wenn die Knoten des Spielgraphen in zwei Levels eingeteilt sind.

Ein wesentlicher Vorteil unseres Spiel-Algorithmus besteht darin, dass die untersuchenden Spielgraphen immer kleiner werden, da mindestens ein Knoten nach jedem Iterationslauf aussortiert wird.

Eine Weiterentwicklung der hier erzielten Ergebnisse könnte darin bestehen, dass man die Einteilung der Knoten in Levels parametrisiert und die Laufzeitkomplexität besser abschätzt. Es sei auch interessant, wenn man das Tool MetaGame für Zwei-Personen-Spiele ohne Zusammenhang mit Model-Checking einsetzt und den Laufzeitaufwand für die komplexen Spielgraphen empirisch untersucht.

Literaturverzeichnis

[AKM95] S. Ambler, M. Kwiatkowska, N. Measor. *Duality and the completeness of the modal μ-calculus.* Theoretical Comp. Science, 151: 3-27, 1995.

[And92] H. Anderson. *Model Checking and Boolean Graphs.* In Proceedings of ESOP 92, LNCS 582, Springer Verlag, 1992.

[ArNi90] A. Arnold, D. Niwinski. *Fixed point characterization of Büchi automata on infinite trees.* Inf. Process. Cybern., EIK, 26:451-459, 1990.

[Arn99] A. Arnold. *The μ-calculus alternaation hierarchy is strict on binary trees.* Theoretical Informations and Applications 33, pp. 329-340, 1999.

[BhCl96] G. Bhat, R. Cleaveland. *Efficient Local Model Checking for Fragments of the Modal μ-Calculus.* Proc. Tools and Algorithms for the Construction and Analysis of Systems, TACAS, LNCS 1055, pp. 107-126, 1996.

[BhCle96] G. Bhat, R. Cleaveland. *Efficient Model Checking via the Equational μ-Calculus.* In Eleventh Annual Symposium on Logic in Computer Science, LICS, IEEE Computer Society Press, pp. 304-312, 1996.

[BMSY98] V. Braun, T. Margaria, B. Steffen, H. Yoo *Automatic Error Location for IN Service Definition.* In Lecture Notes In Comp. Science, vol. 1385, pp. 222-237, 1998.

[Brad96] J. C. Bradfield. *The modal mu-calculus alternation hierarchy is strict.* In U. Montanari and V. Sassone, editors, Proc. CONCUR 1996, pp. 233-246, Theoretical Comp. Science 195, pp. 133-153, 1998.

[BSV03] H. Björklund, S. Sandberg, and S. Vorobyov. *A discrete subexponential algorithm for parity games.* In STACS 2003, vol. 2607 of LNCS, pp. 663-674, Springer-Verlag, 2003.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. *Automatic Verification of Finite State Concurrent Programs using Temporal Logic Specifications.* ACM

Transactions on Programming Languages and Systems, vol. 8, no. 2, pp. 244-263, 1986.

[CGH97] E. M. Clarke, O. Grumberg, H. Hamaguchi. *Another Look at LTL Model Checking.* Formal Methods In System Design, vol. 10, no. 1, pp. 47-71, Feb. 1997.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. *Faster Model Checking for the Modal μ-Calculus.* Theoretical Comp. Science, 663, 410-422, 1992.

[ClEm81] E. M. Clarke and E. A. Emerson. *The Design and Synthesis of Synchronization Skeletons Using Temporal Logic.* In Proceedings of the Workshop on Logics of Programs, IBM Watson Research Center, Yorktown Heights, New York, Springer-Verlag LNCS 131, pp. 52-71, 1981.

[Cl90] R. Cleaveland. *Tableau-based Model Checking in the Propositional μ-Calculas.* Acta Inf. 27, 725-747, 1990.

[Dam94] M. Dam. *CTL* and ECTL* as fragments of the modal mu-calculus.* Theoretical Comp. Science, 126, 1994.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. *On model-checking for fragments of μ-calculus.* In International Conference on Computer-Aided Verificaiton, CAV 93, Bd. 697 LNCS, pp. 385-396, Springer, 1993.

[Em81] E. A. Emerson. *Branching Time Temporal Logic and the Design of Correct Concurrent Programs.* Ph.D. Dissertaton, Division of Applied Sciences, Harvard University, 1981.

[Em90] E. A. Emerson. *Temporal and modal logic.* In J. van Leeuwen, Hrsg., Handbook of Theoretical Comp. Science, Bd. B, Kap. 16, pp. 996-1072. Elsevier Science Publishers B.V., Amsterdam, 1990.

[Em96] E. A. Emerson. *Automated temporal reasoning about reactive szstems.* In F. Moller, G. Birtwistle, Hrsg., Logics for Concurrency, Bd. 1043 von LNCS, pp. 41-101, springer, Berlin, 1996.

[EmHa86] E. A. Emerson and J. Y. Halpern. *"Sommetimes" and "not never" revisited: on branching time versus linear time temporal logic.* Journal of the Association for Computing Machinery, 33(1): 151-178, 1986.

[EmJu88] E. A. Emerson and C. S. Jutla. *The Complexity of Tree Automata and Logics of Programs.* In 29th Annual Symposium on Foundations of Computer Science, pp. 328-337. IEEE, 24-26 Oktober 1988.

[EmJu91] E. A. Emerson and C. S. Jutla. *Tree Automata μ-Calculus and Determinacy.* In Proc. 32th IEEE Symposium on Foundations of Computer Science(FOCS). pp. 368-277. 1991.

[EmLe86] E. A. Emerson and C.-L. Lei. *Efficient Model Checking in Fragments of the Propositional Mu-Calculus.* Proceedings of the IEEE-CS Conference on Logic in Computer Science, Cambridge, Massachusetts, pp. 267-278, 1986.

[FiLa79] M. J. Fischer and R. E. Ladner. *Propositional dynamic logic of regular programs.* J. Computer and System Science 18, pp. 194-211, 1979.

[GTW02] E. Grädel, W. Thoma, and T. Wilke, editors. *Automata, Logics, and Infinite Games.* A Guide to Current Research, vol. 2500 LNCS, Springer, 2002.

[HeMi85] M. C. B. Hennessy and R. Milner. *Algebraic laws for nondeterminism and concurrency.* Journal of the ACM 32, pp. 137-161, 1985.

[JPZ06] M. Jurdziński, M. Paterson, U. Zwick. *A deterministic subexponential algorithm for solving parity games.* In SODA 2006, pp. 117-123, 2006.

[Ju00] M. Jurdziński. *Small progress measures for solving parity games.* In STACS 2000, vol. 1770 of LNCS, pp. 290-301, Springer-Verlag, 2000.

[Ju98] M. Jurdziński. *Deciding ther winner in parity games is in UP$\cap$ co-UP.* Information Processing Letters, 68(3), pp. 119-124, 1998.

[Ki96] A. Kick. *Generation of Counterexamples and Witnesses for Model Checking.* PhD thesis, Fakultät für Informatik, Univ. Karlsruhe, July 1996.

[Kl52] S. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, 1952.

[Kl94] M. Klein. *The Fixpoint Analysis Machine.* Shaker Verlag, 1994.

[Ko83] D. Kozen. *Results on the propositional mu-calculus.* Theoretical Comp. Science 27, pp. 333-354, 1983.

[Kr53] S. Kripke. *Semantical considerations on modal logics.* Acta Philosophica Fennica 16, pp. 83-94, 1953.

[Kr59] S. Kripke. *A completeness theorem in modal logic.* J. Symbolic Logic 24, pp. 1-14, 1959.

[LBCJM94] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. *An Improved Algorithms for the Evaluation of Fixpoint Expressions.* Im Procedddings of CAV '94, LNCS 818, pp. 338-350, Springer Verlag, 1994.

[Len96] G. Lenzi. *A hierarchy theorem for the mu-calculus.* Proc. ICALP '96, LNCS 1099, pp. 87-109, 1996.

[Mad95] A. Mader. *Modal μ-calculus, model checking and Gauß elimination.* In E. Brinksma, W. R. Cleaveland, K. G. Larsen, T. Maragria, B. Steffen, Hrsg., Tools and Algorithms for the Construction and Analysis of Systems, Bd. 1019 von LNCS. pp. 72-88, Berlin, 1995.

[MaPn95] Z. Manna and A. Pnueli. *Temporal Verfications of Reactive Systems.* Springer, Heidelberg, 1995.

[McN93] R. McNaughton. *Infinite games played on finite graphs.* Ann. Pure Apple Logic 65, pp. 149-184, 1993.

[Mo91] A. W. Mostowski. *Games with forbidden positions.* Technical Report 78, University of Gdansk, 1991.

[MuYo] M. Müller-Olm, H. Yoo. *MetaGame: An Animation Tool for Model-Checking Games.* TACAS 2004, pp. 163-167, 2004

[Ni86] D. Niwiński. *On fixed-point clones.* In L. Kott, Hrsg., Automata, Languages and Programming, Bd. 226 von LNCS, pp. 464-473, Berlin, 1986. Springer.

[Ni88] D. Niwiński. *Fixed points vs. infinite generation.* In Third Annual Szmposium on Logic in Computer Science, pp. 402-409, IEEE computer Society, Edinburgh, Scotland, 5-8 Juli 1988.

[Ob03] J. Obdrzálek. *Fast mu-calculus model checking when tree-width is bounded.* In International Conference on Computer-Aided Verification, CAV 2003, vol. 2725 LNCS, pp. 80-92, Springer, 2003.

[Pn77] A. Pnueli. *The temporal logic of programs.* In Proc. 18th IEEE Symposium on Foundations of Computer Science, pp. 46-57, 1977.

[Pr76] V. Pratt. *Semantical condierations of Floyd-Hoare logic.* Proc. 16th IEEE FOCS, pp 109-121, 1976.

[QuSi82] J. P. Queille and J. Sifakis. *Specification and Verification of Concurrent Programs in CESAR.* Proc. 5th Int. Symp. Prog., Springer LNCS no. 137, pp. 195-220, 1982.

[Sti92] C. Stirling. *Modal and temporal logics.* In Handbook of Logic in Computer Science, S. Abramskz, Dov M. Gabbay, and T. S. E. Maibaum, eds., Clarendon Press, 477-563, 1992.

[Sti95] C. Stirling. *Local model checking games.* Lecture Notes in Computer Science, 962, 1-11, 1995.

[Sti96] C. Stirling. *Modal and temporal logics for processes.* Lecture Notes in Computer Science, 1043, 149-237, 1996.

[StSt98] P. Stevens and C. Stirling. *Practical model-checking using games.* Lecture Notes in Computer Science, 1384, 83-101, 1998.

[StWa89] C. Stirling and D. Walker *Local Model Checking in the Model Mu-Calculus.* In Proceedings of TAPSOFT 89, LNCS 53l, Springer Verlag, 369-383, 1989.

[Tar55] A. Tarski. *A lattice-theoretical fixpoint theorem and its application.* Pacific Journal of Mathematics 5, pp. 285-309, 1955.

[Tho02] W. Thomas. *Infinite games and verification.* In Proceedings of the International Conference on Computer Aided Verification CAV 02, volume 2404 of Lecture Notes in Computer Science, pages 58-64. Springer, 2002.

[Tho90] W. Thomas. *Automata on infinite objects.* In J. van Leeuwen, editor, Handbook of Theoretical Comp. Science, volume B, chapter, 4, pages 131-191. North-Holland, Amsterdam, 1990.

[Tho95] W. Thomas. *On the synthesis of strategies in infinite game.* In Ernst W. Mayr and Claude Puech, editors, STACS 95, volume 900 of LNCS, pages 1-13, Berlin, 1995, Springer-Verlag.

[Tho97] W. Thomas. *Languages, Automat, and Logic.* In Handbok of formal languages, vol. 3, pages 389-445, New York, 1997, Springer-Verlag.

[VaWo94] M. Y. Vardi and P. Wolper. *Reasoning about infinite computations.* Information and Computation 115, pp. 1-37, 1994.

[VoJu00] J. Vöge and M. Jurdziński. *A discrete strategy improvement algorithm for solving parity games.* In CAV 2000, vol. 1855 of LNCS, pp. 202-215. Springer-Verlag, 2000.

[Wa93] I. Walukiewicz. *On completeness of the μ-Kcalculcs.* In IEEE Symposium on Logic in Computer Science, pp. 136-146. IEEE computer Society Press, 1993.

[Wa95] I. Walukiewicz. *Completeness of Kozen's axiomatisation of the propositional μ-calculus.* In Tenth Annal IEEE Symposium on Logic in Computer Science, pp. 14-24, San Diego, California, 26-29 Juni IEEE computer Society Press, 1995.

[Wo83] P. Wolper. *Temporal logic can be more expressive.* Information and Control, 56:185-194, 1983.

Stichwortverzeichnis

$K \models f$, 14
μ-Kalkül
  Semantik, 18
  Syntax, 17
σ-Teilformel, 21

Abhängigkeitsgraphen, 47, 48
Alternierungstiefe, 21

Beweiseregeln, 54

CTL, 14
CTL^*
  Semantik, 10
  Syntax, 9

Fixpunkt, 28
Forcemenge, 64
Forcemenge
  maximal, 64
Forcestrategie, 64
Formel
  geschlossen, 19
  Länge, 12
  PNF, 19
  positive Normalform, 13
  wohlbenannt, 18
  wohlgeformt, 19

Gewinnbedingungen, 59
Gewinnmenge, 62
Gewinnstrategie, 61
Gleichungssystem, 42

Halbordnung, 26

$Inf(\pi)$, 59
Infimum, 26

Kette, 27
Kettenvollständigkeit, 27
Kripke-Struktur, 9
Kripke-Struktur
  Pfad, 10

Level, 93
LTL, 15

$maxInf(\pi)$, 59
$minInf(\pi)$, 59
Modell, 16
Monotonie, 27

Partielle Ordnung, 26

reduzierte Alternierungstiefe, 22

Schachtelungstiefe, 20
Spiel, 58
Spielgraph, 57
Spiellauf, 58
Spielregel, 58
Stetigkeit, 27
Strategie, 61
Strategie

speicherfrei, 61
Supremum, 26

Tableau, 54

Lebenslauf

Name: Haiseung Yoo
Geburtsdatum: 22.02.1958
Geburtsort: Seoul in Korea

03/1964 - 03/1970 Sun-Shin Grundschule in Seoul
03/1970 - 03/1973 Kwang-Shin Mittelschule in Seoul
03/1973 - 03/1976 Dong-Sung Oberschule in Seoul
03/1976 - 03/1978 Studium der Germanistik an der Korea-Universität in Seoul
10/1978 - 10/1981 Militärdienst
03/1982 - 03/1984 Studium der Germanistik an der Korea-Universität in Seoul
03/1984 Erlangung des Bachelors an der Korea-Universität
10/1986 - 03/1994 Studium der Informatik an der Christian Albrechts-Universität zu Kiel
11/1994 Diplom in Informatik (Nebenfach: Mathematik)
06/1996 - 06/1997 Wissenschaftlicher Mitarbeiter am Lehrstuhl für Programmiersysteme der Universität Passau
07/1997 - 07/2002 Wissenschaftlicher Mitarbeiter am Lehrstuhl 5 für Programmiersysteme der Universität Dortmund
08/2002 - 02/2007 Promotion