Symbolische Synthese zustandsbasierter reaktiver Programme

von
Nico Wallmeier
Matr.-Nr. 214248

Diplomarbeit

in Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen
im Januar 2003

Angefertigt am
Lehrstuhl für Informatik VII
(Logik und Theorie diskreter Systeme)
bei
Professor Dr. Wolfgang Thomas

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

Aachen, den 3. Januar 2003

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Hintergrund und Aufgabenstellung . . . 1

1.2 Aufbau der Arbeit . . . 4

2 Grundlagen **5**

2.1 Reaktive Programme über einem abstrakten Zustandsraum . . . 5

2.1.1 Garantie- und Sicherheitsspiele . . . 7

2.1.1.1 Sicherheitsspiel . . . 8

2.1.1.2 Linearzeitalgorithmus für den Attraktor . . . 9

2.1.2 Schwache Paritätsspiele . . . 9

2.1.3 Staiger-Wagner-Spiele . . . 12

2.1.4 Request-Response-Spiele . . . 13

2.1.5 Büchi-Spiele . . . 15

2.1.6 Paritätsspiele . . . 18

2.1.7 Muller-Spiele . . . 20

2.1.8 Rabin- und Streett-Spiele . . . 21

2.1.9 Übersicht über die vorgestellten Spiele . . . 22

2.2 Binäre Entscheidungsbäume (Binary Decision Diagrams) . . . 23

3 Reaktive Programme über einen symbolischen Zustandsraum **27**

3.1 Terminologie . . . 27

3.2 Spielgraph . . . 27

3.3 Garantie- und Sicherheitsspiele . . . 29

3.4 Schwache Paritätsspiele . . . 32

3.5 Staiger-Wagner-Spiele . . . 34

3.6 Request-Response-Spiele . . . 36

3.7 Büchi-Spiele . . . 39

3.8 Paritätsspiele . . . 42

3.9 Muller-, Rabin- und Streett-Spiele . . . 46

3.10 Erreichte Ergebnisse . . . 47

4 Einsatz der Implementierung und Fallstudien **48**
4.1 Die Eingabesprache . . . 48
4.1.1 Grammatik der Sprache . . . 49
4.1.2 Beispiele . . . 50
4.2 Die einzelnen Seiten der GUI . . . 51
4.2.1 Die Eingabeseite . . . 52
4.2.2 Die Parser-Seite . . . 54
4.2.3 Die Zwischenergebnis-Seite . . . 54
4.2.4 Die Ergebnis-Seite . . . 55
4.3 Fallstudie . . . 56
4.3.1 Spieldefinition . . . 57
4.3.2 Ergebnisse . . . 60

5 Die Implementierung **62**
5.1 Prozeßentscheidungen . . . 62
5.2 Architekturentscheidungen . . . 63
5.2.1 Der Parser-Generator SableCC . . . 63
5.2.2 Das BuDDy-Paket . . . 65
5.2.3 SWT - Standard Widget Toolkit . . . 66
5.3 Die Architektur . . . 67
5.3.1 Das Paket bdd: Kapselung der BDDs . . . 68
5.3.2 Das Paket games: Aplication Programming Interface (API) . . . 70
5.3.3 Das Paket gui: Das grafische Benutzerinterface . . . 72
5.3.4 Das Paket parser: String → BDD . . . 75
5.3.5 Weitere Spiele hinzufügen . . . 76
5.4 Qualitätssicherung . . . 76
5.4.1 Testziele . . . 77
5.4.2 Überprüfung des Deckungsgrades . . . 78

6 Zusammenfassung und Ausblick **80**

A Inhalt der CD **82**

B Beispiele auf der CD **83**

C Verwendete Software **85**

Literaturverzeichnis **87**

Kapitel 1

Einleitung

1.1 Hintergrund und Aufgabenstellung

Die Erfahrungen aus dem Absturz der ersten Ariane-5 Rakete (nachzulesen in der Zeitschrift iX, Ausgabe September 1996, Seite 32) oder der Fehler des Intel Pentium 1-Prozessors haben gezeigt, dass Verifikation von Systemen eine wichtige Rolle spielen sollte. Mögliche Ansätze dazu können im Testen und Simulieren bestehen. Allerdings bieten diese Verfahren keine Korrektheitsgarantie und sind unter Umständen nur eingeschränkt anwendbar. Somit rücken die computergestützten Verfahren zur formalen Verifikation in den Vordergrund.

Eine Möglichkeit besteht im sogenannten „Model Checking“ ([CGP99]). In [CS01] wird Model Checking definiert als:

> Model checking is an automatic technique for verifying correctness properties of safety-critical reactive systems.

Der Ansatz beim Model Checking besteht darin, ein System gegen eine Spezifikation zu prüfen. Dazu wird ausgehend vom System ein Modell des Systems erzeugt, welches auch Kripke-Struktur genannt wird. Die Spezifikation wird in einer temporalen Logik angegeben und zusammen mit dem Modell des Systems dem Model-Checker übergeben. Erfüllt das Modell die Formel nicht, wird ein Fehlerszenarium zurückgegeben.

Ein feineres Modell besteht aus zwei Agenten, wobei einer die Kontrolle darstellt und der andere die Umgebung. Somit lassen sich nichtterminierende reaktive Prozesse modellieren, die dadurch gekennzeichnet sind, dass ihre Aktionen im fortlaufendem Wechselspiel mit der Umgebung ausgeführt werden. Die Entwicklung heutiger Systeme geht immer mehr in diese Richtung und wird von Milner als „computing is interaction“ ([Mil89]) bezeichnet.

Aufbauend auf den Arbeiten der sechziger Jahre von Church [Chu62], Büchi [Büc62], Rabin [Rab69] und Weiteren sind die unendlichen Zweipersonen-Spiele auf endlichen Spielgraphen entstanden. In den Arbeiten von Church, Büchi und Rabin wird eine Logik-Beschreibung in ein deterministisches Transitionssystem und einer Akzeptierbedingung (ω-Automat) umgesetzt. McNaughton ([**?**, McN93]) sowie Gurevich und Harrington ([GH82]) haben damit angefangen, Systemtransitionen und Gewinnbedingung zu trennen. Die Systemtransitionen

werden in der Literatur auch als „Arena“ oder Spielgraph bezeichnet. Dabei kann für die Gewinnbedingung entweder direkt eine Akzeptierbedingung angegeben werden oder sie wird mittels einer Logik spezifiziert, aus der wiederum eine Akzeptierbedingung abgeleitet werden kann. Dies entspricht der heutigen Form unendlicher Zweipersonen-Spiele auf endlichen Spielgraphen. Diese werden spezifiziert durch ihren Spielgraphen sowie einer Gewinnbedingung an die unendlichen Läufe auf dem Spielgraphen. In der Praxis müssen Spielgraph und Gewinnbedingung gemeinsam behandelt werden, da sich die Anforderungen an ein Spiel nur mittels beider Komponenten zusammen festlegen lassen.

Die Gewinnbedingungen greifen in der Art auf die Theorie der ω-Automaten zurück, dass Spieler 0 genau die Partien gewinnt, die der zugrundeliegende Automat akzeptiert. Büchi hat in seiner Arbeit ([Büc62]) angefangen, Automaten mit unendlichen Läufen zu untersuchen. Weitere ω-Automaten wurden eingeführt von Muller ([Mul63]), Rabin ([Rab72]), Streett ([Str82]), Staiger und Wagner ([SW74]), sowie von Hütten ([Hüt03]). Eine Sonderrolle nimmt die Paritätsbedingung ([Mos84, **?**]) ein, da sich alle genannten Gewinnbedingungen darauf zurückführen lassen. Dazu ist aber bei den meistens Gewinnbedingungen eine Vergrößerung des Zustandsraumes der betrachteten Automaten notwendig. Zu einem Muller-Automaten kann der konstruierte Paritätsautomat exponentiell größer werden als der gegebene. Im nächsten Kapitel werden verschiedene Spiele mit ihren Gewinnbedingungen noch genauer vorgestellt.

Die für Spiele interessanten Fragen sind die Gewinnbereiche und -strategien. Dabei ist unter einer Gewinnstrategie zu verstehen, dass ein Spieler den Gewinn erzwingen kann. Alle Knoten von denen ein Spieler eine Gewinnstrategie besitzt, umfassen seinen Gewinnbereich. Bei allen hier betrachteten Spielen hat für jeden Zustand des Spielgraphens genau einer der beiden Spieler eine Gewinnstrategie. Dies folgt aus der Tatsache, dass die automatentheoretischen Gewinnbedingungen zu Spielen führen, die in der deskriptiven Mengenlehre zu den ersten beiden Stufen der Borel-Hierarchie gehören. Nach Martin ([**?**]) folgt somit, dass alle betrachteten Spiele determiniert sind. Eine bedeutende Arbeit in dem Bereich der Gewinnstrategien ist die Arbeit von Büchi und Landweber ([BL69]), da sie in Ihrer Arbeit gezeigt haben, dass für endliche Spielgraphen mit Muller-Bedingung gilt:

- Es ist algorithmisch entscheidbar, welcher der beiden Spieler zu einem gegebenen Zustand des Spielgraphens aus eine Gewinnstrategie besitzt.
- Eine Gewinnstrategie ist durch einen „Strategieautomaten“, einem endlichen Automaten mit Ausgabe, aus dem Spielgraphen und der Gewinnbedingung effektiv konstruierbar. Somit wird nur Speicher fester Größe für den bisherigen Verlauf der Partie benötigt, um mit diesen Informationen die Zugauswahl zu treffen.

Emerson, Jutla und Mostowski ([Mos84, **?**]) haben gezeigt, dass bei einer Paritätsgewinnbedingung für beide Spieler positionale Gewinnstrategien ausreichen. Dies bedeutet, dass die Kantenauswahl nur anhand des aktuellen Zustandes getroffen wird und somit kein Speicher für den Verlauf der Partie benötigt wird.

Die wichtigste Kontruktion zur Lösung von reaktiven Systemen ist die Attraktor-Berechnung. Diese berechnet zu einer gegebenen Zustandsmenge die Zustände, von denen Spieler 0 den Besuch der Menge erzwingen kann. Neben naiven Implementierungen ist in [Tho00] eine

Variante vorgestellt worden, die den Attraktor in Linearzeit berechnet. Durch den Attraktor lassen sich die Garantie-, Sicherheits- und schwachen Paritätsspiele lösen. Mittels einer geringen Modifikation ist davon der Attraktor+ ableitbar, welcher zusammen mit der Recur-Berechnung die Grundlage für das Lösungsverfahren der Büchi-Spiele liefert. Die Request-Response Spiele werden nach [Hüt03] auf Büchi-Spiele reduziert und Paritätsspiele werden mittels des McNaughtons-Algorithmus ([McN93]) gelöst.

Eine praktische Umsetzung ist bisher in der Literatur nicht durchgeführt. Für das Model Checking existieren Ansätze auf zwei unterschiedlichen Ebenen, um es in der Praxis einsetzen zu können.

1. Gute gewählte Spezifikationslogik für Polynomzeitverfahren - Computation Tree Logic, kurz CTL, nach Clarke und Emerson. CTL umfasst sowohl Pfadquantoren als auch temporale Operatoren und gehört somit zu den Logiken mit verzweigter Zeit (branching time). Mit ihren Algorithmen zur Modellüberprüfung mit CTL legten Clark und Emerson Anfang der 80er Jahre den Grundstein zur effizienten automatisierten Verifikation von Zustandsübergangssystemen durch Modellüberprüfung. Der Aufwand für den Test, ob eine CTL-Formel in einem Zustand einer Kripke-Struktur erfüllt ist, ist linear sowohl in der Länge der Formel als auch der Größe des Modells.
2. Eine symbolische Methode ([BCM+90, **?**]), um die Zustandsexplosion zu umgehen, welche in der Literatur auch unter „State Explosion Problem“ zu finden ist. Dazu werden statt expliziter Repräsentationen von Zuständen boolesche Formel verwendet. Als Datenstruktur für die booleschen Formeln werden „Binary Decision Diagrams“, kurz BDDs, die auf Lee ([Lee59]), Akers ([Ake78]) und Moret ([Mor82]) zurückgehen und von Bryant ([Bry86, Bry92]) zu „Reduced Ordered Binary Decision Diagrams“ umgesetzt worden sind. Dadurch ist eine erhebliche Platzersparnis erreicht worden.

Die Algorithmen zur Lösung der unendlichen Spiele über einem endlichen Spielgraphen sind über den abstrakten Zustandsräumen konzipiert. In dieser Arbeit wurde der zweite Ansatz, die symbolische Darstellung der Zustände, vom Model Checking aufgegriffen, um erstmals praktische Studien durchführen zu können. Somit verknüpft die Arbeit zwei unterschiedliche Methoden - einerseits die „Binary Decision Diagrams“ zur Darstellung boolescher Formeln und andererseits Algorithmen aus dem Bereich der zustandsbasierten reaktiven Systeme. Daraus ergeben sich zwei Aufgaben, welche zu lösen sind:

1. Umsetzung der Algorithmen auf symbolischer Ebene
2. Entwicklung eines Interfaces für die Durchführung von Fallstudien und Beispielen

Als erstes Resumé lässt sich festhalten, dass die Kodierung praktischer Beispiele jetzt möglich ist. Dies ist durch die Eingabesprache aus dem Kapitel 4.1 erreicht worden. Es zeigten sich jedoch enggezogene Grenzen, da beispielsweise bei der Reduktion der Request-Response Spiele auf Büchi Spiele ein exponentielles Wachstum des Zustandsraumes in Bezug auf die Anzahl der Request-Response Paare auftritt. Als mögliche Ansätze haben sich während dieser Arbeit die folgenden Punkte ergeben, um den praktischen Einsatz der unendlichen Spiele genauso wie beim Model Checking ermöglichen zu können, wie im Kapitel 6 nachgelesen werden kann:

- Finden einer Spezifikationslogik analog CTL im Model Checking

- Finden eines Fragmentes von QBF (qualifizierte boolesche Formeln), um die Quantorenanzahl in der Spezifikation von Spielen zu reduzieren
- Erweiterung bestehender Ansätze von Alur, Kannan und Yannakakis [?] bzw. von Benedikt, Godefroid und Reps [?] auf das spieltheoretische Paradigma

1.2 Aufbau der Arbeit

Im nächsten Abschnitt wird eine Einführung in die zustandsbasierten reaktiven Systeme über einem abstrakten Zustandsraum gegeben. Dazu werden dort die wichtigsten Definitionen und Algorithmen zur Lösung der Spiele vorgestellt. Dabei hat sich herausgestellt, dass bei allen dort vorgestellten Spielreduktionen ein exponentielles Wachstum der Zustandsanzahl stattfindet, entweder in Abhängigkeit der Zustände des gegebenen Spielgraphens oder der Gewinnbedindung. Weiterhin werden die „Binary Decision Diagrams", kurz BDDs, nach Lee ([Lee59]), Akers ([Ake78]) und Moret ([Mor82]) eingeführt. ROBBDs von Bryant ([Bry86, Bry92]) bieten eine kompakte und eindeutige Darstellung boolescher Formeln sowie effiziente Möglichkeiten zur Manipulation.

Im dritten Kapitel wird der symbolische Zustandsraum mit der dazugehörenden Notation eingeführt. Weiterhin werden die Ergebnisse vorgestellt, die in dieser Arbeit bei der Transformation der Algorithmen vom abstrakten auf den symbolischen Zustandsraum erzielt werden konnten. Von einer symbolischen Umsetzung der Muller-, Streett- und Rabin-Spiele ist abgesehen worden, da kein Gewinn zu erwarten gewesen wäre. Alle anderen Spiele sind umgesetzt. Allerdings hat die Lösung der Staiger-Wagner Spiele auch keinen Vorteil im Symbolischen gebracht.

Das vierte Kapitel stellt die entwickelte Eingabesprache zur Spezifikation der Spiele vor. Die Eingabesprache zeichnet sich durch ihre existenziellen und universellen Quantoren aus und ermöglicht somit kompakte Spezifikationen der Spiele. Auf der anderen Seite hat sich herausgestellt, dass der Einsatz zu vieler verschachtelter Quantoren problematisch ist. Weiterhin wird das entstandene Programm, welches die Algorithmen aus dem dritten Kapitel umsetzt, aus Benutzersicht beschrieben. Den Schluss dieses Abschnittes bildet die Fallstudie, die auch die Grenzen dieser Arbeit aufgezeigt hat. Dort wird eine Aufzugssteuerung bestehend aus zwei Aufzügen analysiert.

Zum Abschluss der Arbeit wird im fünften Kapitel noch der interne Aufbau des Programmes vorgestellt. Neben dem internen Aufbau des Programmes sind die wichtigsten Entscheidungen, wie z.B. die Wahl des BDD-Paketes, bei der Entstehung des Programmes dokumentiert worden. In diesem Abschnitt befinden sich auch die nötigen Informationen, wenn das Programm um weitere Spiele erweitert werden soll. Die möglichst leichte Erweiterbarkeit um zusätzliche Spiele war neben der Softwarequalitätssicherung einer der Schwerpunkte bei der Implementierung des Programms.

Kapitel 2

Grundlagen

In diesem Kapitel wird zuerst eine Einführung von reaktiven Programmen über einem abstrakten Zustandsraum gegeben. In einem weiteren Abschnitt werden dann noch die BDDs (binary decision diagrams) vorgestellt. Dabei handelt es sich um eine Datenstruktur für boolesche Formeln. BDDs spielen für diese Diplomarbeit insofern eine Rolle, als dass sie für die Implementierung der im dritten Kapitel erarbeiteten Algorithmen als Datenstruktur für die booleschen Formeln ausgewählt worden sind.

2.1 Reaktive Programme über einem abstrakten Zustandsraum

Ziel dieses Abschnittes ist es, eine Einführung in die reaktiven Systeme zu geben. Dabei wird zwischen einem abstrakten und einem symbolischem Zustandsraum unterschieden. Unter einem abstrakten Zustandsraum ist eine endliche Anzahl von konkreten Zuständen zu verstehen - wie zum Beispiel die Menge $Q = \{q_0, q_1, q_2, \ldots, q_{10}\}$. Grundlage des symbolischen Zustandsraum bilden Aussagevariablen - wie z.B. $v_0, v_1, \ldots, v_5$. Ein Zustand des symbolischen Zustandsraum ist eine konkrete Belegung dieser Variablen. Dies bedeutet, dass sich mit i Variablen 2^i-Zustände beschreiben lassen (jede Variable kann mit $true$ oder $false$ belegt werden). Dieses Kapitel soll helfen, das nächste Kapitel - die reaktiven Programme über einem symbolischen Zustandsraum - besser verstehen und die Äquivalenzen zwischen Symbolischen und Abstrakten einfacher erkennen zu können.

Unter reaktiven Systemen werden unendliche Spiele mit einer Gewinnbedingung über einen endlichen Spielgraphen verstanden. Es gibt verschiedene Spiele, die sich durch ihre jeweiligen Gewinnbedingungen unterscheiden. Eine Gewinnbedindung ist eine Anforderung an eine Partie, damit sie als für Spieler 0 gewonnen gilt. Einige dieser Spiele (und somit auch Beispiele für Gewinnbedingungen), werden in den weiteren Unterkapiteln vorgestellt. Spielgraphen sind dabei gerichtete Graphen, deren Knoten sich disjunkt den Spielern 0 und 1 zuordnen lassen. Abbildung 2.1 liefert ein Beispiel für einen Spielgraphen. Dabei sind die Knoten von Spieler 0 als Kreise dargestellt und die von Spieler 1 als Rechtecke. Damit ein unendliches Spiel immer möglich ist, muss jeder Knoten mindestens einen Nachfolger haben. Denn sonst müsste der Sonderfall betrachtet werden, dass ein Spiel vorzeitig terminiert.

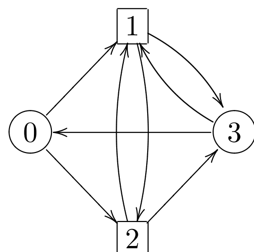


Abbildung 2.1: Beispiel eines Spielgraphen

Def. 2.1 (Spielgraph) *Ein Spielgraph G ist ein Tupel (Q, E) mit Zustandsmenge $Q = Q_0 \dot{\cup} Q_1$, Kantenrelation $E \subseteq Q \times Q$ und für alle $q \in Q$ gilt $qE \neq \emptyset$ (d.h. jedes q hat eine ausgehende Kante).*

Eine Partie ρ ist eine Folge von Zuständen $\rho = \rho_0\rho_1\rho_2\ldots$ mit $(\rho_i, \rho_{i+1}) \in E$. Auf einer unendlichen Partie lassen sich die beiden folgenden Zustandsmengen definieren:

- $Oc(\rho) = \{q \in Q \mid \exists i \; \rho(i) = q\}$ die Menge der in ρ vorkommenden Zustände - auch occurence set genannt
- $In(\rho) = \{q \in Q \mid \exists^\omega i \; \rho(i) = q\}$ die Menge der in ρ unendlich oft vorkommenden Zustände - auch infinity set genannt

Def. 2.2 (Gewinnbereiche) *Zu einem Spielgraphen G und einer Gewinnmenge Win_0 ($Win_0 \subseteq Q^\omega$ - Gewinnpartien von Spieler 0) sei*

$$W_0 := \{q \in Q \mid \textit{Spieler 0 hat eine Gewinnstrategie von } q\}$$

Analog ist W_1 für Spieler 1 definiert.

Eine Strategie bestimmt den folgenden Zug anhand der bisher gespielten Partie. Unter einer *Gewinnstrategie* ist zu verstehen, dass, wenn ein Spieler gemäß seiner Gewinnstrategie spielt, er einen Sieg von seinem Gewinnbereich erzwingen kann. Eine Gewinnstrategie ist dann *positional*, wenn die Entscheidung für den nächsten Zug nur von dem aktuellen Zustand abhängt. Neben positionalen Gewinnstrategien gibt es noch Automatenstrategien. Diese haben einen endlichen Speicher für den bisherigen Verlauf der Partie und treffen aufgrund dieser Informationen ihre Entscheidung. Solche Automatenstrategien lassen sich mittels endlichen Automaten mit Ausgabe realisieren.

Es gibt kein $q \in Q$, von dem sowohl Spieler 0 als auch Spieler 1 eine Gewinnstrategie haben. Wähle entsprechende Gewinnstrategien f, g von q aus für Spieler 0 bzw. 1. Da f Gewinnstrategie für Spieler 0 ist, folgt für die entstehende Partie $\rho \in Win_0$. Da g Gewinnstrategie für Spieler 1 ist, gilt $\rho \notin Win_0$ ↯. Daraus folgt die Eigenschaft, dass $W_0 \cap W_1 = \emptyset$ gilt.

Ein Spiel heißt dann *determiniert*, wenn jeder Knoten des Spielgraphen entweder zu W_0 oder zu W_1 gehört - also gilt $W_0 \dot{\cup} W_1 = Q$. Alle in dieser Diplomarbeit betrachteten Spiele erfüllen diese Bedingung.

2.1.1 Garantie- und Sicherheitsspiele

Def. 2.3 (Garantiespiel) *Sei $G = (Q, E)$ ein Spielgraph und $F \subseteq Q$. Ein Garantiespiel hat die Gewinnbedingung:*

$$\rho \in Win_0 :\Leftrightarrow Oc(\rho) \cap F \neq \emptyset$$

Ein Garantiespiel ist dadurch gekennzeichnet, dass der Besuch eines Zustandes aus der Endzustandsmenge F zum Gewinn der Partie ausreicht. Abbildung 2.2 zeigt eine Partie, die von Spieler 0 gewonnen wird. Denn die Partie sieht erst Zustände aus $Q \backslash F$, anschließend mindestens einen Zustand aus F und verbleibt dann in der Menge $Q \backslash F$.

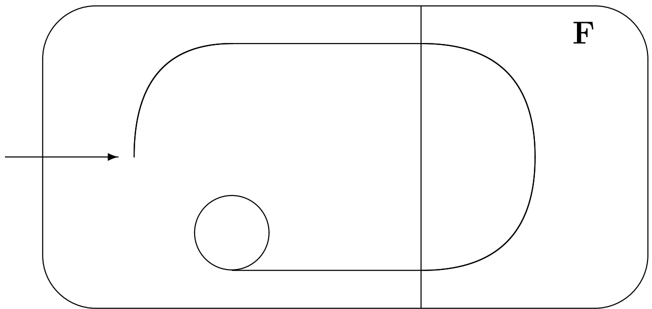


Abbildung 2.2: Charakterisierung von Garantiespielen

Satz 2.1 *Für Garantiespiele sind die Gewinnbereiche W_0 und W_1 von Spieler 0 bzw. Spieler 1 in Polynomzeit berechenbar, ebenso wie entsprechende positionale Gewinnstrategien.*

Beweis: Konstruiere für $i = 0, 1, 2, \ldots$ die die Mengen $Attr_0^i(F)$. Hierbei ist

$$Attr_0^i(F) := \{q \in Q \mid \text{von } q \text{ aus kann Spieler 0 in } \leq i \text{ Zügen den Besuch in } F \text{ erzwingen}\}$$

Induktive Definition:

$$\begin{aligned}
Attr_0^0(F) &:= F \\
Attr_0^{i+1}(F) &:= Attr_0^i(F) \cup \\
&\quad \{q \in Q_0 \mid \exists (q, r) \in E \text{ mit } r \in Attr_0^i(F)\} \cup \\
&\quad \{q \in Q_1 \mid \forall (q, r) \in E \text{ gilt } r \in Attr_0^i(F)\}
\end{aligned}$$

Schlussfolgerungen:

(i) $Attr_0^i(F) \subseteq Attr_0^{i+1}(F)$

(ii) Für ein $l \leq |Q|$ gilt: $Attr_0^l(F) = Attr_0^{l+1}(F)$

(iii) Setze $Attr_0(F) = \bigcup_{i=0}^{|Q|} Attr_0^i(F) \overset{(i)}{=} Attr_0^{|Q|}(F)$ (oder statt $|Q|$ auch l wie oben)
Die Menge $Attr_0(F)$ wird auch der 0-Attraktor von F genannt.

Wenn ausgedrückt werden soll, welcher Spielgraph der Attraktorberechnung zugrunde liegt, kann die Schreibweise $Attr_0^G(F)$ benutzt werden. Dabei wird dann explizit angegeben, dass der Spielgraph G für die Berechnung benutzt werden soll.

Die Behauptung ist, dass $W_0 = Attr_0(F)$ und $W_1 = Q \backslash Attr_0(F)$ gilt. Auf W_0 und W_1 existieren jeweils positionale Gewinnstrategien. Der Beweis ergibt sich aus den folgenden Bemerkungen. Definiere

$$dist(q, F) := \begin{cases} \min\{i \mid q \in Attr_0^i(F)\} & \text{falls } q \in Attr_0(F) \\ \infty & \text{sonst} \end{cases}$$

Bemerkungen

1. $\left.\begin{array}{l} q \in Attr_0(F)\backslash F,\ q \in Q_0 \Rightarrow \exists (q,r) \in E \text{ mit } dist(r,F) < dist(q,F) \\ q \in Attr_0(F)\backslash F,\ q \in Q_1 \Rightarrow \forall (q,r) \in E \text{ mit } dist(r,F) < dist(q,F) \end{array}\right\}$ Spieler 0 kann Besuch in F erzwingen

2. $\left.\begin{array}{l} q \notin Attr_0(F)\backslash F,\ q \in Q_0 \Rightarrow \forall (q,r) \in E \text{ gilt } r \notin Attr_0(F) \\ q \notin Attr_0(F)\backslash F,\ q \in Q_1 \Rightarrow \exists (q,r) \in E \text{ gilt } r \notin Attr_0(F) \end{array}\right\}$ Spieler 1 kann Besuch in F verhindern

Auf der Menge $Attr_0(F)$ kann Spieler 0 den Besuch von F erzwingen (Bemerkung 1), die dafür verwendete Strategie besteht darin, den Abstand zu F zu verringern. Dagegen kann Spieler 0 auf der Menge $Q \backslash Attr_0(F)$ keinen Besuch von $Attr_0(F)$ erzwingen (Bemerkung 2).

2.1.1.1 Sicherheitsspiel

Def. 2.4 (Sicherheitsspiel) *Sei $G = (Q, E)$ ein Spielgraph und $F \subseteq Q$. Ein Sicherheitsspiel hat die Gewinnbedingung:*

$$\rho \in Win_0 :\Leftrightarrow Oc(\rho) \subseteq F$$

Ein Sicherheitsspiel ist dadurch gekennzeichnet, dass nur Zustände aus F besucht werden dürfen, wie Abbildung 2.3 zeigt. Ein Sicherheitsspiel ist das komplementäre Spiel zu den Garantiespielen. Somit kann für die Lösung statt einem Sicherheits- auch ein Garantiespiel betrachtet werden, bei dem Spieler 0 und Spieler 1 vertauscht und $F' = Q \backslash F$ gesetzt worden ist.

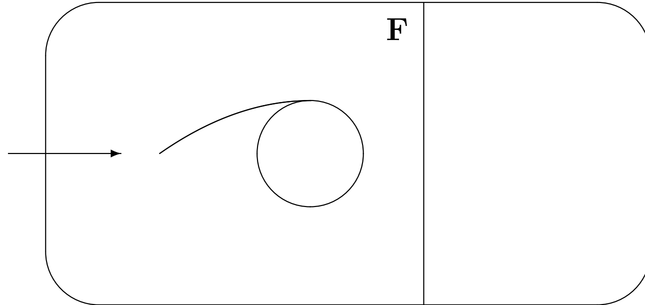


Abbildung 2.3: Charakterisierung von Sicherheitsspielen

2.1.1.2 Linearzeitalgorithmus für den Attraktor

Da naive Implementierungen für die Attraktor-Berechnung meistens in $\mathcal{O}(|Q|^3)$ liegen, sei hier noch kurz auf einen Linearzeitalgorithmus ([Tho00]) hingewiesen, siehe Algorithmus 2.1. Dieser berechnet den Attraktor in $\mathcal{O}(|Q + E|)$. Voraussetzung hierfür ist, dass der Spielgraph $G = (Q, E)$ durch Adjazenzlisten gegeben ist.

Paramater: $F \subseteq Q$, $G = (Q, E)$ durch Adjazenzlisten gegeben

```
queue:=∅
for each q ∈ Q
    vorg(q):=∅                                        5
    if q ∈ Q1 then count(q):=|nachf(q)|
for each q ∈ Q
    for each p ∈ nachf(q)
        vorg(p):=vorg(p) ∪ {q}
A:=F                                                  10
for each q ∈ F
    Enqueue(q,queue)
while (not Empty(queue))
    q:=Head(queue)
    Dequeue(queue)                                    15
    for each p ∈ vorg(q)
        if p ∉ A then
            if p ∈ Q0 then
                A := A ∪ {q}
                Enqueue(p,queue)                      20
            if p ∈ Q1 then
                count(p)=count(p)−1
                if count(p)=0 then
                    A := A ∪ {p}
                    Enqueue(p,queue)                  25
return A
```

Ergebnis: Attraktor−Menge (A)

Algorithmus 2.1: Linearzeitalgorithmus für den Attraktor

2.1.2 Schwache Paritätsspiele

Die im folgenden betrachteten schwachen Paritätsspiele stellen eine boolesche Kombination von Garantie- bzw. Sicherheitsbedingungen dar.

Def. 2.5 (Schwaches Paritätsspiel) *Sei $G = (Q, E)$ ein Spielgraph und c eine Abbildung $c : Q \to \{0, \ldots, k\}$, die auch Färbung genannt wird. Die Partie $\rho = \rho(0)\rho(1)\rho(2)\ldots$ liefert*

eine Farbfolge $c(\rho) = c(\rho(0))c(\rho(1))c(\rho(2))\dots$ *Die schwache Paritätsbedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \max(Oc(c(\rho)) \textit{ ist gerade}$$

Die Definition besagt, dass die maximal in der Partie ρ auftretende Farbe gerade sein muss, damit Spieler 0 diese gewinnt.

Garantiespiele über einen Spielgraphen G lassen sich als spezielle schwache Paritätsspiele über G mit der Färbung

$$c(q) = \begin{cases} 2 & \text{für } q \in F \\ 1 & \text{für } q \notin F \end{cases}$$

darstellen. Somit sind Garantie- und Sicherheitsspiele eine Teilmenge der schwachen Paritätsspiele.

Beispiel 2.1 *Von den mit + gekennzeichneten Knoten in Abbildung 2.4 hat Spieler 0 eine Gewinnstrategie, von den mit - gekennzeichneten Spieler 1. Spieler 0-Zustände sind durch Kreise dargestellt und Spieler 1-Zustände durch Rechtecke.*

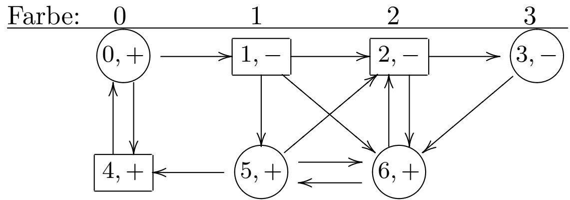


Abbildung 2.4: Beispiel schwaches Paritätsspiel

Satz 2.2 *Für schwache Paritätsspiele können die Gewinnbereiche* W_0, W_1 *berechnet und entsprechende positionale Gewinnstrategien angegeben werden.*

Beweis: Sei $G = (Q, E)$, $c : Q \to \{0, \dots, k\}$. Setze $C_i = \{q \in Q \mid c(q) = i\}$. Dann sind die folgenden Mengen $A_k, A_{k-1}, \dots, A_0$ berechenbar:

$$A_i := \begin{cases} - \text{ Spieler 0 kann Besuch der Farbe i erzwingen,} & \\ \quad \text{keine höhere ungerade Farben wird gesehen} & \text{falls } i \text{ gerade} \\ - \text{ Spieler 1 kann Besuch der Farbe i erzwingen,} & \\ \quad \text{keine höhere gerade Farbe wird gesehen} & \text{falls } i \text{ ungerade} \end{cases}$$

$$A_k := Attr^G_{k \bmod 2}(C_k)$$

$$A_i := \begin{cases} Attr^{G'}_0(C_i \backslash (A_{i+1} \cup \dots \cup A_k) & i \text{ gerade} \\ Attr^{G'}_1(C_i \backslash (A_{i+1} \cup \dots \cup A_k) & i \text{ ungerade} \end{cases}$$

$$\text{mit } G' = (Q', E') \text{ und } Q' = Q \backslash (A_{i+1} \cup \dots \cup A_k),\ E' = E|_{Q' \times Q'}$$

Somit gehört jeder Knoten zu genau einer Menge A_i.

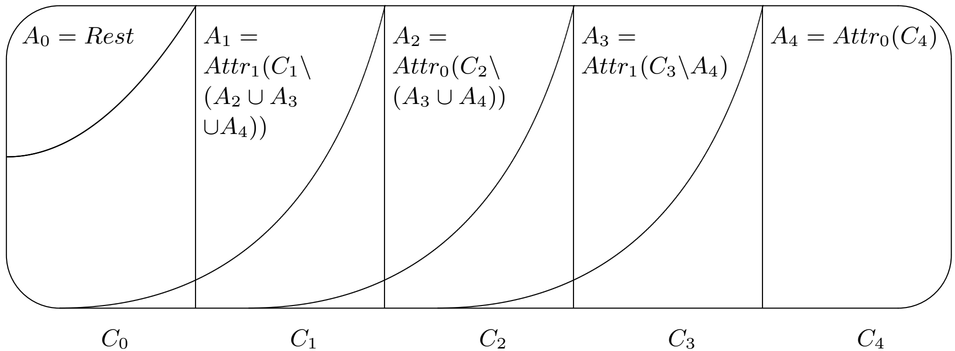


Abbildung 2.5: Veranschaulichung für k=4

Dann ist die Behauptung, dass gilt:

$$W_0 := \bigcup_{i \text{ gerade}} A_i \; , \; W_1 := \bigcup_{i \text{ ungerade}} A_i$$

Die Vereinigungen der Attraktor-Strategien liefern dann die positionalen Gewinnstrategien für Spieler 0 und Spieler 1 auf W_0 bzw. W_1. Diese Behauptung kann induktiv bewiesen werden, indem für $i = 0, \dots, k$ die folgende Aussage gezeigt wird:

$$\bigcup_{\substack{i=k-j \\ i \text{ gerade}}}^{k} A_i \subseteq W_0 \qquad \bigcup_{\substack{i=k-j \\ i \text{ ungerade}}}^{k} A_i \subseteq W_1$$

Induktionsanfang j=0: Sei k oBdA gerade. Dann ist $A_k = Attr_0(C_k) \subseteq W_0$ klar. Analog für k ungerade.

Induktionsschritt Betrachte $A_{k-(j+1)}$, wobei $k-(j+1)$ oBdA gerade ist und im folgenden mit i bezeichnet wird.

Zeige: von $q \in A_i$ gewinnt Spieler 0 mit Attraktorstrategie und somit ist $q \in W_0$. Die Anwendung der Attraktorstrategie liefert einen Besuch in $C_i \backslash (A_{i+1} \cup \ldots \cup A_k)$. Garantiere, dass anschließend nur noch Farben $\leq i$ besucht werden oder ein Übergang nach A_j mit geradem j und $j > i$ erfolgt (dann hat Spieler 0 nach Induktionsvoraussetzung gewonnen).

- Der Besuch von C_i erfolgt durch einen Knoten $q \in Q_0$. Dann ist es unmöglich, dass alle Ausgangskanten zu A_k-Mengen mit $k > i$ führen, denn dann wäre q in einer dieser Mengen. Also gibt es Kante nach einer A_k-Menge mit $k \leq i$. Diese Kante wird gewählt.

- $q \in Q_1$. Führen alle Ausgangskanten zu A_k-Mengen mit $k \leq i$, sind wir fertig. Für Kanten nach A_k mit geradem k und $k > i$ liefert die Induktionsvoraussetzung das Ergebnis. Eine Kante zu einer Farbe $> i$ in A_j mit ungeradem j zu wählen, ist unmöglich, da in diesem Fall q bereits im entsprechenden A_j enthalten wäre. $\square$

Zurückkommend auf das Beispiel 2.1 liefert das gerade gezeigte Lösungsverfahren die erwarteten Ergebnisse:

$$\begin{aligned}
A_3 &= Attr_1(\{3\}) = \{1,2,3\} \\
A_2 &= Attr_0(\{2,6\}\backslash\{1,2,3\}) = Attr_0(\{6\}) \text{ auf dem Teilspiel} = \{5,6\} \\
A_1 &= Attr_1(\{1,5\}\backslash\{1,2,3,5,6\}) = \emptyset \\
A_0 &= Attr_0(\{0,4\}\backslash\{1,2,3,5,6\}) = Attr_0(\{0,4\}) \text{ auf dem Teilspiel} = \{0,4\} \\
W_0 &= \bigcup_{i \text{ gerade}} A_i = \{0,4,5,6\} \\
W_1 &= \bigcup_{i \text{ ungerade}} A_i = \{1,2,3\}
\end{aligned}$$

2.1.3 Staiger-Wagner-Spiele

Auch die Staiger-Wagner Gewinnbedingung ist eine Kombination von Garantie- und Sicherheitsbedingungen. Diese ist daduch gekennzeichnet, dass in der Akzeptierkomponente genau die Zustandsmengen angegeben sind, die eine Partie besuchen kann und muss. Die Reduktion auf die schwachen Paritätsspiele zur Lösung der SW-Spiele zeigt, dass die schwachen Paritätsspiele von der Ausdrucksstärke ausreichen.

Def. 2.6 (Staiger-Wagner Spiel - [SW74]) *Sei $G = (Q, E)$ ein Spielgraph und $\mathcal{F} \subseteq Pot(Q)$. Die Staiger-Wagner Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow Oc(\rho) \in \mathcal{F}$$

Schwache Paritätsspiele sind ein Spezialfall der Staiger-Wagner Spiele. Zu einem gegebenen Spielgraphen $G = (Q, E)$ und einer Färbungsfunktion c kann ein $\mathcal{F} \subseteq Pot(Q)$ konstruiert werden, so dass $\rho \in Win_0 \Leftrightarrow Oc(\rho) \in \mathcal{F}$ gilt. Setze dazu $\mathcal{F} := \{P \subseteq Q | \max(c(P)) \text{ ist gerade}\}$.

Beispiel 2.2 $Q = \{0,1,2\}$, $\mathcal{F} = \{\{0,1,2\}\}$

0 1 2

Alle bisher betrachteten Spiele hatten positionale Gewinnstrategien. Bei Staiger-Wagner Spielen ist Gedächtnis über den bisherigen Spielverlauf für die Strategiekonstruktion nötig, siehe dazu Beispiel 2.2. Beide möglichen positionalen Strategien sind keine Gewinnstrategien auf $W_0 = \{0,1,2\}$.

Um bei Staiger-Wagner Spielen die Gewinnbereiche für Spieler 0 bzw. 1 sowie deren Gewinnstrategien bestimmen zu können, werden diese auf schwache Paritätsspiele reduziert.

Def. 2.7 (Spielreduktion) *Seien $G = (Q, E)$ und $G' = (Q', E')$ Spielgraphen mit Gewinnbedingungen Win_0 und Win'_0. Das Spiel (G, Win_0) ist reduzierbar auf (G', Win'_0), geschrieben als $(G, Win_0) \leq (G', Win'_0)$, genau dann, wenn folgende Bedingungen erfüllt sind:*

1. *$Q' = Q \times S$ für eine endliche Menge S.*

2. *Jede Partie ρ über G wird in die Partie ρ' über G' überführt durch*
   (a) *eine Funktion $f : Q \to Q \times S$ (der Anfang von ρ'),*
   (b) *$\forall (q, s) \in Q \times S$, $\forall p$ gilt $(q, p) \in E \Rightarrow$ ex. genau ein s' mit $((q, s), (p, s')) \in E'$*
   (c) *$\forall ((q, s), (p, s')) \in E'$ gilt $(q, p) \in E$*
3. *Für ρ und ρ' gemäß 2. gilt: $\rho \in Win_0 \Leftrightarrow \rho' \in Win'_0$.*

Satz 2.3 *Es gelte $(G, Win_0) \leq (G', Win'_0)$, und über den Gewinnbereichen W'_0 und W'_1 von G' gewinne Spieler 0 bzw. Spieler 1 mit einer positionalen Gewinnstrategie. Dann lassen sich aus W'_0, W'_1 für (G, Win_0) die Gewinnbereiche W_0, W_1 bestimmen und für $q \in W_0$ (bzw. W_1) eine entsprechende Automatenstrategie angeben.*

Beweis: Zu $q \in Q$ bestimme $f(q) \in Q'$ und prüfe, ob $f(q) \in W'_0$. Ist dies der Fall gilt $q \in W_0$ und es ex. eine Automatenstrategie. Analog kann für Spieler 1 vorgegangen werden.

Satz 2.4 *Staiger-Wagner Spiele lassen sich auf schwache Paritätsspiele reduzieren und somit kann man die Gewinnbereiche und entsprechende Automatenstrategien für SW-Spiele berechnen.*

Beweis: Sei (G, Win_0) ein SW-Spiel, definiert durch die Familie $\mathcal{F} \subseteq Pot(Q)$ mit $\rho \in Win_0 \Leftrightarrow Oc(\rho) \in \mathcal{F}$. Dann kann ein schwaches Paritätsspiel (G', Win'_0) angegeben werden mit $(G, Win_0) \leq (G', Win'_0)$. Die Idee dabei ist, sich die bereits besuchten Knoten zu merken. Somit haben die Knoten von G' die Form (p, R) mit $R \subseteq Q$. Definiere also $G' = (Q', E')$ und Färbung $c : Q' \to \{0, \ldots, k\}$ wie folgt:

$$\begin{aligned} Q' &= Q \times Pot(Q) \\ E' &= \{((p, R), (q, R \cup \{q\})) \mid (p, q) \in E,\ R \subseteq Q\} \\ c(p, R) &= \begin{cases} 2 \cdot |R| & \text{falls } R \in \mathcal{F} \\ 2 \cdot |R| - 1 & \text{falls } R \notin \mathcal{F} \end{cases} \end{aligned}$$

Für die Bedingungen 1. und 2. der Definition der Spielreduktion (2.7) setze $S = Pot(Q)$ und $f : q \to (q, \{q\})$. Für 3. betrachte eine Partie ρ über G und dazu die induzierte Partie ρ' über G'. In ρ wird irgendwann ein Präfix erreicht, welches alle Zustände aus $Oc(\rho)$ enthält. In ρ' ist von dort an die 2. Komponente konstant ($=Oc(\rho)$). Die maximale besuchte Farbe ist also gerade gdw. $Oc(\rho) \in \mathcal{F}$, d.h. $\rho \in Win_0 \Leftrightarrow \rho' \in Win'_0$.

2.1.4 Request-Response-Spiele

Grundlage für die hier vorgestellten Request-Response-Spiele bilden die in Abschnitt 2.1.8 vorgestellten Streett-Spiele. Die Gewinnbedingung dort lautet, dass, wenn unendlich oft ein F_i besucht wird, auch unendlich oft das passende E_i besucht werden muss. In der Realität ist das aber ein eher selteneres Beispiel. Wenn ein Aufzug anfordert wird, soll er auf jeden Fall kommen, und nicht nur, wenn unendlich oft der Taster gedrückt wurde. Ein ähnlicher Fall ist der Doppelklick auf ein Programm-Symbol auf dem Desktop.

Dieses Unterkapitel basiert auf der Diplomarbeit “Automatische Synthese optimaler Controller für request-response-Spezifikationen“ von Patrick Hütten [Hüt03].

Def. 2.8 (Request-Response Spiel - [Hüt03]) *Sei $G = (Q, E)$ ein Spielgraph, $P_i, R_i \subseteq Q$ für $1 \leq i \leq r$ mit $r \in \mathbb{N}$. Die Request-Response Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigwedge_{i=1}^{r} \bigwedge_{j} \rho(j) \in P_i \Rightarrow \exists_{j' \geq j}\ \rho(j') \in R_i$$

In LTL[1] ausgedrückt ergibt sich:

$$\rho \in Win_0 :\Leftrightarrow \bigwedge_{i=1}^{r} G(P_i \rightarrow F\ R_i)$$

Eine RR-Partie wird dann von Spieler 0 gewonnen, wann immer, wenn ein Zustand aus einer Anforderungsmenge (P_i) besucht wird, anschließend ein Zustand aus der passenden Antwortmenge (R_i) gesehen wird.

Satz 2.5 ([Hüt03]) *Request-Response Spiele lassen sich auf Büchi-Spiele reduzieren und somit lassen sich die Gewinnbereiche und entsprechende Automatenstrategien berechnen.*

Idee: Zur Bestimmung der Gewinnbereiche und -strategie werden die RR-Spiele auf Büchi-Spiele reduziert. Dazu werden in dem erweiterten Spielgraphen die geforderten Bedingungen gespeichert sowie eine Markierung, welche Bedingung als nächstes erfüllt werden soll. Wird diese Bedingung erfüllt, wird ein Endzustand besucht und die Markierung wechselt zur nächst höheren.

Beweis: Sei (G, Win_0) ein RR-Spiel, definiert durch die Mengen $P_i, R_i \subseteq Q$ für $1 \leq i \leq r$ mit $r \in \mathbb{N}$. Falls $r = 1$ gilt, füge Bedingungspaar (P_2, R_2) mit $P_2 = Q_2 = \emptyset$ hinzu. Dann kann ein Büchi-Spiel (G', Win'_0) angegeben werden mit $(G, Win_0) \leq (G', Win'_0)$. Definiere $G' = (Q', E')$ und Endzustandsmenge F folgendermaßen:

- $Q' := Q \times Pot\{1, \ldots, r\} \times \{1, \ldots, r\} \times \{0, 1\}$
- $((q, M, m, f), (q', M', m', f')) \in E' \Leftrightarrow$
  - $(q, q') \in E$
  - $M' = (M \cup \{i \mid q' \in P_i\}) \setminus \{i \mid q' \in R_i\}$
  - $m' = \begin{cases} m & \text{falls } m \in M' \\ m+1 & \text{falls } m \notin M' \text{ und } m < r \\ 1 & \text{falls } m \notin M' \text{ und } m = r \end{cases}$
  - $f' = \begin{cases} 0 & \text{falls } m = m' \\ 1 & \text{falls } m \neq m' \end{cases}$
- $F = Q \times Pot\{1, \ldots, r\} \times \{1, \ldots, r\} \times \{1\}$

Die Funktion $f : Q \rightarrow Q'$ ist definiert als: $f : q \rightarrow (q, \{i \mid q \in P_i\}, 1, 0)$

Damit die zweite Bedingung einer Spielreduktion (Definition 2.7) erfüllt ist, muss nur noch der Unterpunkt b) gezeigt werden. Dieser Punkt ist aber auch erfüllt, da M' eindeutig aus M, m' eindeutig aus m und f' eindeutig aus f entsteht.

[1] Linear Temporal Logic

Behauptung: $\rho \in Win_0 \Leftrightarrow \rho' \in Win'_0$

$\rho' \notin Win'_0$

$\iff$ $\exists j_0 \in \mathbb{N} : \forall j > j_0$ gilt: $\rho'(j) \notin F$

$\iff$ $\exists j_0 \in \mathbb{N} : \forall j > j_0$ gilt: $\rho'(j) \notin \{(q, M, m, 1) \mid q \in Q, M \subseteq \{1, \ldots, r\}, 0 \leq m \leq r\}$

$\iff$ $\exists j_0 \in \mathbb{N} : \forall j > j_0$ mit $\rho'(j) = (q, M, m, f)$ gilt: $f = 0$

$\iff$ $\exists j_0 \in \mathbb{N} : \forall j > j_0$ mit $\rho'(j) = (q, M, m, f), \rho'(j+1) = (q', M', m', f')$ gilt: $m = m'$ beachte, dass $r \geq 2$ gilt)

$\iff$ $\exists j_0 \in \mathbb{N}$ mit $\rho'(j_0) = (q_0, M_0, m_0, f_0)$, so dass $\forall j \geq j_0$ mit $\rho'(j) = (q', M', m', f')$ gilt: $m_0 = m'$

$\iff$ $\exists j_0 \in \mathbb{N}$ mit $\rho'(j_0) = (q_0, M_0, m_0, f_0)$, so dass $\forall j \geq j_0$ mit $\rho'(j) = (q', M', m', f')$ gilt: $m_0 \in M_0 \cup M'$

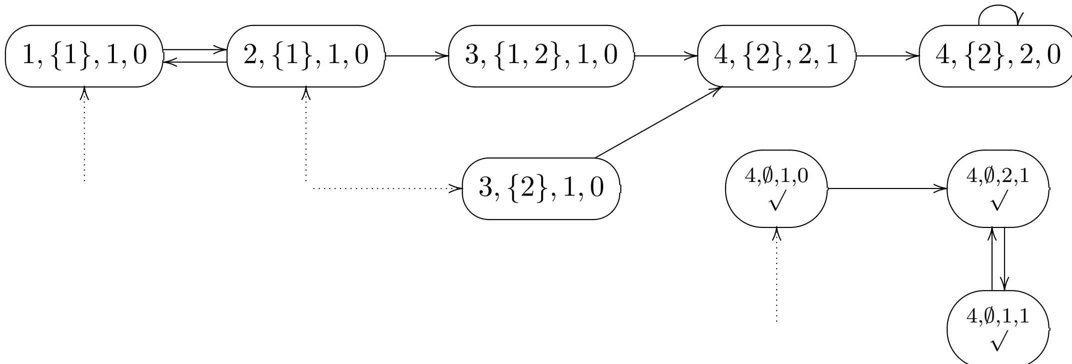
$\overset{*}{\iff}$ $\exists j_1 \leq j_0, 1 \leq m_0 \leq r$ mit $\rho(j_1) \in P_{m_0}$ und $\forall j \geq j_1$ gilt: $\rho(j) \notin Q_{m_0}$

$\iff$ $\rho \notin Win_0$

Bemerkung zu $\overset{*}{\Leftarrow}$: Ist $\rho'(j_0) = (v_0, M_0, m'_0, f_0)$ mit $m'_0 \neq m_0$, so gilt auch für $P_{m'_0}$, dass es zu einem gewissen Zeitpunkt gefordert und nie durch einen Besuch von $Q_{m'_0}$ erfülllt wird. Wähle in diesem Fall $m_0 := m'_0$. □

Beispiel 2.3 *Gegeben sei der folgende Spielgraph und die Mengen $P_1 = \{1, 2\}$, $R_1 = \{4\}$, $P_2 = \{3\}$ und $R_2 = \emptyset$.*

Der daraus resultierende Spielgraph G' (gemäß oben angegebener Konstruktion) sieht dann so aus - dabei ist der Gewinnbereich des Spielers 0 auf dem Büchi-Spiel bereits in den Graphen eintragen und die Zustände $f(q)$ mit $q \in Q$ sind mit gepunkteten Pfeilen markiert:



Das Ergebnis des Büchi-Spieles eingeschänkt auf die Zustände $f(q)$ mit $q \in Q$ liefert dann das Ergebnis des RR-Spieles. Somit ergeben sich die Gewinnbereiche zu:

$$W_0 = \{4\},\ W_1 = \{1, 2, 3\}$$

2.1.5 Büchi-Spiele

Bei allen folgenden Spielen macht die Gewinnbedindung keine Aussage mehr über die einzelnen besuchten Zustände (occurence set), sondern nur noch über den unendlich oft besuchten (infinity set), d.h. es werden jetzt Spiele der zweiten Stufe der Borel-Hierarchie betrachtet,

wo vorher nur welche der ersten Stufe betrachtet worden sind. Somit beeinflussen nur endlich oft gesehene Zustände den Gewinn der Partie nicht. Das hier vorgestellte Büchi-Spiel ist das Pendant zum Garantiespiel auf dem infinity set.

Def. 2.9 (Büchi Spiel - [Büc62]) *Sei $G = (Q, E)$ ein Spielgraph und $F \subseteq Q$. Die Büchi-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow In(\rho) \cap F \neq \emptyset$$

Damit Spieler 0 die Partie gewinnt, müssen immer wieder Zustände aus der Endzustandsmenge F besucht werden. Dies ist in Abbildung 2.6 dargestellt - zuerst werden Zustände aus $Q \backslash F$ besucht und anschließend geht die Partie in eine Schleife über, bei der bei jedem Durchlauf mindestens ein Zustand aus F gesehen wird.

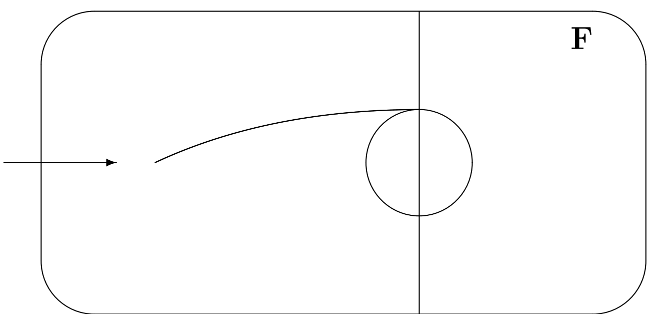


Abbildung 2.6: Charakterisierung von Büchi-Spielen

Satz 2.6 *In Büchi-Spielen sind die Gewinnbereiche W_0, W_1 berechenbar und Spieler 0 bzw. Spieler 1 haben positionale Gewinnstrategien.*

Beweis: Die Idee zur Lösung ist, Mengen $Recur_0^i$ zu bestimmen. Dabei sind in der Menge $Recur_0^i$ genau die Zustände $q \in F$, von denen Spieler 0 mindestens i Wiederbesuche in F erreichen kann. Somit gilt:

$$F \supseteq Recur_0^1 \supseteq Recur_0^2 \supseteq \dots$$

Zur Vorbereitung wird dazu erstmal eine Abwandlung des bereits bekannten Attraktors eingeführt:

$$Attr_0^+(F) := \{q \in Q| \text{ von } q \text{ kann Spieler 0 Besuch in } F \text{ in } \geq 1 \text{ Schritten erzwingen}\}$$

$$\begin{aligned}
& A_0^0 && = \emptyset \\
& A_0^1 && = \{q \in Q_0 \mid \exists (q,r) \in E : r \in F\} \cup \{q \in Q_1 \mid \forall (q,r) \in E : e \in F\} \\
i \geq 1 \quad & A_0^{i+1} && = A_0^i \cup \{q \in Q_0 \mid \exists (q,r) \in E : r \in A_0^i\} \cup \\
& && \quad \{q \in Q_1 \mid \forall (q,r) \in E : r \in (A_0^i \cup F)\} \\
& Attr_0^+(F) && = \bigcup_{i \geq 0} A_0^i
\end{aligned}$$

Analog zum Attraktor (siehe Seite 7) gilt auch beim Attraktor+:

(i) $A_0^i \subseteq A_0^{i+1}$

(ii) Für ein $l \leq |Q|$ gilt $A_0^l = A_0^{l+1}$

(iii) Setze $Attr_0^+(F) \overset{(ii)}{=} \bigcup_{i=0}^{|Q|} A_0^i \overset{(i)}{=} A_0^{|Q|}$ (oder statt $|Q|$ auch l wie oben)

Die positionalen Strategien vom Attraktor werden übernommen. Damit lässt sich jetzt auch $Recur_0(F)$ induktiv definieren:

$$\begin{aligned} Recur_0^0(F) &:= F \\ Recur_0^{i+1}(F) &:= F \cap Attr_0^+(Recur_0^i(F)) \\ Recur_0(F) &:= \bigcap_{i \geq 0} Recur_0^i(F) \end{aligned}$$

Schlussfolgerungen:

(i) $Recur_0^{i+1}(F) \subseteq Recur_0^i(F)$

(ii) Für ein $l \leq |Q|$ gilt $Recur_0^{l+1}(F) = Recur_0^l(F)$

(iii) Setze $Recur_0(F) \overset{(ii)}{=} \bigcap_{i=0}^{|Q|} Recur_0^i(F) \overset{(i)}{=} Recur_0^{|Q|}(F)$ (oder statt $|Q|$ auch l wie oben)

Auf $Recur_0(F) \cap Q_0$ ist eine Kantenauswahl zurück nach $Recur_0(F)$ möglich und auf $Recur_0(F) \cap Q_1$ gibt es nur noch Kanten zurück nach $Recur_0(F)$. Somit gilt $Attr_0(Recur_0(F)) \subseteq W_0$. Auf $Q \backslash Attr_0(Recur_0(F))$ hat Spieler 1 eine positionale Gewinnstrategie. Daraus folgt:

$$\begin{aligned} W_0 &:= Attr_0(Recur_0(F)) \\ W_1 &:= Q \backslash Attr_0(Recur_0(F)) \end{aligned}$$

Beispiel 2.4 $Q = \{1, 2, 3, 4\}, F = \{2\}$

$$\begin{aligned} Attr_0^+(\{2\}) : A_0^0 &= \emptyset \\ A_0^1 &= \{4, 5\} \\ A_0^2 &= \{2, 4, 5\} \\ A_0^3 &= \{2, 4, 5\} \\ Recur_0^0(\{2\}) &= F = \{2\} \\ Recur_0^1(\{2\}) &= \{2\} \cap \{2, 4, 5\} = \{2\} \\ Attr_0(\{2\}) &= \{2, 4, 5\} \\ W_0 &= \{2, 4, 5\} \\ W_1 &= Q \backslash W_0 = \{1, 3\} \end{aligned}$$

2.1.6 Paritätsspiele

Paritätsspiele nehmen in der theoretischen Informatik eine gewisse Rolle ein, da sich einerseits andere Spiele darauf reduzieren lassen (wie z.B. Muller- und Streett-Spiele - siehe die folgenden beiden Unterkapitel), sie aber andererseits auch eine Bedeutung für das Model Checking haben. Denn nach [EJS93] ist ein Paritätsspiel polynomiell zeitäquivalent zum Model-Checking für den modalen μ-Kalkül und daher können Verbesserungen bei den Algorithmen zu Paritätsspielen nach [Jur00] zu besseren Model-Checking-Werkzeugen führen, was ein wichtiges Ziel in der computergestützten Verifikation ist.

Def. 2.10 (Paritätsspiel - [Mos84]) *Sei $G = (Q, E)$ ein Spielgraph und c eine Abbildung $c : Q \to \{0, \ldots, k\}$, die auch Färbung genannt wird. Die Partie $\rho = \rho(0)\rho(1)\rho(2)\ldots$ liefert eine Farbfolge $c(\rho) = c(\rho(0))c(\rho(1))c(\rho(2))\ldots$ Die Paritätsbedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \max(In(c(\rho)) \textit{ ist gerade}$$

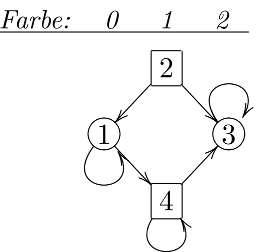
Paritätsspiele sind das Pendant zu den schwachen Paritätsspielen mit dem Unterschied, dass die Gewinnbedingung nicht mehr das occurence set sondern das infinity set benutzt. Dies bedeutet, dass nur Zustände, die unendlich oft besucht werden, einen Einfluss auf den Gewinn der Partie haben.

Eine in der Literatur verbreite Variante der Paritätsbedingung ist die Rabinkettenbedingung. Diese ist eine spezielle Form der Rabin-Bedingung - siehe Kapitel 2.1.8. Sie wird durch eine Menge $(E_1, F_1, \ldots, E_m, F_m)$ definiert, für die gilt $E_1 \subseteq F_1 \subseteq \ldots \subseteq E_m \subseteq F_m (\subseteq Q)$. Spieler 0 gewinnt eine solche Partie, wenn es ein $i \in |m|$ gibt, so dass die unendlich oft vorkommenden Knoten nicht alle in E_i, aber alle in F_i enthalten sind.

Satz 2.7 *Sei $G = (Q, E)$ ein Spielgraph mit Färbung $c : Q \to \{0, \ldots, k\}$ und Gewinnbedingung $\rho \in Win_0 \Leftrightarrow \max(In(c(\rho)))$ gerade. Dann kann man die Gewinnbereiche W_0, W_1 (mit $W_0 \cup W_1 = Q$) und entsprechende positionale Gewinnstrategien für Spieler 0 bzw. Spieler 1 effektiv bestimmen.*

Beweis: In [Tho95] kann ein induktiver Beweis nachgelesen werden. Der Beweis hat das von McNaughton entwickelte Verfahren, welches in [McN93] beschrieben ist, als Grundlage. Das Verfahren basiert darauf, rekursiv die Strategiekonstruktion für Teilspiele vorzunehmen und die Ergebnisse geeignet zusammenzufügen. An dieser Stelle sei auf den Algorithmus 2.2 - entnommen in [Vög00] - verwiesen, der diesen Ansatz umsetzt.

Beispiel 2.5 *Gegeben ist der folgende, gefärbte Spielgraph:*



Paramater: Q_i Knoten des Spielers i , E Kantenrelation , c Farbfunktion

---

$parity(Q_0, Q_1, E, c):$
if $(Q_0 \cup Q_1 = \emptyset)$ **then**
  return $(\emptyset, \emptyset, \emptyset)$ 5
else if odd$(\max_{v \in Q_0 \cup Q_1} c(v))$ **then**
  $(W_1, W_0, \sigma) := parity(Q_1, Q_0, E, c+1)$
  return (W_0, W_1, σ)
else
  select $p \in c^{-1}(\max_{v \in Q_0 \cup Q_1} c(v))$ 10
  $(R, \rho) := attract(Q_0, E, Q_0 \cup Q_1, \{p\})$
  $\bar{Q} := Q_0 \cup Q_1 \backslash R$
  $(W_0, W_1, \sigma) := parity(Q_0 \cap \bar{Q}, Q_1 \cap \bar{Q}, E \cap (\bar{Q} \times \bar{Q}), c)$
  if $((p \in Q_0 \wedge \exists (p,q) \in E : q \notin W_1) \vee (p \in Q_1 \wedge \forall (p,q) \in E : q \notin W_1))$ **then**
    if $p \in Q_1$ **then** 15
      return $(W_0 \cup R, W_1, \rho \cup \sigma)$
    else
      select $q \in \{q | \exists (p,q) \in E : q \notin W_1\}$
      return $(W_0 \cup \{p\} \cup R, W_1, \rho \cup \{(p \mapsto q\} \cup \sigma)$
  else 20
    $(R', \rho') := attract(Q_1, E, Q_0 \cup Q_1, W_1)$
    $\bar{Q} := Q_0 \cup Q_1 \backslash R'$
    $(U_0, U_1, \sigma') := parity(Q_0 \cap \bar{Q}, Q_1 \cap \bar{Q}, E \cap (\bar{Q} \times \bar{Q}), c)$
    return $(U_0, U_1 \cup R', \sigma|_{W_1} \cup \rho' \cup \sigma')$

25

Ergebnis: Lösung des Paritätsspieles
(Gewinnbereich Spieler 0 , Gewinnbereich Spieler 1 , Gewinnstrategie)

Algorithmus 2.2: McNaughtons Algorithmus

Der McNaughtons-Algorithmus zur Lösung des Paritätsspiels liefert auf dem gegebenen Spielgraphen folgenden Lauf (dabei entspricht jeder folgende Punkt einem Aufruf der parity-Funktion):

1. *Berechnung des Attraktors zum Knoten 3:* $Attr_0(\{3\}) = \{3\}$
2. *Da die größte noch nicht betrachtete Farbe ungerade ist, wird die parity-Funktion aufgerufen, wobei die Spieler 0 bzw. 1 Knoten vertauscht wurden und 1 zur Farbfunktion addiert wird (Zeilen 6-8 des Algorithmus).*
3. *Berechnung des Attraktors zum Knoten 4 auf dem verbliebenen Restgraphen (Unterschied zum gegebenen liegt darin, dass die Knoten des* $Attr_0(\{3\})$ *herausgenommen wurden):* $Attr_0(\{4\}) = \{4\}$
4. *Berechnung des Attraktors zum Knoten 2 auf dem Restgraphen mit den Knoten {1,2}:*

$Attr_0(\{2\}) = \{2\}$

5. *analog zu 2.*
6. *Berechnung des Attraktors zum Knoten 1: $Attr_0(\{1\}) = \{1\}$*

Da der vierte Aufruf der parity-Funtkion die Bedingung in Zeile 14 des Algorithmus nicht erfüllt, wird dort der Attraktor zum Gewinnbereich des Spielers 1 auf dem Restspiel berechnet, also wird $Attr_1(\{1\})$={1,2} auf dem Graphen mit den Knoten {1,2} berechnet. Alle anderen Aufrufe der parity-Funtkion habe die Bedingung erfüllt. Somit ergeben sich die Gewinnbereiche zu: $W_0 = \{1, 2, 3\}$ und $W_1 = \{4\}$.

2.1.7 Muller-Spiele

Die Muller-Gewinnbedingung (wie auch die Rabin- bzw. Streett-Gewinnbedingung) sind von besonderem Interesse, da sie sich dazu eignen, viele Eigenschaften von nebenläufigen System zu spezifieren, wie z.B. Fairness-Bedingungen, siehe auch [MP92].

Def. 2.11 (Muller Spiel - [Mul63]) *Sei $G = (Q, E)$ ein Spielgraph und $\mathcal{F} \subseteq Pot(Q)$. Die Muller-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow In(\rho) \in \mathcal{F}$$

Somit sind die Muller-Spiele auf dem infinity set das Pendant zu den Staiger-Wagner-Spielen auf dem occurence set.

Satz 2.8 ([BL69]) *In einem Mullerspiel über einem endlichen Graphen $G = (Q, E)$ kann man die Gewinnbereiche W_0, W_1 effektiv bestimmen und für $q \in W_i$ jeweils eine Automaten-Gewinnstrategie für Spieler i von q aus konstruieren.*

Beweis: Reduktion auf Paritätsspiele mittels LAR-Konstruktion:
Grundlage bilden die „latest appearence records“ LARs. Dabei handelt es sich um Listen von Zuständen, bei denen die letztbesuchten Zustände vorne stehen. Sie sind unter dem Namen „later appearence record“ LAR in [GH82], als „order-vector“ in [Büc83] und als „latest visitation record“ in [McN93] zu finden. Die Grundidee dieses Ansatzes geht auf einen technischen Report von McNaughton aus dem Jahre 1965 zurück ([**?**]).

Def. 2.12 *Zu $Q = \{1, \ldots, n\}$ sei $LAR(Q)$ die Menge der Paare $((i_1, \ldots, i_k), h)$ mit nicht leeren Listen von $i_1 \ldots i_k$ paarweise verschiedenen Zuständen und $h \in \{0, \ldots, n\}$ (=Hit).*

Notation für $h > 0$: $i_1 \ldots \underline{i_h} \ldots i_k$. $\{i_1, \ldots, i_h\}$ wird Hitabschnitt von $(i_1 \ldots i_k, h)$ genannt und $(i_{h+1} \ldots i_k)$ der Vergangenheitsanteil.

Damit lässt sich die Reduktion auf Paritätsspiele wie folgt definieren:

- $Q' = Q \times S = LAR(Q)$ (d.h. S=Menge der Vergangenheitsanteile von $LAR(Q)$ $\times$ Menge $\{0, \ldots, |Q|\}$ der Hitwerte)

- $f : q \to (q, \varepsilon)$ mit Hitwert 0
- $((q,s),(r,s')) \in E' :\Leftrightarrow$ LAR s' ergibt sich aus LAR s durch Voranstellen bzw. Vorziehen von r.
- Definition der Färbung $c : LAR(Q) \to \{0, \ldots, 2 \cdot |Q|\}$ durch

$$c(q,s) = \begin{cases} 0 & \text{falls } Hit(qs) = 0 \\ 2h & \text{falls } Hit(qs) > 0 \text{ und Hitabschnitt } \in \mathcal{F} \\ 2h-1 & \text{falls } Hit(qs) > 0 \text{ und Hitabschnitt } \notin \mathcal{F} \end{cases}$$

Nach dem Satz über die Transformation von Muller- in Paritätsautomaten ([Tho96]) gilt (*) $In(\rho) \in \mathcal{F} \Leftrightarrow \max(In(c(\rho')))$ ist gerade. Somit gilt:

$$\rho \in Win_0 \Leftrightarrow In(\rho) \in \mathcal{F} \overset{(*)}{\Leftrightarrow} \max(In(c(\rho'))) \text{ gerade } \Leftrightarrow \rho' \in Win'_0$$

2.1.8 Rabin- und Streett-Spiele

Def. 2.13 (Rabin-Spiel - [Rab69, Rab72]) *Sei $G = (Q, E)$ ein Spielgraph und*

$$\Omega = ((E_1, F_1), (E_2, F_2), \ldots, (E_r, F_r))$$

mit $E_i, F_i \subseteq Q$. Die Rabin-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch

$$\rho \in Win_0 :\Leftrightarrow \bigvee_{i=1}^{r} (In(\rho) \cap E_i = \emptyset \wedge In(\rho) \cap F_i \neq \emptyset)$$

Def. 2.14 (Streett-Spiel - [Str82]) *Sei $G = (Q, E)$ ein Spielgraph und $\Omega = ((E_1, F_1), \ldots, (E_r, F_r))$ wie oben. Die Streett-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigwedge_{i=1}^{r} (In(\rho) \cap F_i \neq \emptyset \to In(\rho) \cap E_i \neq \emptyset)$$

Rabin- und Streettbedingung weisen komplementäres Verhalten auf, wie schon die Garantie- und Sicherheitsbedingung. Für jede Partie ρ und eine gegebene Menge von Paaren von Knotenmengen $(E_1, F_1), \ldots, (E_r, F_r)$ mit $E_i, F_i \subseteq Q$ gilt, dass ρ die Rabin-Bedingung genau dann erfüllt, wenn ρ die Streett-Bedingung nicht erfüllt.

Satz 2.9 *In einem Rabin-Spiel gewinnt Spieler 0 von seinem Gewinnbereich aus mit einer positionalen Gewinnstrategie.*

Beweis: siehe [Kla94]

Satz 2.10 *Zu einem Streett-Spiel* (G, Win_0) *über* $G = (Q, E)$ *mit den Paaren* (E_1, F_1), ..., (E_r, F_r) *kann man ein Paritätsspiel* (G', Win'_0) *konstruieren, mit* $(G, Win_0) \leq (G', Win'_0)$. *Somit können die Gewinnbereiche berechnet und für die Spieler Automaten-Gewinnstrategien angegeben werden.*

Beweis: Der Beweis benutzt die Datenstruktur „index appearence record" IAR ([Saf92]) über r für die Reduktion von Streett-Spielen mit r Paaren auf Paritätsspiele.

Def. 2.15 *Ein IAR über* r *ist ein Tripel* (π, e, f) *mit* π *ist Permutation von* $(1 \ldots r)$ *und* $e, f \in \{1, \ldots, r\}$. *IAR(r) := Menge der IAR über* r.

Reduktion der Streett-Spiele auf Paritätsspiele:

- $Q' = Q \times IAR(r)$
- $f : q \rightarrow (q, ((1 \ldots r), r, r))$
- Zu einem IAR $(q, (\pi, e, f))$ führe für jede Kante $(q, q') \in E$ die Kante $((q, (\pi, e, f)), (q', (\pi', e', f'))) \in E'$ ein. Dabei entstehe π' aus π durch die Verschiebung aller j mit $q' \notin E_j$ nach rechts. Für $\pi' = (j_1, \ldots, j_r)$ setze e' := letztes k mit $q' \in E_{j_k}$ und f' := letztes k mit $q' \in F_{j_k}$.
- Die Färbung $c : IAR(r) \rightarrow \{1, \ldots, 2r\}$ ist definiert durch
$$c((q, ((j_1 \ldots j_r), e, f))) = \begin{cases} 2e & \text{falls } e \geq f \\ 2f - 1 & \text{falls } e < f \end{cases}$$

2.1.9 Übersicht über die vorgestellten Spiele

Tabelle 2.1 fasst noch einmal die hier vorgestellten Spiele zusammen. Dabei ist in Spalte 3 angegeben, ob die Gewinnstrategien positional sind, und in Spalte 4, ob die Gewinnbedingung eine Aussage übers occurence oder infinity set macht.

Spiel (kompl. Spiel)	Lösungsmethode	pos.	In/Oc
Garantiespiel (Sicherheitsspiel)	Attraktor	$\surd$	Oc
schwaches Paritätsspiel	mittels Attraktor-Berechnung	$\surd$	Oc
Staiger-Wagner-Spiel	Reduktion auf schwaches Paritätsspiel		Oc
Request-Response-Spiel	Reduktion auf Büchi-Spiel		Oc
Büchi-Spiel	Lösung mittels *Attr*$^+$ und *Recur*	$\surd$	In
Paritätsspiel	McNaughtons Algorithmus	$\surd$	In
Muller-Spiel	Reduktion auf Paritätsspiel mittels LAR		In
Rabin-Spiel (Street-Spiel)	Reduktion auf Paritätsspiel mittels IAR		In

Tabelle 2.1: Übersicht über die vorgestellten Spiele

2.2 Binäre Entscheidungsbäume (Binary Decision Diagrams)

Da im nächsten Kapitel ein symbolischer Zustandsraum betrachtet wird, wo Zustände und Transitionen durch boolesche Formeln dargestellt werden, wird eine kompakte Darstellung und effiziente Möglichkeit zur Manipulation boolescher Formeln benötigt. Dabei fiel die Wahl auf die binären Entscheidungsdiagramme, kurz BDDs (binary decision diagrams). Diese wurden erstmals 1959 von Lee [Lee59] eingeführt und später von Akers [Ake78] und Moret [Mor82] weiter verbreitet. Dabei basiert die Grundidee bei den BDDs auf der Shannon-Entwicklung boolescher Funktionen, welche besagt, dass sich jede beliebige boolesche Funktion $f(x_1, x_2, \dots, x_n)$ entwickeln lässt zu

$$f(x_1, x_2, \dots, x_n) = \overline{x_1} f(0, x_2, \dots, x_n) + x_1 f(1, x_2, \dots, x_n)$$

Wird dieser Ansatz rekursiv angewendet, bekommt man die BDD-Darstellung einer booleschen Funktion. Das Ergebnis dieses Vorganges für die parity-Funktion ($parity^{(n)}(x_1, \dots, x_n) = x_1 \oplus x_2 \oplus \dots \oplus x_n$) mit 4 Bits ist in Abbildung 2.7 dargestellt. Dabei sind die 0-Nachfolger durch gestrichelte Pfeile dargestellt und die 1-Nachfolger mittels durchgezogenen Pfeilen.

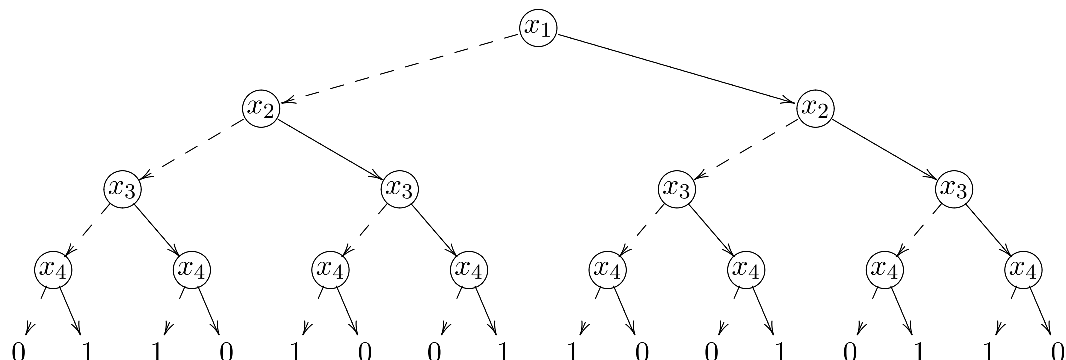


Abbildung 2.7: Beispiel parity-Funktion für 4-Bits

Def. 2.16 (Binäre Entscheidungsbäume) *Ein binärer Entscheidungsbaum (BDD) auf den Variablen $X = \{x_1, \dots, x_n\}$ ist ein Tupel (Q, E, var, q_0) mit:*

1. *V ist eine endliche Knotenmenge.*
2. *$E \subseteq Q \times \mathbb{B} \times Q$ ist eine endliche Kantenmenge zwischen den Knoten, welche mit 0 oder 1 beschriftet sind.*
3. *$q_0 \in Q$.*
4. *(Q, E, q_0) bildet einen gerichteten azyklischen Graphen mit der Wurzel q_0.*
5. *Die Abbildung $var : Q \to X \cup \mathbb{B}$ ist eine Knotenbeschriftung.*

Weiterhin gilt:

1. *Die Knotenmenge Q enthält maximal zwei terminale Knoten, welche mit 0 bzw. 1 beschriftet sind.*
2. *Die restlichen Knoten sind nichtterminal Knoten. Jeder nichtterminal Knoten q ist mit einer Variablen $var(q) = x_i$ beschriftet und besitzt genau zwei ausgehende Kanten,*

die mit 0 (0-Kante, gestrichelt dargestellt) bzw. 1 (1-Kante, durchgezogen dargestellt) beschriftet sind.

3. *Auf jedem Pfad von der Wurzel q_0 kommt jede Variable x_i genau einmal vor.*

Durch die Hinzunahme von zusätzlichen Ordnungsbedingungen sowie Reduktionsmechanismen verbesserte Bryant [Bry86, Bry92] diesen Ansatz zur Darstellung boolescher Funtkionen.

Def. 2.17 (Geordnete binäre Entscheidungsbäume) *Sei π eine totale Ordnung auf den Variablen $X = \{x_1, \ldots, x_n\}$. Ein geordneter Entscheidungsbaum (OBDD) ist ein binärer Entscheidungsbaum (Q, E, var, q_0), bei dem die Knoten entlang eines Pfades bezüglich ihrer Beschriftung gemäß der Ordnung π geordnet sind. Für jede Kante $(q, b, q') \in E$ gilt daher: $var(q) <_\pi var(q')$.*

Abbildung 2.7 ist somit ein OBDD. Ausserdem ist in dieser Abbildung erkennbar, dass OBDDs Redundanzen enthalten.

Def. 2.18 (Reduktionsregeln)
Eliminationsregel: *Wenn die 0- und die 1-Kante eines Knotens q auf den gleichen Knoten q' zeigen, dann eliminiere q und lenke alle in q eingehenden Kanten auf q' um.*

Isomorphieregel: *Identifiziere zwei Knoten, deren Unterdiagramme isomorph sind und eliminiere einen der beiden und lenke alle in diesen Knoten eingehenden Kanten auf den verbleibenden anderen Knoten um.*

Können die Reduktionsregeln nicht mehr weiter angewendet werden, heißt der OBDD auch reduziert, abgekürzt als ROBDD.

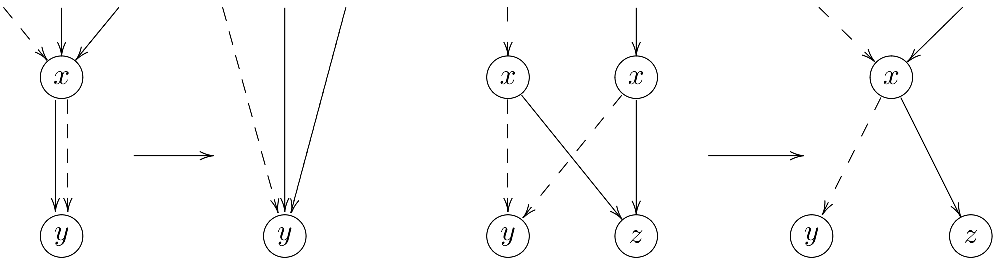


Abbildung 2.8: Reduktionsregeln

Werden die Reduktionsregeln auf die parity-Funktion mit 4-Bits aus Abbildung 2.7 angewendet, fällt auf, dass die Unterbäume zur Variablen x_4 nur in zwei unterschiedlichen Formen auftreten. Somit kann die Isomorphieregel angewendet werden. Das Ergebnis davon ist dann im linken Teil der Abbildung 2.9 dargestellt. Im rechten Teil sind dann noch die isomorphen x_3-Bäume zusammengefasst worden.

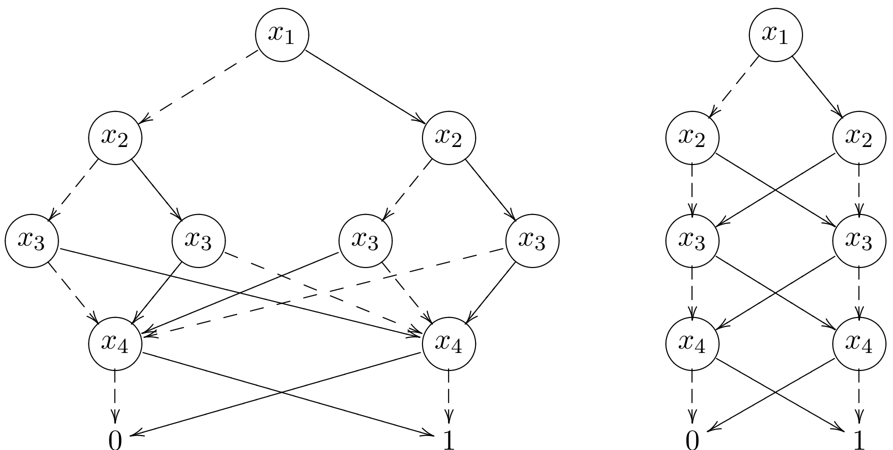


Abbildung 2.9: Reduzierte Darstellung der parity-Funktion mit 4-Bits

Def. 2.19 (Reduzierter geordneter binärer Entscheidungsbaum)
ROBBDs über den Variablen $x_1, x_2, \ldots, x_n$ sind induktiv aufgebaut gemäß:

- *0 und 1 sind ROBBDs*
- *mit s_l und s_r ist jeweils auch $x_i(s_l, s_r)$ ein ROBDD, sofern $s_l \neq s_r$ und für alle Variablen x_j in s_l, s_r gilt $j > i$.*

Def. 2.20 *Die durch den ROBDD s über den Variablen $x_1, x_2, \ldots, x_n$ definierte boolesche Funktion $f_s : \mathbb{B}^n \to \mathbb{B}$ ist induktiv definiert durch:*

- *$s = 0$: f_s = Konstante 0*
  $s = 1$: f_s = Konstante 1
- *$s = x_i(s_l, s_r)$: $f_s(b_1, \ldots, b_n) = \begin{cases} f_{s_l}(b_1, \ldots, b_n) & \text{falls } b_i = 0 \\ f_{s_r}(b_1, \ldots, b_n) & \text{falls } b_i = 1 \end{cases}$*

Satz 2.11 (Eindeutigkeit von ROBBDs [Bry86, Bry92])
Wenn für zwei ROBBDs über den Variablen $x_1, x_2, \ldots, x_n$ gilt $f_s = f_t$, dann gilt auch $s = t$.

Beweis: Siehe [MT98].

Die Definition der OBDDs fordert eine feste Variablenordnung. Allerdings hat diese einen entscheidenden Einfluß auf die Größe der ROBDD-Darstellung und somit auch auf den Speicher- und Rechenzeitbedarf.

Abbildung 2.10 zeigt am Beispiel der Funtkion $f = (x_1x_2 + \overline{x_1x_2})(x_3x_4 + \overline{x_3x_4})$, wie sich die Variablenordnung auf die Komplexität des resultierenden ROBBDs auswirkt. Eine "gute" Variablenordnung kann zu einer kompakten ROBDD-Darstellung führen und infolgedessen zu geringeren Rechenzeiten bei Operationen auf diesem BDD, während eine ungünstige Ordnung im schlimmsten Fall dazu führt, dass der vorhandene Arbeitsspeicher zur Speicherung des BDDs nicht ausreicht und somit die Berechnung abbricht.

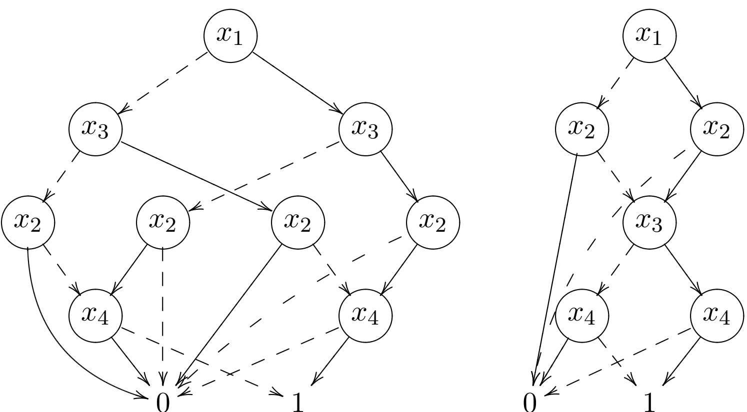


Abbildung 2.10: Unterschiedliche Variablenordnungen für die Funktion $f = (x_1x_2+\overline{x_1x_2})(x_3x_4+\overline{x_3x_4})$

Empirische Untersuchungen haben die zwei folgenden Grundregeln für eine günstige Variablenordnung hervorgebracht:

1. Variablen, welche stark voneinander abhängen, sollten auch in der Variablenordnung nahe beieinander stehen.
2. Variablen, die die boolesche Funtkion stark beeinflussen, sollten nahe an den terminalen Knoten liegen, da dort die Knotenanzahl potentiell höher ist.

Weitere Informationen zur Variablenordnung können beispielsweise in [SBB99] nachgelesen werden.

Kapitel 3

Reaktive Programme über einen symbolischen Zustandsraum

Dieses Kapitel präsentiert die Ergebnisse des theoretischen Teils dieser Diplomarbeit. Nachdem im letzten Kapitel die reaktiven Programme über einen abstrakten Zustandsraum betrachtet worden sind, liegt jetzt ein symbolischer Zustandsraum zugrunde. Somit ist mit den hier vorgestellten Ergebnissen eine praktische Anwendung möglich. Beim abstrakten Zustandsraum ist das Problem, dass die Anzahl der Zustände, welche zur Modellierung einer realen Systems benötigt werden, oftmals größer ist, als dass ein Computer damit umgehen könnte. Dieses Problem wird auch "State Explosion Problem" genannt. [Val98] beschäftigt sich mit diesem Problem allgemein und stellt mehrere Verfahren vor, um die Zustandsanzahl zu reduzieren. In den meisten Fällen können diese Methoden nur bestimmte Analyse bzw. Verifikationsfragen beantworten, ohne dass sie die Möglichkeit verlieren, die Zustandsanzahl zu reduzieren.

Der Ansatz des symbolischen Zustandsraumes stellt eine Möglichkeit dar, dieses Problem zu vermeiden. Der Grundgedanke ist vergleichbar mit dem symbolischen Model Checking im Gegensatz zu dem Normalen, wie in [BCM$^+$90] beschrieben.

3.1 Terminologie

Um den abstrakten und symbolischen Zustandsraum deutlich voneinander abzugrenzen und somit Verwechselungen zu vermeiden, werden im Symbolischen alle booleschen Formeln durch kleine griechische Buchstaben dargestellt. Die Tabelle 3.1 liefert eine Gegenüberstellung der abstrakten und symbolischen Schreibweisen, sowie eine kurze Erläuterung der Symbole.

3.2 Spielgraph

Im Symbolischen ist der Zustandsraum gegeben durch eine Anzahl boolescher Aussagevariablen. Zustände sind dann konkrete Belegungen dieser Variablen. Bei zwei Variablen sind dann vier Zustände möglich - $\{\binom{0}{0}, \binom{0}{1}, \binom{1}{0}, \binom{1}{1}\}$. Die Transitionen im Spielgraphen werden durch boolesche Formeln über den Variablen definiert.

Abstrakt	Symbolisch	Bedeutung
Q	V	Zustände im Abstrakten, Variablenmenge im Symbolischen
Q_i	φ_i	Zustände Spieler i
E	τ	Transitionen des Spielgraphens
W_i	ω_i	Gewinnbereich Spieler i
F	λ	Zustandsmenge für Gewinnbedingung (Büchi, Garantie)
C_i	γ_i	Zustände der Farbe i
P_i	ϕ_i	RequestMenge i (Request-Response-Spiel)
R_i	ψ_i	ResponseMenge i (Request-Response-Spiel)
	σ	Gewinnstrategie

Tabelle 3.1: Terminologie für den symbolischen Zustandsraum

Def. 3.1 (Spielgraph) *Ein Spielgraph ist definiert durch das Quadrupel $G = (V, \varphi_0, \varphi_1, \tau)$ mit:*

- $V = \{v_0, \ldots, v_n\}$ *Aussagevariablen (keine konkreten Zustände wie im abstrakten Fall)*
- φ_0*: boolesche Formel, die die Zustände von Spieler 0 beschreibt*
- φ_1*: analog für Spieler 1*
- τ*: boolesche Formel, die die Transitionen im Spielgraph beschreibt. Diese Formel ist über den Variablen V und $V' = \{v'_0, \ldots, v'_n\}$ definiert. Die Variablen V dienen zu der Beschreibung der Knoten, von denen die Kanten ausgehen, und V' zu der Beschreibung der Knoten, zu denen die Kanten hingehen. Dies entspricht im abstrakten Fall $E \subseteq Q \times Q$. Auch hier wird verlangt, dass jeder Zustand eine ausgehende Kante hat.*

Aus $Q = Q_0 \dot\cup Q_1$ (Definition 2.1) folgt, dass $\varphi_0 \wedge \varphi_1 = false$ gilt. Aus der Definition 3.1 folgt nicht, dass $\varphi_0 \vee \varphi_1 = true$ gelten muss. Denn es müssen prinzipiell nicht sämtliche Zustände, die mit den Variablen aus V möglich sind, für den Spielgraphen genutzt werden - sonst würde es nur Spielgraphen mit 2^i Zuständen ($i \in \mathbb{N}$) geben.

Beispiel 3.1 *Die Abbildung 3.1 zeigt eine grafische Visualisierung des unten angegebenen symbolischen Spielgraphens. Dieser entspricht dem in Abbildung 2.1 dargestellten abstraktem Beispiel. In der Abbildung sind die Zustände umbenannt worden, dabei entspricht $\binom{0}{0}$ dem Fall, dass v_0 und v_1 $false$ sind. Außerdem sind die Transitionen in der Art beschriftet worden, dass hervorgeht, aus welchem Teil der Formel τ die Transition entstanden ist.*

- $V = \{v_0, v_1\}$
- $\varphi_0 = \neg v_0$
- $\varphi_1 = v_0$
- $\tau = \underbrace{(\neg v_0 \wedge \neg v_1 \wedge v'_0)}_{i} \vee \underbrace{(\neg v_0 \wedge v_1 \wedge \neg v'_1)}_{ii} \vee \underbrace{(v_0 \wedge \neg v_1 \wedge v'_1)}_{iii} \vee \underbrace{(v_0 \wedge v_1 \wedge (v'_0 \Leftrightarrow \neg v'_1))}_{iv}$

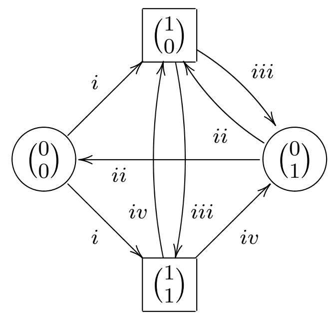


Abbildung 3.1: Visualisierung eines symbolischen Spielgraphens

Die Transitionsformel τ im Beispiel 3.1 setzt sich aus vier gleichwertigen Teilen zusammen, deshalb wird an dieser Stelle nur kurz der erste Teil erläutert. Aus $\neg v_0 \wedge \neg v_1$ folgt, dass Kanten, die vom Zustand $\binom{0}{0}$ ausgehen, beschrieben werden, und aus v_0' folgt, dass die Kanten zu den Zuständen gehen, bei denen die erste Komponente *true* ist. Sie gehen also zu den Zuständen $\binom{1}{0}$ und $\binom{1}{1}$, wie auch die Abbildung 3.1 zeigt.

Analog zum abstrakten Zustandsraum lassen sich auch für den symbolischen Zustandsraum die folgenden Formeln über den Variablen von V definieren:

- $Oc(\rho)$: die Menge der in der Partie ρ vorkommenden Zustände (occurence set)
- $In(\rho)$: die Menge der in der Partie ρ unendlich vorkommenden Zustände (infinity set)
- ω_0: Gewinnbereich von Spieler 0
- ω_1: Gewinnbereich von Spieler 1

3.3 Garantie- und Sicherheitsspiele

Für die Lösung der Garantie- bzw. Sicherheitsspiele wird die Attraktor-Konstruktion aus dem Abstrakten ins Symbolische übertragen.

Def. 3.2 (Garantiespiel) *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph und λ eine Formel über den Variablen von V. Ein Garantiespiel hat die Gewinnbedingung:*

$$\rho \in Win_0 :\Leftrightarrow Oc(\rho) \wedge \lambda \neq false$$

Satz 3.1 *Für Garantiespiele sind die Gewinnbereich ω_0 und ω_1 von Spieler 0 bzw. Spieler 1 in Polynomzeit berechenbar, ebenso wie entsprechende positionale Gewinnstrategien.*

Beweis: Die Attraktorberechnung wird ins Symbolische transformiert:

$$\begin{aligned}
Attr_0^0(\lambda) &:= \lambda \\
Attr_0^{i+1}(\lambda) &:= Attr_0^i(\lambda) \vee \\
&\quad (\varphi_0 \wedge (\tau \wedge Attr_0^i(\lambda)|_{V \to V'})|_V) \vee \\
&\quad (\varphi_1 \wedge \neg(\tau \wedge \neg Attr_0^i(\lambda)|_{V \to V'})|_V)
\end{aligned}$$

Behauptung: $Attr_0(\lambda) = \bigvee_i Attr_0^i(\lambda)$

Die Definition von $Attr_0^{i+1}$ setzt sich aus drei Teilen zusammen. Der erste Teil sorgt dafür, dass die Zustände, die bereits in $Attr_0^i$ enthalten sind, auch in $Attr_0^{i+1}$ kommen. Der zweite Teil nimmt die Zustände von Spieler 0 auf, die mindestens eine Transition nach $Attr_0^i$ haben. Der dritte Teil ist analog für Spieler 1 mit dem Unterschied, dass alle Transitionen nach $Attr_0^i$ gehen müssen, damit der Zustand aufgenommen wird. Betrachtet man den zweiten Teil genauer,

$$\overbrace{\varphi_0 \wedge \underbrace{(\tau \wedge \overbrace{\underbrace{Attr_0^i(\lambda)|_{V\to V'}}_{i})}^{ii}|_V}_{iii}}^{iv}$$

bewirken die einzelnen Formelelemente:

(i) Die Variablen in $Attr_0^i$ werden umbenannt nach V', d.h. $v_0 \to v_0'$, $v_1 \to v_1'$, ...

(ii) Beschreibt alle Transitionen im gegebenen Spielgraphen, die nach $Attr_0^i$ gehen.

(iii) Schränkt diese auf die Variablen aus V ein, damit bleiben die Zustände übrig, die eine Kante nach $Attr_0^i$ haben. Dies entspricht einer existenziellen Quantifizierung der Variablen aus V'.

(iv) Schränkt die Menge auf Knoten vom Spieler 0 ein.

Für den dritten Teil der Formel ist die Äquivalenz benutzt worden, daß $\forall (q, r) \in E$ gilt $r \in Attr_0^i(F)$ äquivalent ist zu $\nexists (q, r) \in E$ mit $r \notin Attr_0^i(F)$ ist.

Dann kann die Attraktor-Strategie analog zum Abstrakten definiert werden:

$$\begin{aligned} Strat_0^0(\lambda) &= false \\ Strat_0^{i+1}(\lambda) &= Strat_0^i(\lambda) \vee \\ &\quad (Attr_0^{i+1}(\lambda) \wedge \neg Attr_0^i(\lambda) \wedge \tau \wedge (\varphi_1 \vee (\varphi_0 \wedge Attr_0^i(\lambda)|_{V\to V'}))) \end{aligned}$$

Bei $Strat_0^{i+1}(\lambda)$ werden für die Zustände, die noch nicht in $Attr_0^i(\lambda)$ aber in $Attr_0^{i+1}(\lambda)$ sind, Strategiekanten hinzugefügt. Für Zustände von Spieler 1 können sämtliche Kanten zu Nachfolgezuständen hinzugenommen werden und für Spieler 0-Zustände nur die Kanten, so dass der Nachfolgezustand in $Attr_0^i(\lambda)$ landet.

Die beiden Routinen $Strat_0(F)$ und $Attr_0(F)$ lassen sich auch zu einer einzigen zusammenfassen. Das Ergebnis davon ist im Algorithmus 3.1 dargestellt.

Dann gilt für die Gewinnbereiche dasselbe wie im Abstrakten:

$$\begin{aligned} \omega_0 &= Attr_0(\lambda) \\ \omega_1 &= (\varphi_0 \vee \varphi_1) \wedge \neg\omega_0 \end{aligned}$$

Korrektheitsbeweis Attraktor:
Dazu wird mittels Induktion über i gezeigt, dass $Attr_{abs}^i(F)$=$Attr_0^i(F)$ im Abstrakten genau die selben Zustände beinhaltet, die $Attr_{sym}^i(\lambda)$=$Attr_0^i(\lambda)$ im Symbolischen beschreibt:

- $i = 0$: $Attr^0_{abs}(F) = F$, $Attr^0_{sym}(\lambda) = \lambda$ $\surd$
- Angenommen es gilt, dass $Attr^n_{abs}(F)$ und $Attr^n_{sym}(\lambda)$ dieselben Zustände beschreiben. Dann muss noch gezeigt werden, dass dies auch für $i = n+1$ gilt:
  - $Attr^{n+1}_{abs}(F) \subseteq Attr^{n+1}_{sym}(\lambda)$:

    1. Fall: Sei $q \in Attr^{n+1}_{abs}(F)$, $q \notin Attr^n_{abs}(F)$ und $q \in Q_0$:
    $\Rightarrow \exists (q, r) \in E$ mit $r \in Attr^n_{abs}(F)$
    $\Rightarrow$ die symbolische Darstellung von q ist erfüllende Belegung von $\varphi_0 \wedge (\tau \wedge Attr^n_{sym}(\lambda)|_{V \to V'})|_V$, da $Attr^n_{sym}(\lambda) = Attr^n_{abs}(F)$
    $\Rightarrow$ die symbolische Darstellung von q ist erfüllende Belegung von $Attr^{n+1}_{sym}(\lambda)$

    2. Fall: Sei $q \in Attr^{n+1}_{abs}(F)$, $q \notin Attr^n_{abs}(F)$ und $q \in Q_1$:
    $\Rightarrow \forall (q, r) \in E$ mit $r \in Attr^n_{abs}(F)$
    $\Rightarrow$ die symbolische Darstellung von q ist erfüllende Belegung von $\varphi_1 \wedge \neg(\tau \wedge \neg Attr^n_{sym}(\lambda)|_{V \to V'})|_V$, da $Attr^n_{sym}(\lambda) = Attr^n_{abs}(F)$
    $\Rightarrow$ die symbolische Darstellung von q ist erfüllende Belegung von $Attr^{n+1}_{sym}(\lambda)$

    3. Fall: Sei $q \in Attr^{n+1}_{abs}(F)$ und $q \in Attr^n_{abs}(F)$, da $Attr^n_{sym}(\lambda) = Attr^n_{abs}(F)$ gilt, ist die symbolische Darstellung von q erfüllende Belegung von $Attr^{n+1}_{smy}(\lambda)$

  - $Attr^{n+1}_{sym}(\lambda) \subseteq Attr^{n+1}_{abs}(F)$, die abstrakte Darstellung von q sei $abs(q)$:

    1. Fall: Sei q erfüllende Belegung von $Attr^{n+1}_{sym}(\lambda)$ und φ_0, aber nicht von $Attr^n_{sym}(\lambda)$:
    $\Rightarrow q$ erfüllende Belegung von $\varphi_0 \wedge (\tau \wedge Attr^n_{sym}(\lambda)|_{V \to V'})|_V$
    $\Rightarrow \exists (abs(q), r) \in E$ mit $r \in Attr^n_{abs}(F)$, da $Attr^n_{abs}(F) = Attr^n_{sym}(\lambda)$
    $\Rightarrow abs(q) \in Attr^{n+1}_{abs}(F)$

    2. Fall: Sei q erfüllende Belegung von $Attr^{n+1}_{sym}(\lambda)$ und φ_1, aber nicht von $Attr^n_{sym}(\lambda)$:
    $\Rightarrow q$ erfüllende Belegung von $\varphi_1 \wedge \neg(\tau \wedge \neg Attr^n_0(\lambda)|_{V \to V'})|_V$
    $\Rightarrow \forall (abs(q), r) \in E$ mit $r \in Attr^n_{abs}(F)$, da $Attr^n_{abs}(F) = Attr^n_{sym}(\lambda)$
    $\Rightarrow abs(q) \in Attr^{n+1}_{abs}(F)$

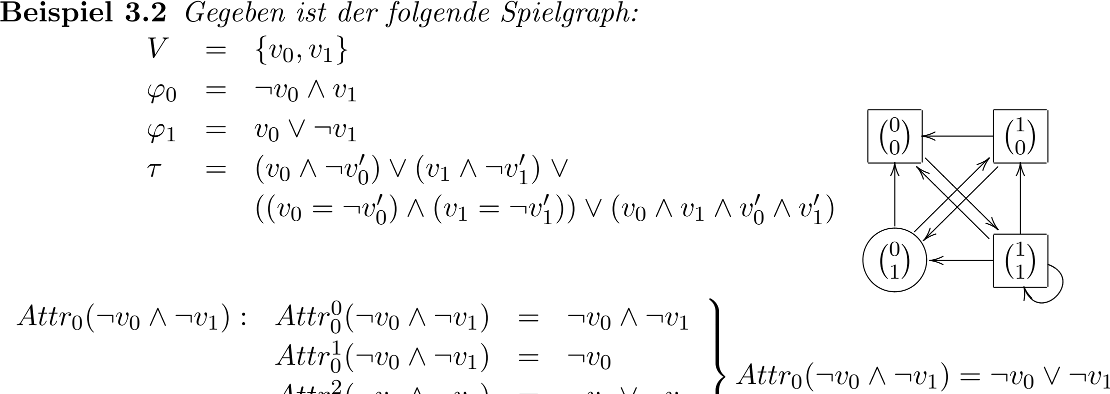
    3. Fall: Sei q erfüllende Belegung von $Attr^{n+1}_{sym}(\lambda)$ und $Attr^n_{sym}(\lambda)$: da $Attr^n_{abs}(F) = Attr^n_{sym}(\lambda)$ gilt $abs(q) \in Attr^n_{abs}(F)$ und somit $abs(q) \in Attr^{n+1}_{abs}(F)$ $\square$

Eine Laufzeitabschätzung für $Attr_0(\lambda)$ ergibt:
$|2^V|$ Schleifendurchläufe (dies entspricht im Abstrakten $|Q|$) und pro Durchlauf

- 2 BDD-Operationen ($\vee$) über V
- 2 BDD-Operationen ($\wedge$) über V
- 2 BDD-Operationen ($\wedge$) über $V \cup V'$ mit anschließender existenzieller Quantifizierung der Variablen von V'
- 2 BDD-Operationen ($|_{V \to V'}$) über V
- 2 BDD-Operationen ($\neg$) über V

Beispiel 3.2 *Gegeben ist der folgende Spielgraph:*

$$\begin{aligned} V &= \{v_0, v_1\} \\ \varphi_0 &= \neg v_0 \wedge v_1 \\ \varphi_1 &= v_0 \vee \neg v_1 \\ \tau &= (v_0 \wedge \neg v_0') \vee (v_1 \wedge \neg v_1') \vee \\ &\quad ((v_0 = \neg v_0') \wedge (v_1 = \neg v_1')) \vee (v_0 \wedge v_1 \wedge v_0' \wedge v_1') \end{aligned}$$



$$Attr_0(\neg v_0 \wedge \neg v_1): \left.\begin{aligned} Attr_0^0(\neg v_0 \wedge \neg v_1) &= \neg v_0 \wedge \neg v_1 \\ Attr_0^1(\neg v_0 \wedge \neg v_1) &= \neg v_0 \\ Attr_0^2(\neg v_0 \wedge \neg v_1) &= \neg v_0 \vee \neg v_1 \\ Attr_0^3(\neg v_0 \wedge \neg v_1) &= \neg v_0 \vee \neg v_1 \end{aligned}\right\} Attr_0(\neg v_0 \wedge \neg v_1) = \neg v_0 \vee \neg v_1$$

Paramater: φ_i Formel der Spielers i Knoten, τ Transitionsformel, λ Knotenformel, zu der der Attraktor berechnet werden soll

---

attraktor(φ_0, φ_1, τ, λ):
$\omega := \lambda$; $\sigma :=$ **false**; 5
do
  $\omega' := \omega$
  $\omega := \omega' \vee (\varphi_0 \wedge (\tau \wedge \omega'|_{V \to V'})|_V) \vee (\varphi_1 \wedge \neg(\tau \wedge \neg\omega'|_{V \to V'})|_V)$
  $\sigma := \sigma \vee (\omega \wedge \neg\omega' \wedge \tau \wedge (\varphi_1 \vee (\varphi_0 \wedge \omega'|_{V \to V'})))$
while ω != ω' 10
return (ω, σ)

Ergebnis: Attraktormenge (ω) und −strategie (σ)

Algorithmus 3.1: Attraktor

3.4 Schwache Paritätsspiele

Bei der Definition der schwachen Paritätsspiele im Symbolischen gibt es einen Unterschied zu der Definition im Abstrakten. Anstatt einer Funktion, welche zu einem gegebenen Knoten dessen Farbe berechnet, werden Farbformeln für die einzelnen Farben angegeben, welchen den C_i-Mengen aus dem Abstrakten entsprechen.

Def. 3.3 (Schwaches Paritätsspiel) *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph und $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_k\}$ eine Menge von Formeln. Sei dabei γ_i eine Formel über den Variablen von V, die die Zustände der Farbe i beschreibt. Die schwache Paritätsbedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigvee_{i=0}^{k} ((Oc(\rho) \wedge \gamma_i \neq false) \wedge (\bigwedge_{j=i+1}^{k} Oc(\rho) \wedge \gamma_j = false)) \text{ mit } i \text{ gerade}$$

Satz 3.2 *Für schwache Paritätsspiele können die Gewinnbereiche ω_0, ω_1 berechnet und entsprechende positionale Gewinnstrategien angegeben werden.*

Beweis: Sei $G = (V, \varphi_0, \varphi_1, \tau)$, und γ_i eine Formel über V, die die Knoten der Farbe i beschreibt, für $i = 0, \ldots, k$ gegeben. Dann sind analog zum Abstrakten die folgenden Formeln $\alpha_k, \alpha_{k-1}, \ldots, \alpha_0$ über den Variablen von V bestimmbar, siehe auch Kapitel 2.1.2:

$$\alpha_i \quad := \quad \begin{cases} - \text{ Spieler 0 kann Besuch der Farbe i erzwingen,} & \\ \quad \text{keine höhere ungerade Farbe wird gesehen} & \text{falls } i \text{ gerade} \\ - \text{ Spieler 1 kann Besuch der Farbe i erzwingen,} & \\ \quad \text{keine höhere gerade Farbe wird gesehen} & \text{falls } i \text{ ungerade} \end{cases}$$

$$\begin{aligned} \alpha_k \quad &:= \quad Attr^G_{k \bmod 2}(\gamma_k) \\ \alpha_i \quad &:= \quad \begin{cases} Attr^{G'}_0(\gamma_i \wedge \neg(\alpha_{i+1} \vee \ldots \vee \alpha_k) & i \text{ gerade} \\ Attr^{G'}_1(\gamma_i \wedge \neg(\alpha_{i+1} \vee \ldots \vee \alpha_k) & i \text{ ungerade} \end{cases} \\ & \quad\quad \text{mit } G' = (V, \varphi'_0, \varphi'_1, \tau') \text{ und} \\ & \quad\quad \varphi'_0 = \varphi_0 \wedge \neg(\alpha_{i+1} \vee \ldots \vee \alpha_k),\ \varphi'_1 = \varphi_1 \wedge \neg(\alpha_{i+1} \vee \ldots \vee \alpha_k), \\ & \quad\quad \tau' = \tau \wedge \neg(\alpha_{i+1} \vee \ldots \vee \alpha_k) \wedge \neg(\alpha_{i+1} \vee \ldots \vee \alpha_k)|_{V \to V'} \end{aligned}$$

Da nur Attraktorberechnungen benutzt werden und die Berechnungen identisch zum abstrakten Fall sind, folgt:

$$\omega_0 = \bigvee_{i \text{ gerade}} \alpha_i \ , \ \omega_1 = \bigvee_{i \text{ ungerade}} \alpha_i$$

Analog zum Abstrakten liefern die Vereinigungen der Attraktor-Stratgien dann die positionalen Gewinnstrategien für Spieler 0 und 1. Zusammengefasst ist alles in Algorithmus 3.2.

```
Paramater: φi Formel der Spielers i Knoten, τ Transitionsformel,
λ Farbformeln für die Farben 0 bis k

weakparity(φ0, φ1, τ, Γ):
ω0:=false; ω1:=false; σ:=false;                                   5
for(i:=0; i<=k; i++)
    if (i%2=0) then
        (ω',σ'):=attraktor(φ0\(ω0 ∨ ω1), φ1\(ω0 ∨ ω1), τ, γi)
        ω0 = ω0 ∨ ω'
    else                                                         10
        (ω',σ'):=attraktor(φ1\(ω0 ∨ ω1), φ0\(ω0 ∨ ω1), τ, γi)
        ω1 = ω1 ∨ ω'
    σ := σ ∨ σ'
return (ω0, ω1, σ)
                                                                 15
Ergebnis: Gewinnbereiche (ωi) und -strategie (σ)
```

Algorithmus 3.2: Schwache Paritätsspiel

3.5 Staiger-Wagner-Spiele

Die Lösung der Staiger-Wagner lässt sich leider nicht so elegant wie bei den anderen Spielen vom abstrakten auf den symbolischen Zustandsraum transformieren. Die Lösung benutzt eine Tiefensuche über den Spielgraphen, um den für die Reduktion auf die schwachen Paritätsspiele nötigen erweiterten Spielgraphen zu erzeugen.

Def. 3.4 (Staiger-Wagner Spiel - [SW74]) *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph und $\Lambda = \{\lambda_1, \ldots, \lambda_k\}$ eine Menge von Formeln über den Variablen von V. Die Staiger-Wagner Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigvee_{i=1}^{k} Oc(\rho) \Leftrightarrow \lambda_i$$

Satz 3.3 *Staiger-Wagner Spiele lassen sich auf schwache Paritätsspiele reduzieren und somit lassen sich die Gewinnbereiche und entsprechende Automatenstrategien berechnen.*

Beweis: Zu gegebenen Spielgraphen $G = (V, \varphi_0, \varphi_1, \tau)$ (mit n Variablen für V) konstruiere $G' = (V', \varphi'_0, \varphi'_1, \tau')$ wie folgt:

- V' mit $(n + 2^n)$-Aussagevariablen:
  - $v_0 \ldots v_{n-1}$ für den alten Zustand
  - $v_n \ldots v_{2^n+(n-1)}$ für $Pot(Q)$
- Der Spielgraph G' wird mittels einer Tiefensuche aufgebaut, siehe Algorithmus 3.5.

Die Variablen zur Speicherung von $Pot(Q)$ werden in der Art genutzt, dass v_{n+i} für $i = 0$ bis $2^n - 1$ genau dann *true* ist, wenn Zustand q mit valueOf(q)= i bereits besucht wurde. Dabei ist valueOf durch den folgenden Algorithmus beschrieben:

```
Paramater: λ Zustand, zu dem der Wert in kanonischer Reihenfolge bestimmt
werden soll
------------------------------------------------------------------------
valueof(λ):
value:=0                                                               5
for (i:=0; i<n; i++)
   if (v_i=true) then value:=value+2^i
return value
------------------------------------------------------------------------
Ergebnis: Position des Zustandes in kanonischer Reihenfolge           10
```

Algorithmus 3.3: Funktion valueOf, die den Wert eines BDDs liefert

Die Definitionen der Farbformeln γ_0 bis $\gamma_{2 \cdot n}$, der Formeln der Spieler 0 bzw. 1 Knoten, sowie der Transitionsformel, werden über eine Tiefensuche über den gegebenen Spielgraphen

G gemacht. Dabei wird von jedem Zustand $v_0 \dots v_{n-1}$ des ursprünglichen Spielgraphens die Tiefensuche gestartet. Für jeden dabei gefundenen Zustand wird dann die Farbe bestimmt, der Knoten als bearbeitet markiert und dieser der entsprechenden Farbformel hinzugefügt. Trifft die Tiefensuche auf einen bereits bearbeiteten Knoten, wird die weitere Suche an dieser Stelle abgebrochen. Für eine Pseudo-Code-Angabe dieses Ansatzes siehe Algorithmus 3.5.

Paramater: φ_i Formel der Spielers i Knoten, τ Transitionsformel, Λ Menge von Knotenformeln, in denen sich das Spiel aufhalten muss

```
staiger-wagner(φ0, φ1, τ, Λ):
φ'0:=false; φ'1:=false; τ':=false;                    5
for (int i:=0; i<=2n; i++)
   γi:=false
depth_first_search(τ,φ0 ∨ φ1, null)
return weakparity(φ'0,φ'1,τ',Γ)
                                                      10
```

Ergebnis: Gewinnbereiche (ω_i) und -strategie (σ), welche durch die Reduktion auf das schwache Paritätsspiel gewonnen wurden

Algorithmus 3.4: Staiger-Wagner Spiel

Erläuterungen zu der Tiefensuche, Algorithmus 3.5, welche sowohl die Farbformeln γ_i als auch den erweiterten Spielgraphen erzeugt:

- Zeilen 6 bis 8 behandeln den Fall, dass ein Knoten $f(q)$ mit q Knoten des ursprünglichen Spielgraphens betrachtet wird. Dazu wird genau die Variable v_{n+i} mit $i = valueof(q)$ true gesetzt und alle anderen auf false.

- Zeilen 10 und 11 betrachten Knoten, die nicht direkt aus dem gegebenen Spielgraphen transformiert wurden. Dazu wird die Menge der besuchten Zustände in den Variablen v_n bis $v_{2^n+(n-1)}$ gleichgesetzt der Menge der bereits besuchten Zustände, welche im Vorgängerknoten ϕ übergeben wurde. Ausserdem wird der aktuelle Zustand hinzugefügt. Weiterhin wird eine neue Transition zu τ' hinzugefügt, wobei der Ausgangspunkt der alte Zustand ist und der Zielpunkt der neu berechnete.

- Zeile 12 fragt ab, ob der Knoten bereits bearbeitet wurde. Falls dies nicht der Fall ist, wird er als bearbeitet markiert, eine Formel erzeugt (Zeile 14), die alle bereits besuchten Zustände beinhaltet, und anhand dieser der Zustand der passenden Farbformel hinzugefügt und die Tiefensuche auf den Nachfolgern des Knoten fortgesetzt.

Versuche, mit einer geringeren Platzkomplexität zur Speicherung von $Pot(Q)$ auszukommen, haben gezeigt, dass 2^n-zusätzliche Variablen benötigt werden. Zwar reichen zur Speicherung einer Menge von Zuständen über n-Variablen weitere n-Variablen, allerdings ist dann ein Zustand des reduzierten Spieles keine Belegung aller Variablen mehr, wie folgendes kleines Beispiel zeigt. Betrachtet wird ein Spielgraph mit einer Variablen, der die folgende Gestalt hat:

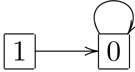
Paramater: τ Transitionsformel, λ Knoten, für die die Nachfolger bestimmt werden sollen, ϕ Vorgängerknoten

---

depth_first_search(τ, λ, ϕ):
for ω:=fulfilling_assignment(λ) 5
  if (ϕ=null) **then** // *Startknoten des neuen Graphens*
    for (int i=0; i<2^n; i++)
      $\omega := \omega \wedge (v_{n+i} \Leftrightarrow (i = valueOf(\omega)))$
  else
    $\omega := \omega \wedge (\phi|_{v_n \ldots v_{2^n+(n-1)}} \vee v_{n+valueOf(\omega)})$ 10
    $\tau' := \tau' \vee (\phi \wedge \omega|_{v_0 \ldots v_{2^n+(n-1)} \rightarrow v'_0 \ldots v'_{2^n+(n-1)}})$
  if not ismarked(ω) **then**
    mark(ω)
    ω':=createset($\omega|_{v_n \ldots v_{2^n+(n-1)}}$)
    i:=count_fulfilling_assignments(ω') 15
    if (ω' element **of** Λ) **then**
      $\gamma_{2 \cdot i} := \gamma_{2 \cdot i} \vee \omega$
    else
      $\gamma_{2 \cdot i-1} := \gamma_{2 \cdot i-1} \vee \omega$
    depth_first_search(τ, successors($\omega|_{v_0 \ldots v_{n-1}}$), ω) 20

Ergebnis: Farbformeln (γ_i) und Transitionsformel (τ')

Algorithmus 3.5: Konstruktion des gefärbten Spielgraphens bei der Reduktion des Staiger-Wagner Spiels



Dabei entspricht der Zustand 0 $\neg v_0$ und Zustand 1 v_0. Sei $\Gamma := \{v_0, \neg v_0\}$. Dann würde der resultierende gefärbte Spielgraph bei nur n weiteren Zuständen so aussehen:

$1, \{1\}, 2 \rightarrow 0, \{0, 1\}, 3 \qquad 0, \{0\}, 2$

Dabei ist die erste Komponente der alte Zustand, die zweite Komponente die bereits besuchten Zustände inklusive dem aktuellen und die letzte Komponente die Farbe des Zustandes. Von links beginnend sind die symbolischen Darstellungen $v_0 \wedge v_1$, $\neg v_0$ und $\neg v_0 \wedge v_1$. Das Problem hierbei ist, dass der zweite Zustand nicht beide Variablen belegt. Somit liefert der Algorithmus für schwache Paritätsspiele für dieses Spiel ein falsches Ergebnis, denn beide Zustände des ursprünglichen Spielgraphens werden dem Gewinnbereich des Spielers 1 zugeschlagen ↯

3.6 Request-Response-Spiele

Für die Request-Response-Spiele von Patrick Hütten ist es gelungen, die Vorgehensweise zur Lösung der Spiele auf den symbolischen Zustandsraum zu überführen.

Def. 3.5 (Request-Response Spiel - [Hüt03]) *Sei* $G = (V, \varphi_0, \varphi_1, \tau)$ *ein Spielgraph,* ϕ_i *und* ψ_i *Formeln über den Variablen von* V *für* $1 \leq i \leq r$ *mit* $r \in \mathbb{N}$*. Die Request-Response Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigwedge_{i=1}^{r} \bigwedge_{j} (\rho(j) \wedge \phi_i = \rho(j)) \Rightarrow \exists_{j' \geq j}\ \rho(j') \wedge \psi_i = \rho(j')$$

Als LTL-Formel ergibt sich:

$$\rho \in Win_0 :\Leftrightarrow \bigwedge_{i=1}^{r} G(\phi_i \rightarrow F\ \psi_i)$$

Satz 3.4 ([Hüt03]) *Request-Response Spiele lassen sich auf Büchi-Spiele reduzieren und somit lassen sich die Gewinnbereiche* ω_0 *und* ω_1 *sowie entsprechende Automatenstrategien berechnen.*

Beweis: Analog zum Abstrakten wird zu einem gegebenen Spielgraphen $G = (V, \varphi_0, \varphi_1, \tau)$ (mit n Variablen für V) ein neuer Spielgraph $\bar{G} = (\bar{V}, \bar{\varphi_0}, \bar{\varphi_1}, \bar{\tau})$ konstruiert. Die Idee der Konstruktion ist die selbe wie beim abstrakten Zustandsraum.

- $\bar{V}$ mit $(n + 2 \cdot r + 1)$-Aussagevariablen:
  - $v_0 \dots v_{n-1}$ für den alten Zustand
  - $v_n \dots v_{n+r-1}$ zur Codierung von M (dabei ist v_{n+i-1} genau dann true, wenn die i-te Bedingung besteht)
  - $v_{n+r} \dots v_{n+2r-1}$ zur Codierung von m (es ist nur genau eine dieser Variablen true und somit gibt diese Variable an, welche Bedingung als nächstes abgearbeitet werden soll)
  - v_{n+2r} zur Codierung von f (Flag, ob der Zustand ein Endzustand ist)
- Die Transitionsformel $\bar{\tau}$:

$$\begin{aligned}
\bar{\tau} = \underbrace{\tau}_{i} \quad & \wedge \quad \underbrace{(\bigwedge_{i=0}^{r-1} v'_{n+i} \Leftrightarrow ((v_{n+i} \wedge \neg\psi_i|_{\bar{V}\rightarrow\bar{V}'}) \vee (\phi_i|_{\bar{V}\rightarrow\bar{V}'})))}_{ii} \\
& \wedge \quad (\bigvee_{i=0}^{r-1} ((\underbrace{v_{n+r+i} \wedge v'_{n+i} \wedge v'_{n+r+i} \wedge \neg v'_{n+2r})}_{iii} \\
& \vee \quad \underbrace{(v_{n+r+i} \wedge \neg v'_{n+i} \wedge v'_{n+r+(i+1 \bmod r)} \wedge v'_{n+2r})))}_{iv} \\
& \wedge \quad \underbrace{(\bigvee_{i=0}^{r-1} v_{n+r+i} \wedge (\bigwedge_{j=0}^{i-1} \neg v_{n+r+j}) \wedge (\bigwedge_{j=i+1}^{r-1} \neg v_{n+r+j}))}_{\alpha, v} \wedge \underbrace{\alpha|_{\bar{V}\rightarrow\bar{V}'}}_{vi}
\end{aligned}$$

- $\varphi_0' = \varphi_0 \wedge \alpha$ und $\varphi_1' = \varphi_1 \wedge \alpha$
- Endzustandsformel $\lambda = v_{n+2r}$
- Formel β', die einen Zustand β über den Variablen 0 bis n-1 in einen neuen Zustand mit den Variablen 0 bis n+2r transformiert:

$$\beta' = \beta \wedge (\bigwedge_{i=0}^{r-1} v_{n+i} \Leftrightarrow \phi_i) \wedge v_{n+r} \wedge (\bigwedge_{i=n+r+1}^{n+2r} \neg v_i)$$

Dabei bedeuten die einzelnen Formelelemente der Transitionsformel:

(i) Analog der Bedingung (i) bei der Konstruktion im Abstrakten.

(ii) Das Flag (abgelegt in der Variablen mit dem Index $n + i - 1$), dass ein Zustand der Menge ϕ_i besucht wurde, ist beim Zielzustand der Transition gesetzt, wenn es entweder beim Ausgangszustand gesetzt war und der Zielzustand keine erfüllende Belegung von ψ_i ist oder der Zielzustand erfüllende Belegung von ϕ_i ist.

(iii) Wenn die markierte Anforderung im Zielzustand noch nicht erfüllt wurde und bleibt sie als markierte Anforderung bestehen und der Zielzustand ist kein Endzustand.

(iv) Ist die markierte Anforderung im Zielzustand erfüllt worden, wird die Position der Markierung um eins erhöht (beim letzten auf eins zurück) und der Zielzustand gehört zur Endzustandsmenge.

(v) Die Formel α stellt sicher, dass nur eine Variable im Bereich von $n + r$ und $n + 2r - 1$ true ist und alle anderen false.

(vi) Stellt sicher, dass auch im Zielzustand der Transition nur eine Variable im Bereich $n+r$ und $n + 2r - 1$ true ist.

Zur Korrektheit der Transitionsformel:
Dazu wird gezeigt, dass wenn $((q, M, m, f), (q', M', m', f')) \in E$ ist, dann gilt: $sym(q, M, m, f) \wedge sym(q', M', m', f')$ ist erfüllende Belegung von $\bar{\tau}$.

q' Für q' gilt, dass $(q, q') \in E$ sein muss. Dies wird durch den mit (i) markierten Teil der Transitionsformel sichergestellt.

M' Für alle $i \in M'$ gilt, dass entweder i bereits in M enthalten war und der Zielzustand nicht in R_i enthalten war oder der Zielzustand in P_i enthalten ist. Dies ist äquivalent zu dem mit (ii) markierten Teil der Transitionsformel.

m' und f' Dort gibt es die beiden Fälle:

- $m = m'$: Dann ist $f = 0$. Dies wird durch den mit (iii) markierten Teil der Transitionsformel ausgedrückt zusammen mit (v) und (vi).
- $m \neq m'$: Daraus folgt, $m' = m + 1$ für $m < r$ und $m' = 0$ für $m = r$. Ausserdem gilt dann, dass $f' = 1$ ist. Dieses ist äquivalent zu dem mit (iv) marktierten Teil der Transitionsformel in Kombination mit den Teilen (v) und (vi). □

Paramater: φ_i Formel der Spielers i Knoten, τ Transitionsformel, Φ Requestformeln, Ψ Antwortformeln

---

requestresponse(φ_0, φ_1, τ, Φ, Ψ):

$\alpha := \vee_{i=0}^{r-1}(v_{n+r+i} \wedge (\wedge_{j=0}^{i-1} \neg v_{n+r+j}) \wedge (\wedge_{j=i+1}^{r-1} \neg v_{n+r+j}))$ 5

$\bar{\tau} := \tau \wedge \alpha \wedge \alpha|_{\bar{V} \to \bar{V}'} \wedge$
$(\wedge_{i=0}^{r-1} v'_{n+i} \Leftrightarrow ((v_{n+i} \wedge \neg\psi_i|_{\bar{V} \to \bar{V}'}) \vee (\phi_i|_{\bar{V} \to \bar{V}'})))\wedge$
$(v'_{n+2r} \Leftrightarrow (\vee_{i=0}^{r-1} v_{n+r+i} \wedge \neg v'_{n+r+i}))\wedge$
$(\vee_{i=0}^{r-1}((v_{n+r+i} \wedge v'_{n+i} \wedge v'_{n+r+i}) \vee (v_{n+r+i} \wedge \neg v'_{n+i} \wedge v'_{n+r+(i+1 \bmod r)})))$

$\bar{\varphi_0} := \varphi_0 \wedge \alpha$; $\bar{\varphi_1} := \varphi_1 \wedge \alpha$ 10
$\lambda := v_{n+2r}$
$(\omega_0, \omega_1, \sigma) :=$ buechi$(\bar{\varphi_0}, \bar{\varphi_1}, \bar{\tau}, \lambda)$
$\alpha := (\wedge_{i=0}^{r-1} v_{n+i} \Leftrightarrow \phi_i) \wedge v_{n+r} \wedge (\wedge_{i=n+r+1}^{n+2r} \neg v_i)$
return $(\omega_0 \wedge \alpha, \omega_1 \wedge \alpha, \sigma)$
15

Ergebnis: Gewinnbereiche (ω_i) und -strategie (σ)

Algorithmus 3.6: Reuqest-Response-Spiel

3.7 Büchi-Spiele

Da die Lösung der Büchi-Spiele im Abstrakten eine Modifikation der Attraktor-Berechnung benutzt, konnte diese einfach ins Symbolische transformiert werden.

Def. 3.6 (Büchi Spiel - [Büc62]) *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph und λ eine Formel über V. Die Büchi-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow In(\rho) \wedge \lambda \neq false$$

Satz 3.5 *In Büchi-Spielen sind die Gewinnbereiche ω_0, ω_1 berechenbar und Spieler 0 bzw. 1 haben positionale Gewinnstrategien.*

Beweis: Es wird die aus dem Abstrakten bekannte Recur-Konstruktion ins Symbolische transformiert. Im Symbolischen sind $Recur_0(\lambda)$ und $Attr_0^+(\lambda)$ Formeln über den Variablen von V. Die Formel $Recur_0^i(\lambda)$ beschreibt genau die Zustände, von denen Spieler 0 mindestens i Wiederbesuche in den von λ beschriebenen Zuständen garantieren kann. Dazu wird als Hilfsmittel für die Recur-Formel erstmal der Attraktor+ im Symbolischen definiert. Dabei beschreibt $Attr_0^+(\lambda)$ genau die Zustände, von denen Spieler 0 den Besuch von den in λ beschriebenen Zuständen in mehr als einem Schritt erzwingen kann.

$$\begin{aligned}
\alpha_0^0 &= false \\
\alpha_0^1 &= (\varphi_0 \wedge (\tau \wedge \lambda|_{V\to V'})|_V) \vee (\varphi_1 \wedge \neg(\tau \wedge \neg\lambda|_{V\to V'})|_V) \\
i \geq 1: \quad \alpha_0^{i+1} &= \alpha_0^i \vee (\varphi_0 \wedge (\tau \wedge \alpha_0^i|_{V\to V'})|_V) \vee (\varphi_1 \wedge \neg(\tau \wedge \neg\alpha_0^i|_{V\to V'})|_V) \\
Attr_0^+(\lambda) &= \bigvee_{i\geq 0} \alpha_0^i
\end{aligned}$$

Die Berechnung von $Attr_0^+(\lambda)$ verläuft analog zum normalen Attraktor, nur dass $\alpha_0^0 = false$ ist und somit α_0^1 angepasst worden ist. Auch die Berechnung der Strategie muss nur minimal angepasst werden und ergibt sich dann wie folgt:

$$\begin{aligned}
Strat+_0^0(\lambda) &= false \\
Strat+_0^1(\lambda) &= \alpha_0^1 \wedge \tau \wedge (\varphi_1 \vee (\varphi_0 \wedge \lambda|_{V\to V'})) \\
Strat+_0^{i+1}(\lambda) &= Strat+_0^i(\lambda) \vee (\alpha_0^{i+1} \wedge \neg\alpha_0^i \wedge \tau \wedge (\varphi_1 \vee (\varphi_0 \wedge \alpha_0^i(\lambda)|_{V\to V'})))
\end{aligned}$$

Auch die Berechnung von $Recur_0(\lambda)$ verläuft analog zum abstrakten Fall:

$$\begin{aligned}
Recur_0^0(\lambda) &:= \lambda \\
Recur_0^{i+1}(\lambda) &:= \lambda \wedge Attr_0^+(Recur_0^i(\lambda)) \\
Recur_0(\lambda) &:= \bigwedge_{i\geq 0} Recur_0^i(\lambda)
\end{aligned}$$

Unter der Voraussetzung, dass die Attraktor+-Formel im Symbolischem korrekt ist, ist auch die Recur-Berechnung korrekt, da sie exakt mit der Berechnung im Abstrakten übereinstimmt und nur den Attraktor+ benutzt. Somit lassen sich dann die Gewinnbereiche festlegen:

$$\begin{aligned}
\omega_0 &:= Attr_0(Recur_0(\lambda)) \\
\omega_1 &:= (\varphi_0 \vee \varphi_1) \wedge \neg Attr_0(Recur_0(\lambda)) = (\varphi_0 \vee \varphi_1) \wedge \neg\omega_0
\end{aligned}$$

Korrektheitsbeweis Attraktor+: Dazu wird mittels Induktion über i gezeigt, dass $A_0^i(F)$ im Abstrakten genau die selben Zustände beinhaltet, die auch $\alpha_0^i(\lambda)$ im Symbolischen beschreibt:

- $i = 0$: $A_0^0(F) = \emptyset$, $A_0^0(\lambda) = false$ $\surd$
- $i = 1$:
  - $A_0^1(F) \subseteq \alpha_0^1(\lambda)$. Sei $sym(q)$ die symbolische Darstellung von q:

    1. Fall: Sei $q \in A_0^1(F)$ und $q \in Q_0$:
    $\Rightarrow \exists(q,r) \in E$ mit $r \in F$
    $\Rightarrow sym(q)$ ist erfüllende Belegung von $\varphi_0 \wedge (\tau \wedge \lambda|_{V\to V'})|_V$
    $\Rightarrow sym(q)$ ist erfüllende Belegung von $\alpha_0^1(\lambda)$

    2. Fall: Sei $q \in A_0^1(F)$ und $q \in Q_1$:
    $\Rightarrow \forall(q,r) \in E$ mit $r \in F$
    $\Rightarrow sym(q)$ ist erfüllende Belegung von $\varphi_1 \wedge \neg(\tau \wedge \neg\lambda|_{V\to V'})|_V$
    $\Rightarrow sym(q)$ ist erfüllende Belegung von $\alpha_0^1(\lambda)$
  - $\alpha_0^1(\lambda) \subseteq A_0^1(F)$. Sei $abs(q)$ die abstrakte Darstellung von q.

1. Fall: Sei q erfüllende Belegung von $\alpha_0^1(\lambda)$ und φ_0:
$\Rightarrow q$ erfüllende Belegung von $\varphi_0 \wedge (\tau \wedge \lambda|_{V\to V'})|_V$
$\Rightarrow \exists(abs(q), r) \in E$ mit $r \in F$
$\Rightarrow abs(q) \in A_0^1(F)$

2. Fall: Sei q erfüllende Belegung von $\alpha_0^1(\lambda)$ und φ_1:
$\Rightarrow q$ erfüllende Belegung von $\varphi_1 \wedge \neg(\tau \wedge \neg\lambda|_{V\to V'})|_V$
$\Rightarrow \forall(abs(q), r) \in E$ mit $r \in F$
$\Rightarrow abs(q) \in A_0^1(F)$

- Der Induktionsschritt von n auf $n+1$ verläuft identisch wie beim normalen Attraktor.

□

Paramater: φ_i Formel der Spielers i Knoten, τ Transitionsformel, λ Knotenformel, wovon mindestens einer unendlich oft besucht werden soll

```
buechi(φ0, φ1, τ, λ):
(ω0,σ):=attraktor(φ0,φ1,τ,recur(φ0,φ1,τ,λ))                5
ω1:=(φ0 ∨ φ1)\ω0
return (ω0,ω1,σ)
```

Ergebnis: Gewinnbereiche (ω_i) und -strategie (σ)

Algorithmus 3.7: Büchi-Spiel

Paramater: φ_i Formel der Spielers i Knoten, τ Transitionsformel, λ Knotenformel, zu der der Attraktor+ berechnet werden soll

```
attraktor+(φ0, φ1, τ, λ):
ω:=ω' ∨ (φ0 ∧ (τ ∧ λ'|V→V')|V) ∨ (φ1 ∧ ¬(τ ∧ ¬λ|V→V')|V)           5
σ:=ω ∧ τ ∧ (φ1 ∨ (φ0 ∧ λ|V→V'))
do
    ω':=ω
    ω:=ω' ∨ (φ0 ∧ (τ ∧ ω'|V→V')|V) ∨ (φ1 ∧ ¬(τ ∧ ¬ω'|V→V')|V)
    σ:=σ ∨ (ω ∧ ¬ω' ∧ τ ∧ (φ1 ∨ (φ0 ∧ ω'|V→V')))                    10
while ω != ω'
return (ω, σ)
```

Ergebnis: Attraktor+ – Menge (ω) und -strategie (σ)

Algorithmus 3.8: Attraktor+

```
Paramater: φ_i Formel der Spielers i Knoten, τ Transitionsformel,
λ Knotenformel, zu der die Recur–Menge berechnet werden soll
-----------------------------------------------------------------
recur(φ_0, φ_1, τ, λ):
ω:=λ                                                            5
do
    ω':=ω
    ω:=λ ∧ attraktor+(φ_0,φ_1,τ,ω')
while ω != ω'
return ω                                                        10
-----------------------------------------------------------------
Ergebnis: Recur–Menge (ω)
```

Algorithmus 3.9: Recur

3.8 Paritätsspiele

Zur Lösung der Paritätsspiele wird eine von Zielonka ([Zie98]) modifizierte Variante des von dem abstrakten Zustandraumes bekannten McNaughtons-Algorithmus ([McN93]) vorgestellt und auf den symbolischen Zustandsraum übertragen.

Def. 3.7 (Paritätsspiel - [Mos84]) *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph und $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_k\}$ eine Menge von Formeln. Sei dabei γ_i eine Formel über den Variablen von V, die die Zustände der Farbe i beschreibt. Die Paritätsbedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigvee_{i=0}^{k}((In(\rho) \wedge \gamma_i \neq false) \wedge (\bigwedge_{j=i+1}^{k} In(\rho) \wedge \gamma_j = false)) \text{ mit } i \text{ gerade}$$

Satz 3.6 *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph mit Färbung $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_k\}$. Dann kann man für Paritätsspiele die Gewinnbereiche ω_0 und ω_1 und entsprechende positionale Gewinnstrategien für Spieler 0 bzw. Spieler 1 effektiv bestimmen.*

Beweis: Analog zum Abstrakten wird eine Variante des McNaughton-Algorithmus angegeben. Diese ist im Algorithmus 3.10 dargestellt. Der Unterschied zum Abstrakten ist, dass nicht ein einzelner Zustand ausgewählt wird und rekursiv das verbliebene Restspiel betrachtet wird, sondern alle Zustände von der höchsten vorhandenen Farbe.

Paramater: φ_i Formel der Spielers i Knoten, τ Transitionsformel,
Γ Farbformeln für die Farben 0 bis k

```
parity(φ0, φ1, τ, Γ):
if (φ0 ∨ φ1 = false) then                                   5
   return (false, false, false)
else if odd(k) then // k maximal auftretende Farbe
   (ω1, ω0, σ) := parity(φ1, φ0, τ, Γ + 1)
   return (ω0, ω1, σ)
else                                                        10
   (ψ, ρ) := attraktor(φ0, φ1, τ, γ[k] ∧ (φ0 ∨ φ1))
   φ̄ := (φ0 ∨ φ1) ∧ ¬ψ
   (ω0, ω1, σ) := parity(φ0 ∧ φ̄, φ1 ∧ φ̄, τ ∩ (φ̄ × φ̄), Γ)
   if (ω1 = false) then
      ρ' := γ[k] ∧ τ                                        15
      return (ω0 ∨ ψ, ω1, σ ∨ ρ ∨ ρ')
   else
      (ψ', ρ') := attraktor(φ1, φ0, τ, ω1)
      φ̄ := (φ0 ∨ φ1) ∧ ¬ψ
      (ω0', ω1', σ') := parity(φ0 ∧ φ̄, φ1 ∧ φ̄, τ ∩ (φ̄ × φ̄), Γ)    20
      return (ω0', ω1' ∨ ψ', ρ' ∨ σ' ∨ (σ ∧ ω1))
```

Ergebnis: Gewinnbereiche (ω_i) und -strategie (σ)

Algorithmus 3.10: McNaughtons Algorithmus im Symbolischen - Variante von [Zie98]

Korrektheitsbeweis: Der Algorithmus basiert auf [Zie98] und ist eine Verbesserung des klassischen McNaughtons Algorithmus. Die Berechnung erfolgt in der Art, dass für die höchste vorkommende Farbe i der Attraktor für die Knoten der Farbe i berechnet wird und anschliessend das Restspiel rekursiv gelöst wird. Abbildung 3.2 zeigt, die Knoten der Farbe i bezeichnet als C_i, den Attraktor dazu, sowie die Gewinnbereiche W_0 und W_1 auf dem Restspiel.

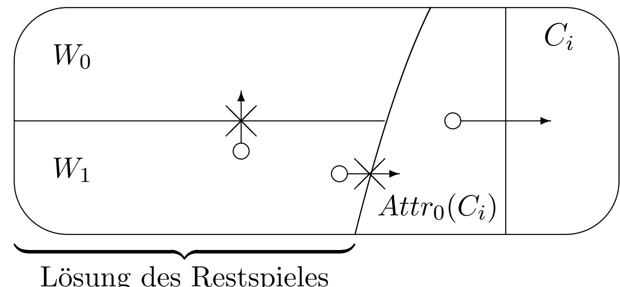


Abbildung 3.2: Charakterisierung McNaughtons Algorithmus im Symbolischen

Bei der Lösung des Restspieles wird zwischen zwei Fällen unterschieden. Für den Fall, dass der Gewinnbereich des Spielers 1 im Restspiel leer ist, wird der Attraktor von C_i komplett dem Gewinnbereich von Spieler 0 zugeschlagen, dies ist in Abbildung 3.3 illustriert.

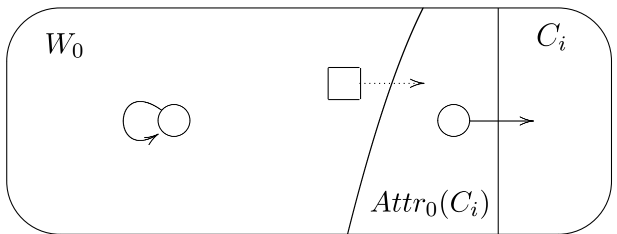


Abbildung 3.3: McNaughtons Algorithmus: Fall $W_1 = \emptyset$

In dem Fall, dass der Gewinnbereich von Spieler 1 im Restspiel nicht leer ist, wird der 1-Attraktor von dem Gewinnbereich des Spielers 1 im Restspiel berechnet und anschliessend muss noch das entstehende Restspiel gelöst werden, wie Abbildung 3.4 zeigt. □

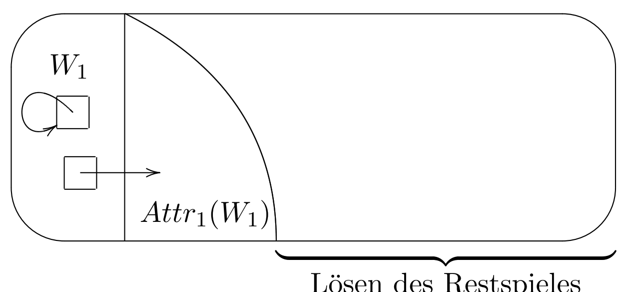


Abbildung 3.4: McNaughtons Algorithmus: Fall $W_1 \neq \emptyset$

2. Variante: Diese Variante unterscheidet sich nur geringfügig von der gerade vorgestellten. Es werden nur die Zeilen 14 und 15 durch die folgenden drei Zeilen ersetzt:

if $((\varphi_0 \wedge \gamma[k] \wedge (\tau \wedge (\psi \vee \omega_0)|_{V \to V'})|_V = \gamma[k] \wedge \varphi_0) \wedge$
$(\varphi_1 \wedge \gamma[k] \wedge \neg(\tau \wedge (\omega_1)|_{V \to V'})|_V = \gamma[k] \wedge \varphi_1))$ **then** 15
$\rho' := (\gamma[k] \wedge \varphi_1 \wedge \tau) \vee ((\gamma[k] \wedge \varphi_0 \wedge \tau) \wedge (\psi \vee \omega_0)|_{V \to V'}))$

Algorithmus 3.11: McNaughtons Algorithmus im Symbolischen - eigene Variante

Einige kurze Erläuterungen zu den ersetzten Zeilen des Agorithmus:

- Zeile 14 stellt sicher, dass alle Zustände von Spieler 0 der Farbe k mindestens eine Transition nach dem Attraktor der Farbe k (als ψ bezeichnet) oder in den Gewinnbereich ω_0 des Restspieles haben
- Zeile 15 analog für die Knoten von Spieler 1 mit der Farbe k mit dem Unterschied, dass alle Transitionen diese Bedingung erfüllen müssen. Dies ist äquivalent dazu, dass es keine Transition zu dem Komplement geben darf (in diesem zum Gewinnbereich von Spieler 1 im Restspiel - kurz ω_1)
- Zeile 16 definiert eine Formel für die Strategiekonstruktion, die in Zeile 17 zu der zurückzugebenen Strategie hinzugefügt wird. Die Formel definiert die Strategie für die Knoten der Farbe k. Da alle Transitionen der Spieler 1-Zustände mit der Farbe k zum Gewinnbereich von Spieler 0 führen, können alle diese Transitionen aufgenommen werden. Von den Spieler 0-Zuständen der Farbe k können nur die Transitionen aufgenommen werden, die wieder in den Gewinnbereich zurückführen.

Vorteil dieser Variante ist, dass weniger Ergebnisse verworfen werden. Dies geschieht nur dann, wenn eine Transition von Spieler 1-Zuständen des Attraktors zum Gewinnbereich des Spielers 1 auf dem Restspiel führt bzw. wenn alle Transition eines Spieler 0-Zustandes des Attraktors diese Bedingung erfüllen.

Beweis: Induktion über $n := |Q|$:

n=1 Der Spielgraph hat die Form ☐ oder ○. Dann klärt die Farbe des Knoten, ob $W_0 = Q$ oder $W_1 = Q$.

n+1 Angenommen, die größte auftretende Farbe ist gerade und hat den Wert k. Ansonsten vertausche Spieler 0 und 1. Die Induktionsvoraussetzung liefert über $Q \backslash Attr_0(C_k)$ eine Zerlegung in Gewinnbereiche U_0, U_1 von Spieler 0 bzw. 1 und zugehörige positionale Gewinnstrategien, da $Q \backslash Attr_0(C_k)$ einen Spielgraphen mit $< n$ Knoten liefert.

1. Fall: Spieler 0 kann für alle $q \in C_k$ aus Transitionen nach $U_0 \cup Attr_0(C_k)$ garantieren.
Behauptung:

$$\left.\begin{array}{ll} (i) & U_0 \cup Attr_0(C_k) \subseteq W_0 \\ (ii) & U_1 \subseteq W_1 \end{array}\right\} \Rightarrow W_0 = U_0 \cup Attr_0(C_k),\ W_1 = U_1$$

Beweis: (ii) ist klar, da von U_1 aus Spieler 1 garantieren kann, dass die Partie in U_1 verbleibt. Die Induktionsvoraussetzung liefert die Behauptung.

Zu (i): Die Partie ρ gemäß der Vorschrift von Knoten $p \in U_0 \cup Attr_0(C_k)$ verbleibt analog in diesem Bereich. Verbleibt ρ schließlich in U_0, so gewinnt Spieler 0 nach Induktionsvoraussetzung. Führt ρ immer wieder durch Zustände $q \in C_k$, so gewinnt Spieler 0, da $q \in C_k$ die maximale Farbe hat, welche gerade ist.

2. Fall: Spieler 1 kann für mindestens ein $q \in C_k$ aus eine Transition nach U_1 erzwingen. Also ist $q \in Attr_1(U_1) \neq \emptyset$. Die Induktionsvoraussetzung liefert die Gewinnbereiche V_0, V_1 und entsprechende positionale Strategien im Spiel $(Q \backslash Attr_1(U_1), E)$, da

$Q \backslash Attr_1(U_1)$ einen Spielgraphen mit $< n$ Knoten liefert.
Behauptung:

$$\left.\begin{array}{ll} (i) & V_0 \subseteq W_0 \\ (ii) & V_1 \cup Attr_1(U_1) \subseteq W_1 \end{array}\right\} \Rightarrow W_0 = V_0,\ W_1 = V_1 \cup Attr_1(U_1)$$

Beweis: (i) ist klar, da Spieler 0 den Verbleib in V_0 nach Induktionsvoraussetzung garantieren kann

Zu (ii): von $p \in Attr_1(U_1)$ kann Spieler 1 nach U_1 ziehen und dort den Verbleib in U_1 garantieren. Von $p_1 \in V_1$ kann Spieler 0 den Übergang nach V_1 oder $Attr_1(U_1)$ wählen - in beiden Fällen wird die Partie durch Spieler 1 gewonnen.

□

3.9 Muller-, Rabin- und Streett-Spiele

Für Muller-, Rabin- und Streett-Spiele liessen sich zwar die Definitionen der Spiele ins Symbolische übertragen, aber von einer algorithmischen Formulierung der Lösungsverfahren ist abgesehen worden, da kein Gewinn zu erwarten gewesen wäre.

Def. 3.8 (Muller Spiel - [Mul63]) *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph und $\Lambda = \{\lambda_1, \ldots, \lambda_r\}$ eine Menge von Formeln über den Variablen von V. Die Muller-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \exists i \in \{i, \ldots, r\} : (In(\rho) \Leftrightarrow \lambda_i)$$

Im Abstrakten sind die Muller-Spiele mittels der LAR-Konstruktion auf die Paritätsspiele reduziert worden, um die Gewinnbereiche W_0 und W_1 der Spieler 0 bzw. 1 zu bestimmen. Eine wesentliche Eigenschaft von LARs ist die sortierte Liste der Zustände, wobei die Reihenfolge der Zustände eine Aussage darüber macht, wann die Zustände zuletzt besucht worden sind. Sortierte Mengen lassen sich nicht einfach ins Symbolische transformieren. Denn boolesche Formeln stellen unsortierte Mengen dar - es gibt nur erfüllende Belegungen einer Formel, aber eine Reihenfolge dabei gibt es nicht. Um dieses zu verdeutlichen werden die beiden Zustände $\binom{0}{0}$ und $\binom{1}{0}$ betrachtet. Als eine boolesche Formel ergeben die beiden Zustände zusammen $\neg v_1$. Diese Formel hat zwar die beiden erfüllenden Belegungen der beiden Zustände, aber eine Reihenfolge ist damit nicht ausdrückbar. Ein weiteres Problem bei einer solchen Darstellung liegt darin, dass die Zustände keine Belegung aller Variablen mehr sind und somit die Algorithmen für die Paritätsspiele nicht laufen würden. Diesen Effekt gab es schon bei den Staiger-Wagner Spielen und konnte dort nur durch ein exponentielles Wachstum der nötigen Variablenanzahl umgangen werden. Somit würde eine symbolische Formulierung auf Einzelzustände zurückführen und dementsprechend keinen Gewinn erbringen.

Def. 3.9 (Rabin-Spiel - [Rab69, Rab72]) *Sei $G = (V, \varphi_0, \varphi_1, \tau)$ ein Spielgraph und $\Omega = ((\phi_1, \psi_1), (\phi_2, \psi_2), \ldots, (\phi_r, \psi_r))$ mit ϕ_i, ψ_i Formeln über V. Die Rabin-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigvee_{i=1}^{r} (In(\rho) \wedge \phi_i = false \wedge In(\rho) \wedge \psi_i \neq false)$$

Def. 3.10 (Streett-Spiel - [Str82]) *Sei* $G = (V, \varphi_0, \varphi_1, \tau)$ *ein Spielgraph und* $\Omega = ((\phi_1, \psi_1), (\phi_2, \psi_2), \ldots, (\phi_r, \psi_r))$ *wie oben. Die Streett-Bedingung definiert die Menge der Gewinnpartien für Spieler 0 durch*

$$\rho \in Win_0 :\Leftrightarrow \bigwedge_{i=1}^{r} (In(\rho) \wedge \psi_i \neq false \rightarrow In(\rho) \wedge \phi_i \neq false)$$

Die für Reduktion auf die Paritätsspiele benutzte IAR-Konstruktion hat genauso wie die LAR-Konstruktion bei den Mullerspielen als wesentlichen Bestandteil eine sortierte Liste, somit gelten die bei den Mullerspielen gemachten Aussagen.

3.10 Erreichte Ergebnisse

Tabelle 3.2 listet die Spiele auf, für die eine symbolische Lösung gefunden wurde. Offen bleiben in dieser Arbeit die Spiele, die in Tabelle 3.3 angeben sind. Die Gründe dafür können im Kapitel 3.9 nachgelesen werden.

Spiel	**kompl. Spiel**	**Lösungsmethode**
Garantiespiel	Sicherheitsspiel	Attraktor
schwaches Paritätsspiel		mittels Attraktor-Berechnung
Staiger-Wagner-Spiel		Reduktion auf schwaches Paritätsspiel
Request-Response-Spiel		Reduktion auf Büchi-Spiel
Büchi-Spiel		Lösung mittels $Attr^{+}$ und *Recur*
Paritätsspiel		McNaughtons Algorithmus

Tabelle 3.2: Spiele für die eine symbolische Lösung gefunden wurde

Spiel	**kompl. Spiel**	**Lösungsmethode**
Muller-Spiel		Reduktion auf Paritätsspiel (LAR)
Rabin-Spiel	Streett-Spiel	Reduktion auf Paritätsspiel (IAR)

Tabelle 3.3: Spiele, die nicht im Symbolischen umgesetzt worden sind

Kapitel 4

Einsatz der Implementierung und Fallstudien

4.1 Die Eingabesprache

Die Eingabesprache ist ein zentraler Teil der Konzeption, da sie die Interaktion zwischen dem entstandenen Programm und dem Benutzer maßgeblich beeinflusst. Denn sie kommt überall dort zum Einsatz, wo der Benutzer etwas spezifizieren muss, wie z.B. bei der Eingabe des Spielgraphens oder der Gewinnbedingung. Demzufolge sind die Anforderungen an die Eingabesprache:

- Sie sollte möglichst einfach zu erlernen sein, damit der Benutzer rasch Ergebnisse erzielen kann.
- Die mit der Eingabesprache definierten Formeln sollten übersichtlich sein, damit sie auch anschliessend noch verstanden werden können.
- Sie sollte möglichst ausdrucksstark sein, damit Formeln kurz und prägnant spezifiziert werden können, wobei dieser Punkt immer etwas im Gegensatz zu dem ersten steht.

Diesen Anforderungen probiert die hier eingesetzte Eingabesprache gerecht zu werden. Dabei sind die Merkmale dieser Sprache:

- Der Sprachumfang boolescher Formeln ist vollständig umgesetzt worden, dabei werden die Operatoren Not, And, Or, XOr, NAnd, NOr und weitere unterstützt.
- Die booleschen Variablen haben die Form $x[i]$, wobei i der Variablenindex ist. Bei der Transitionsformel gibt es noch die Variablen $x'[i]$, um das Ziel einer Transition formulieren zu können. Dabei beginnt die Zählung des Variablenindex bei 0.
- Es gibt existenzielle und universelle Quantoren für die Variablenindices - $Ei\{i < 3\}\ x[i]$ zum Beispiel bedeutet, dass eine der Variablen $x[0]$, $x[1]$ oder $x[2]$ *true* sein soll.
- Eine Arithmetik für die Variablenindices ist eingeführt worden - z.B. $x[i + 3]$.
- Eine Unterstützung für externe Parameter ist eingebaut worden. Somit können Variablen vorab definiert werden, um sie anschliessend in den Formeln einzusetzen.

4.1.1 Grammatik der Sprache

Die Eingabesprache ist durch die Grammatik in Tabelle 4.1 spezifiert, welche auch als Grundlage für die Implementierung herangezogen wurde.

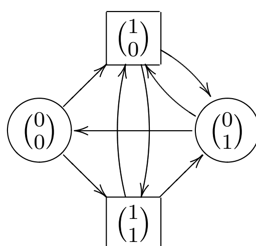
<table>
<tr><th></th><th></th><th>Produktionen</th><th>Bedeutung</th></tr>
<tr><td>Term</td><td>→</td><td>Quantor Var OpVarCond Term |
TermElem BinaryOp Term |
TermElem</td><td>Quantor Variablenindices
Binäre Operation</td></tr>
<tr><td>Var</td><td>→</td><td>$[a-z]^+$</td><td></td></tr>
<tr><td>Quantor</td><td>→</td><td>“E“ |
“A“</td><td>Existenzieller Quantor
Universeller Quantor</td></tr>
<tr><td>OpVarCond</td><td>→</td><td>“{“ VarCond “}“ | ϵ</td><td>VarCond oder leer</td></tr>
<tr><td>BinaryOp</td><td>→</td><td>“&“ | “|“ | “<>“ |
“!&“ | “!|“ |
“->“ | “=“ |
“<-“ |
“-“ | “<“</td><td>And, Or, XOr
NAnd, NOr
Implication, Biimplication
Reverse Implication
Difference, Less</td></tr>
<tr><td>TermElem</td><td>→</td><td>“(“Term“)“ |
“!(“ Term “)“ |
“x[“ VarIndex “]“ |
“!x[“ VarIndex “]“ |
“x’[“ VarIndex “]“ |
“!x’[“ VarIndex “]“ |
“true“ | “false“</td><td>geklammerter Term
negierter Term
Variable
negierte Variable
Variable für Trans.ergebnis
neg. Variable für Trans.ergebnis
boolesche Werte</td></tr>
<tr><td>VarCond</td><td>→</td><td>VarCond “&“ VarCond |
VarCond “|“ VarCond |
VarCondElem</td><td>And
Or</td></tr>
<tr><td>VarCondElem</td><td>→</td><td>“(“ VarCond “)“ |
Var “=“ VarIndex |
Var “!=“ VarIndex |
Var “<“ VarIndex |
Var “>“ VarIndex</td><td>geklammerte Bedingung
gleich
ungleich
kleiner
größer</td></tr>
<tr><td>VarIndex</td><td>→</td><td>VarIndex “*“ VarIndex |
VarIndex “+“ VarIndex |
VarIndex “-“ VarIndex |
VarIndexElem</td><td>Multiplikation
Addition
Subtraktion</td></tr>
<tr><td>VarIndexElem</td><td>→</td><td>“(“ VarIndex “)“ |
Var | $[0-9]^+$</td><td>geklammerter Ausdruck
Variable oder Wert</td></tr>
</table>

Tabelle 4.1: Grammatik der Eingabesprache

Die verwendete Grammatik benutzt reguläre Ausdrücke auf den rechten Regelseiten, wie beispielsweise die Regel für das Nichtterminal Var zeigt.

4.1.2 Beispiele

(a) Wir greifen zurück auf den Spielgraphen aus dem Beispiel 3.1 und wiederholen ihn nochmal kurz:

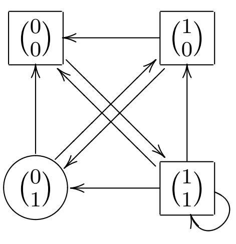


Der abgebildete Spielgraph Spielgraph lässt sich in der gewählten Eingabesprache notieren als:

$$\begin{aligned}
\varphi_0 &= !x[0] \\
\varphi_1 &= x[0] \\
\tau &= (!x[0] \ \& \ !x[1] \ \& \ x'[0]) \mid (!x[0] \ \& \ x[1] \ \& \ !x'[1]) \mid \\
&\quad\ (x[0] \ \& \ !x[1] \ \& \ x'[1]) \mid (x[0] \ \& \ x[1] \ \& \ (x'[0] = !x'[1]))
\end{aligned}$$

(b) Das Beispiel 3.2 lässt sich schreiben als:

$$\begin{aligned}
\varphi_0 &= !x[0] \ \& \ x[1] \\
\varphi_1 &= x[0] \ \& \ !x[1] \\
\tau &= (x[0] \ \& \ !x'[0]) \mid (x[1] \ \& \ !x'[1]) \mid \\
&\quad\ ((x[0] = !x'[0]) \ \& \ (x[1] = !x'[1])) \mid \\
&\quad\ (x[0] \ \& \ x[1] \ \& \ x'[0] \ \& \ x'[1])
\end{aligned}$$



(c) In diesem Beispiel soll eine simple Aufzugssteuerung für 10 Etagen spezifiert werden. Dazu wird der folgende Zustandsraum betrachtet:

$$\underbrace{x[0]\ldots x[9]}_{\text{angefordert}} \ \underbrace{x[10]\ldots x[19]}_{\text{befindet sich in}} \ \underbrace{x[20]}_{\text{Spieler}}$$

Der erste Block signalisiert, welche Etagen angefordert sind. Der zweite gibt an, auf welcher Etage sich der Aufzug befindet und die letzte Variable sagt aus, welcher Spieler an der Reihe ist. Dabei ist Spieler 0 die Aufzugssteuerung und Spieler 1 seien die Benutzer, die den Aufzug anfordern.

$$x[0], \ldots, x[19], 0 \xrightarrow[\text{liefert Personen ab}]{\text{Tür schließt, Lift fährt,}} x'[0], \ldots, x'[19], 1$$

Dabei soll der Aufzug die Etage wechseln und alle Anforderungen bleiben bestehen, bis auf die, in der sich der Aufzug nach dem Zug befindet.

$$x[0], \ldots, x[19], 1 \xrightarrow[\text{drücken Knöpfe}]{\text{Benutzer}} x'[0], \ldots, x'[19], 0$$

Der Aufzug bleibt auf der selben Etage stehen, es kommen nur neue Anforderungen hinzu.

Die erste Formel besteht aus mehreren Elementen, die konjunktiv miteinander verbunden sind:

- !x[20] & x'[20] : beschreibt die Zugehörigkeit der Knoten zu den Spielern
- Ei{(i>9) & (i<20)} x[i] & Aj{(j<9) & (j<20) & (j! =i)} !x[j] : sagt aus, dass sich der Aufzug in genau einer Etage befunden hat
- Ek{(k>9) & (k<20) & (k!=i)} x'[k] & !x'[k-10] & Al{(l<9) & (l<20) & (l!=k)} !x[l] : sagt aus, dass sich der Aufzug anschliessend wieder genau in einer Etage befindet, die unterschiedlich zu der Startetage ist, und die Anforderung der Zieletage zurückgesetzt ist
- Am{(m<10) & (m!=k)} x'[m]=x[m] : alle Anforderungen bleiben bestehen, bis auf die Etage, in der sich der Aufzug anschliessend befindet

Die zweite Formel setzt sich auch wieder aus verschiedenen Elementen zusammen:

- x[20] & !x'[20] : beschreibt die Zugehörigkeit der Knoten zu den Spielern
- Ai{(i<10)} x[i]->x'[i] : es kommen nur Anforderungen dazu
- Aj{(j>9) & (j<20)} x[j]=x'[j] : der Aufzug bleibt in der selben Etage stehen

4.2 Die einzelnen Seiten der GUI

Um das Programm SymProg, welches die Ergebnisse aus dem letzten Kapitel praktisch umsetzt, zu starten, sollte einfach die beiliegende Batchdatei `runme.bat` bzw. das Shellskript `runme.sh` ausgeführt werden. Alternativ kann auch direkt die Klasse `rwthaachen.i7.symprog.gui.SymProg` aufgerufen werden. Dazu muss aber das für das eigene Betriebssystem passende `swt.jar` in die Shell-Variable `CLASSPATH` aufgenommen werden, weiterhin muss unter Linux das Verzeichnis, in dem sich die BDD- und SWT-Bibliotheken befinden, in der Shell-Variablen `LD_LIBRARY_PATH` hinzugefügt werden. Damit das Programm läuft, muss auf dem Rechner ein Java Runtime Environment (JRE) in der Version 1.3.0 oder höher installiert sein.

Dieser Abschnitt soll die Benutzung des entstandenen Programmes verdeutlichen. Dazu wird im folgenden auf die einzelnen Seiten der Benutzeroberfläche eingegangen. Als Grundlage für die hier abgebildeten Screenshots dient das Beispiel `parity.game` (siehe Anhang B). Zur leichteren Lesbarkeit sei es hier präsentiert:

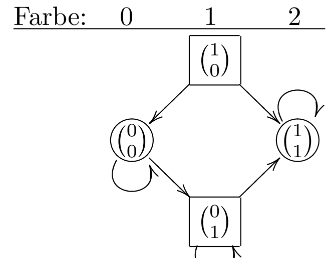


Abbildung 4.1: Beispiel für die Programmbeschreibung

4.2.1 Die Eingabeseite

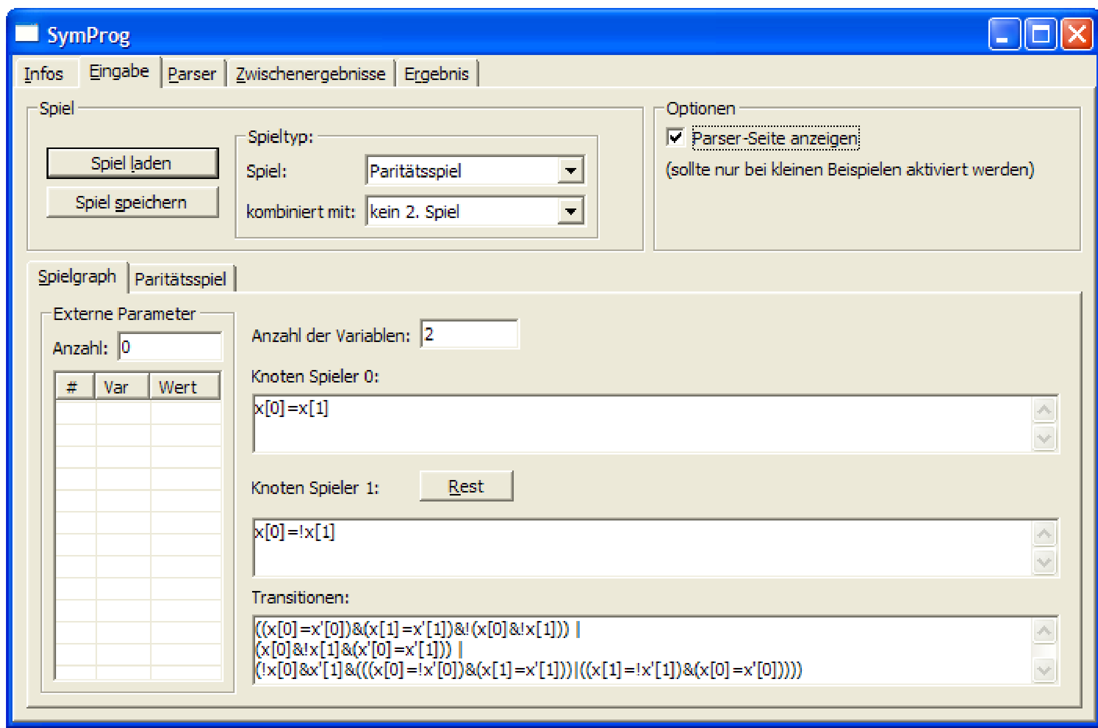


Abbildung 4.2: Eingabe-Seite

Die in Abbildung 4.2 dargestellte Eingabe-Seite ist für den Benutzer des Programmes die zentrale Stelle zur Spezifikation des zu betrachtenden Spieles. Dazu sollte als erstes der Spieltyp ausgewählt werden, da die Unterseite für die Akzeptierbedingung vom Spieltyp abhängig ist. Dies geschieht im Kasten Spieltyp. Mit der ComboBox, die mit "Spiel:" beschriftet ist, wird das erste Spiel festgelegt. Wird ein Spiel in der ComboBox, welche mit "kombiniert mit:" betitelt ist, ausgewählt, wird zuerst das erste ausgewählte Spiel auf dem spezifizierten Spielgraphen ausgewertet und anschliessend auf dem Gewinnbereich des ersten Spieles noch das zweite Spiel berechnet. Somit ist es möglich, beispielsweise Sicherheits- mit Request-Response-Bedingungen zu kombinieren. Wird ein zweites Spiel ausgewählt, wird eine weitere Unterseite für die Akzeptierbedingung des zweiten Spieles eingeblendet (analog zum ersten Spiel).

Natürlich bietet das Programm auch die Möglichkeit, Spiele zu speichern bzw. gespeicherte Spiele zu laden. Eine Übersicht der mitgelieferten Beispiele befindet sich im Anhang B. Die Spiele werden nicht als Textdateien gespeichert, sondern als Binärdateien, da sich diese im Programm einfacher verarbeiten lassen. Dies könnte als Nachteil betrachtet werden, da sich so die Dateien nicht einfach in einem normalen Editor bearbeiten lassen; dafür können aber alle Einstellungen einfach und übersichtlich in diesem Programm bearbeitet werden.

Mit der CheckBox "Parser-Seite anzeigen" wird gesteuert, ob die Parser-Seite angezeigt wird. Diese Option sollte nur bei kleinen Beispielen aktiviert werden, da sich dadurch der Vorgang zwischen dem Parsen der Formeln und dem Lösen des Spieles deutlich vergrößert und die Ergebnisse (zumindest bei der Transitionsformel) unüberschaubar werden.

Der Kasten 'Externe Parameter' dient dazu, Variablen zu definieren, die in den Formeln für die Spieler-Knoten, der Transitionsformel, der Gewinnbedingung(en) oder der Variablenanzahl benutzt werden können. Somit lässt sich beispielsweise eine Aufzugssteuerung unabhängig von der Etagenanzahl spezifizieren. Die Etagenanzahl würde dann in diesem Kasten definiert. Für die Eingabe von Werten in Tabellen bleibt noch zu erwähnen, dass die Enter-Taste gedrückt werden muss, um in den Eingabe-Modus zu wechseln. Um diesen wieder zu verlassen, muss wieder die Enter-Taste gedrückt werden.

Auf dem Rest der Seite kann dann noch der Spielgraph des Spieles festgelegt werden. Dazu muss als erstes die Anzahl der zu verwendenden Variablen festgesetzt werden und anschliessend sollten dann in den entsprechenden Eingabefeldern boolesche Formeln für die Knoten der Spieler sowie der zulässigen Transitionen definiert werden. Mit dem Button "Rest" auf der Seite kann die negierte Formel der Spieler 0-Knoten als Spieler 1-Knotenformel eingetragen werden. Sämtliche einzugebenden Formeln müssen von der im letzten Abschnitt vorgestellten Grammatik ableitbar sein.

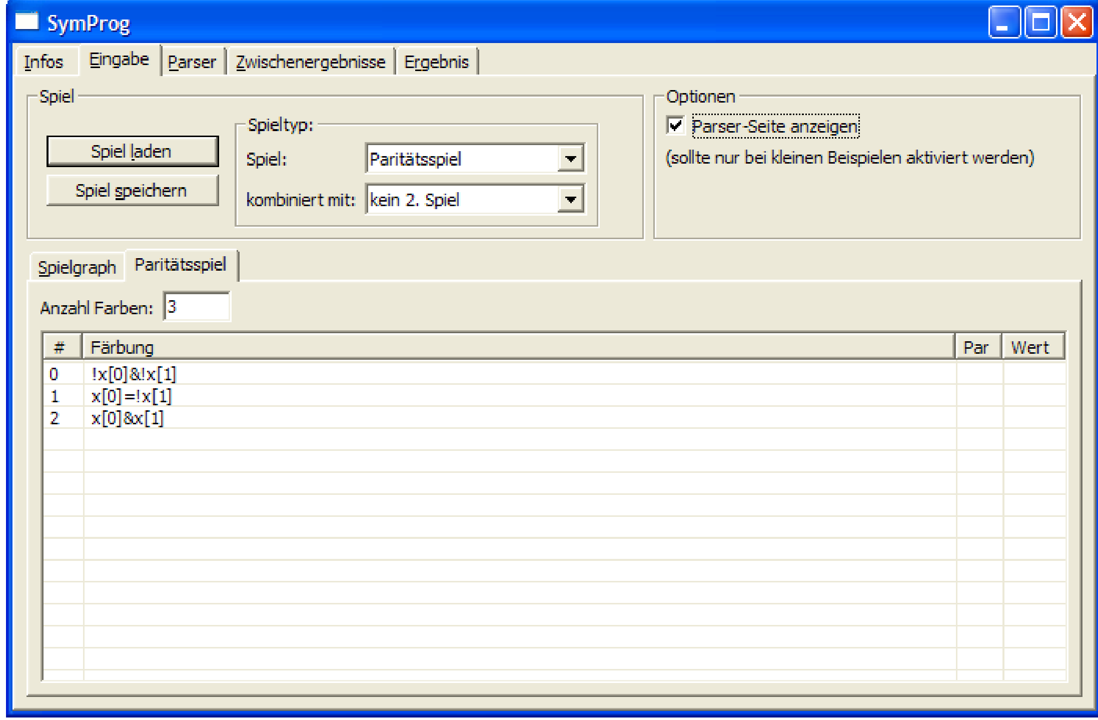


Abbildung 4.3: Eingabe-Seite - Akzeptierbedingung

Abbildung 4.3 zeigt eine der möglichen Akzeptierbedingungen. Das Aussehen dieser Seite hängt von dem ausgewählten Spiel ab. In dem konkreten Beispiel wird auf dieser Seite die Färbung für den Spielgraphen definiert. Ist ein zweites Spiel ausgewählt worden, wird eine weitere Seite zur Eingabe der Akzeptierbedingung des zweiten Spieles angezeigt.

Bei allen Akzeptierbedingungen, die in tabellarischer Form eingegeben werden können, gibt es zwei zusätzliche Spalten, in denen ein Parameter und ein Wert angegeben werden können. Wird diese Option genutzt, wird die einzelne Tabellenzeile durch soviele Zeilen ersetzt, wie in der Spalte Wert angegeben wurde, und der Parameter, der in der vorletzten Spalte angegeben

wurde, wird dabei durch die Werte 0 bis Wert-1 substituiert. Somit lassen sich beispielsweise bei einer Aufzugssteuerung mittels einer einzigen Zeile die Request-Response-Bedingungen definieren, dass der Aufzug jede angeforderte Etage auch anfahren muss.

4.2.2 Die Parser-Seite

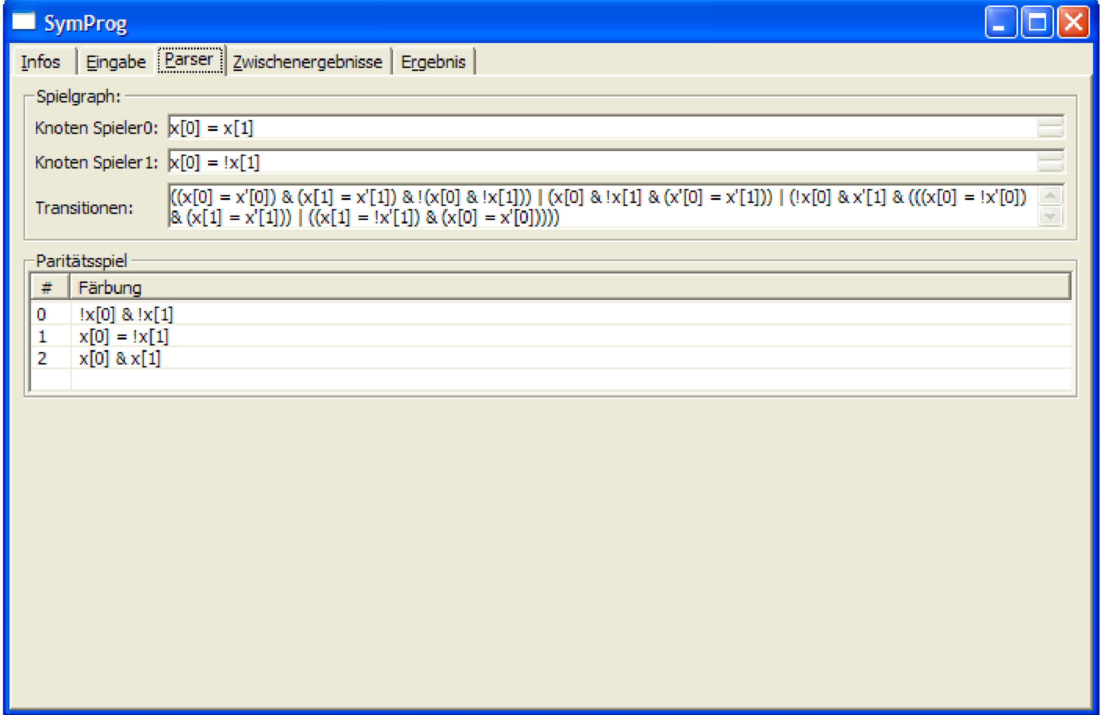


Abbildung 4.4: Parser-Seite

Die Parser-Seite (Abbildung 4.4) ist eine reine Informationsseite und bietet für den Benutzer keine Eingabemöglichkeiten. Auf dieser Seite wird dargestellt, was der Parser aus den eingegebenen Formeln gemacht hat (siehe dazu auch Kapitel 5.3.4). Dies ist vor allen Dingen dann interessant, wenn in den Formeln Quantoren zum Einsatz kommen. Denn so ist erkennbar, wie die Quantoren ausgelöst worden sind. Dabei dürfte auffallen, dass der Einsatz von Quantoren bei vielen Variablen die Formel sehr deutlich aufbläht. Natürlich lässt sich daraus noch nicht auf die Komplexität des resultierenden BDDs schliessen, da der BDD eine minimale Darstellung der booleschen Formel ist. Trotz allem sollten Quantoren mit Bedacht eingesetzt werden.

Wurde ein zweites Spiel ausgewählt, erscheint ein zweiter Kasten, indem das Ergebnis des Parsers der Akzeptierbedingung des zweiten Spieles dargestellt wird.

4.2.3 Die Zwischenergebnis-Seite

Die Zwischenergebnis-Seite, dargestellt in Abbildung 4.5, zeigt, wie der jeweilige Algorithmus zur Berechnung der Gewinnbereiche und -strategien arbeitet. Durch die hierarchische

```
SymProg
Infos | Eingabe | Parser | Zwischenergebnisse | Ergebnis
Paritätsspiel
  parity(((!x[0]&!x[1])|(x[0]&x[1])),((!x[0]&x[1])|(x[0]&!x[1])),2,0)
    attraktor(((!x[0]&!x[1])|(x[0]&x[1])), ((!x[0]&x[1])|(x[0]&!x[1])), x[0]&x[1]) = x[0]&x[1]
      attr(x[0]&x[1]) = x[0]&x[1]
      attr(x[0]&x[1]) = x[0]&x[1]
    parity(!x[0]&!x[1],((!x[0]&x[1])|(x[0]&!x[1])),1,0)
      parity(((!x[0]&x[1])|(x[0]&!x[1])),!x[0]&!x[1],2,1)
        attraktor(((!x[0]&x[1])|(x[0]&!x[1])), !x[0]&!x[1], ((!x[0]&x[1])|(x[0]&!x[1]))) = ((!x[0]&x[1])|(x[0]&!x[1]))
          attr(((!x[0]&x[1])|(x[0]&!x[1]))) = ((!x[0]&x[1])|(x[0]&!x[1]))
          attr(((!x[0]&x[1])|(x[0]&!x[1]))) = ((!x[0]&x[1])|(x[0]&!x[1]))
        parity(false,!x[0]&!x[1],1,1)
          parity(!x[0]&!x[1],false,2,2)
            attraktor(!x[0]&!x[1], false, !x[0]&!x[1]) = !x[0]&!x[1]
              attr(!x[0]&!x[1]) = !x[0]&!x[1]
              attr(!x[0]&!x[1]) = !x[0]&!x[1]
            parity(false,false,1,2)
        attraktor(!x[0]&!x[1], ((!x[0]&x[1])|(x[0]&!x[1])), !x[0]&!x[1]) = !x[1]
          attr(!x[0]&!x[1]) = !x[1]
          attr(!x[1]) = !x[1]
        parity(!x[0]&x[1],false,2,1)
          attraktor(!x[0]&x[1], false, !x[0]&x[1]) = !x[0]&x[1]
            attr(!x[0]&x[1]) = !x[0]&x[1]
            attr(!x[0]&x[1]) = !x[0]&x[1]
          parity(false,false,1,1)
```

Abbildung 4.5: Zwischenergebnis-Seite

Struktur der Tree-Komponente ist gut erkennbar, wie die Berechnung abgelaufen ist. Wie die Abbildung zeigt, wird bei einem Aufruf der `parity`-Funktion zuerst ein Attraktor berechnet und anschliessend die `parity`-Funktion rekursiv für das Restspiel aufgerufen. Wird eine Bedingung nicht erfüllt (siehe dazu Kapitel 3.8), wird zum Gewinnbereich des Spielers 1 der Attraktor berechnet und anschliessend für das Restspiel rekursiv die `parity`-Funtkion aufgerufen.

4.2.4 Die Ergebnis-Seite

Auf der Ergebnis-Seite (Abbildung 4.6) wird die Lösung des Spiels ausgegeben. Dazu werden neben den booleschen Formeln für die Gewinnbereiche der Spieler 0 und 1 auch die Mächtigkeiten dieser Gewinnbereiche angegeben. Desweiteren werden auch die Zeiten angezeigt, die zum Parsen der Formeln und zur Lösung des Spieles gebraucht wurden. Diese Angaben bilden auch die Grundlage für die Zeitangaben bei der Fallstudie im nächsten Abschnitt.

Daneben gibt es noch die Möglichkeit, die berechnete Gewinnstrategie zu überprüfen. Dazu kann im unteren Bereich der Seite ein Zustand angegeben werden, und das Programm liefert dann die möglichen Nachfolgezustände gemäß der Gewinnstrategie. In dem konkret abgebildeten Beispiel wurde die Gewinnstrategie für den Knoten $\binom{0}{1}$ abgefragt. Da es sich bei dem betrachteten Spiel um ein Paritätsspiel handelt, der Knoten die Farbe eins hat und es eine Transition zum selben Knoten zurück exisitiert, wird als Ergebnis der Knoten wieder ausgegeben.

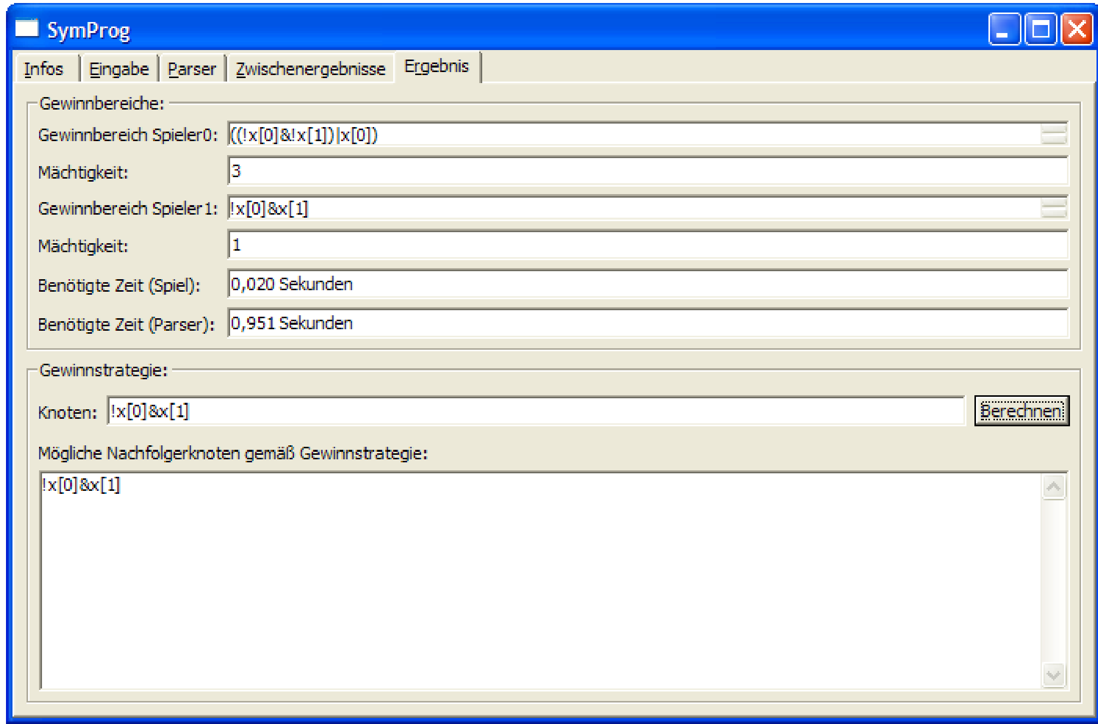


Abbildung 4.6: Ergebnis-Seite

4.3 Fallstudie

In diesem Abschnitt wird gezeigt, dass die in vorherigen Kapiteln beschriebenen Verfahren beispielsweise dazu benutzt werden können, eine Aufzugssteuerung zu verifizieren.

Konkret wird ein Aufzugssystem betrachtet, welches aus 2 Aufzügen in einem Haus mit e Etagen besteht - dargestellt in Abbildung 4.7. Neben der für einen Aufzug erwarteten Anforderung, dass jede angeforderte Etage auch angefahren wird, hat das Aufzugssystem ebenfalls noch die folgenden Bedingungen zu erfüllen:

- Die Aufzüge steuern niemals dieselbe Etage an.
- Die höchste und die niedrigste Etage werden immer direkt bedient.
- Es werden immer mindestens drei Etagen nicht angefordert.
- In der zweiten Etage (als externer Parameter po konfigurierbar) befindet sich die Poststelle und der Aufzug, der diese Etage anfährt, muss einen Zug der Aufzugssteuerung aussetzen, damit die Post ein- und ausgeladen werden kann.
- Kein Aufzug fährt an einer angeforderten Etage vorbei, d.h. wenn sich ein Aufzug von der i-ten zur j-ten Etage bewegt, sind im Zielzustand alle Anforderungen zwischen der i-ten und der j-ten Etage nicht gesetzt.
- Die Aufzüge sollen immer wieder das Erdgeschoss anfahren.

Abbildung 4.7: Schema der Aufzugssteuerung

Somit stellt sich die Frage, wie die aufgestellten Anforderungen an die beide Aufzüge in dem Spielgraphen und der Gewinnbedingung codiert werden können. Dazu sollte als erstes der Spieltyp festgelegt werden. Für diesen konkreten Fall ist ein Request-Response Spiel zu bevorzugen, da somit am einfachsten die Anforderung realisiert werden kann, dass jede von der Umgebung angeforderte Etage auch wirklich von einem der beiden Aufzüge bedient wird. Ebenfalls lässt sich durch ein Request-Response-Paar die letzte Anforderung formulieren, dass das Erdgeschoss immer wieder angefahren werden soll. Alle anderen Anforderungen müssen über den Spielgraphen codiert werden, genauer gesagt beeinflussen die anderen Anforderungen an die beiden Aufzüge die Transitionsformel und die Anzahl der benötigten Variablen. Beispielsweise folgt aus der Anforderung, dass der Aufzug in der Postetage eine Runde aussetzen muss, dass eine weitere Variable benötigt wird, um sich zu merken, dass der Aufzug noch eine Runde pausieren muss. In dem RR-Spiel ist Spieler 0 die Aufzugssteuerung und Spieler 1 die Umgebung, die die Aufzüge anfordert. Damit sichergestellt ist, dass jeder Zustand einen Nachfolgezustand hat, wird ein Senkzustand eingeführt, zu dem von allen Spieler 0-Knoten aus eine Transition hinführt.

4.3.1 Spieldefinition

Der für das Aufzugssystem zugrunde liegende Spielgraph benutzt bei e Etagen $(3 \cdot e + 2)$ Variablen:

- $x[0] \dots x[e-1]$: Position des ersten Liftes (nur eine einzige Variable davon ist true, alle anderen sind false)
- $x[e] \dots x[2e-1]$: analog für den zweiten Aufzug
- $x[2e] \dots x[3e-1]$: die angeforderten Etagen (mindestens drei Variablen davon sind false, die Etagen, in denen sich die beiden Aufzüge befinden, sind auch nicht angefordert)
- $x[3e]$ gibt die Zugehörigkeit zu den Spielern an. Ist die Variable `false`, ist es ein Knoten der Aufzugssteuerung, und ansonsten ist es ein Knoten der Umgebung.
- $x[3e+1]$ signalisiert, dass ein Aufzug in der zweiten Etage ist (Poststelle) und noch eine Runde aussetzen muss.

Bei vier Etagen werden somit 14 Variablen benötigt. In der Tabelle 4.2 sind zwei mögliche Zustände des Spielgraphens dargestellt. Dazu sind für die beiden ausgewählten Zustände die Belegungen aller vierzehn Variablen angegeben.

Bei dem mit (a) gekennzeichneten Zustand befinden sich die beiden Aufzüge im Erdgeschoss bzw. in der dritten Etage. Als einzige Etage ist die Postetage angefordert worden und der Zustand ist ein Zustand der Aufzugssteuerung. In dem mit (b) markierten Zustand sind die Aufzüge in der ersten und zweiten Etage, es sind keine Etagen angefordert worden und der Zustand gehört zu der Umgebung, die Aufzüge anfordern kann. Durch die Variable 13 wird

0	1	2	3	4	5	6	7	8	9	10	11	12	13	Bez.
1	0	0	0	0	0	0	1	0	0	1	0	0	0	(a)
0	1	0	0	0	0	1	0	0	0	0	0	1	1	(b)

Tabelle 4.2: Zwei mögliche Zustände bei vier Etagen

sichergestellt, dass sich der zweite Aufzug auch nach dem nächsten Zug der Aufzugssteuerung noch in der Postetage befindet.

Die Request-Response Paare ergeben sich zu:

Request (ϕ_i)	Response(ψ_i)	Par	Wert
!(x[0]|x[e])	x[0]|x[e]		
x[(2*e)+k+1]	x[k+1]|x[k+e+1]	k	e-2

Tabelle 4.3: Request-Response Paare der Fallstudie

Das erste Paar drückt aus, dass, wenn sich keiner der beiden Aufzüge im Ergeschoss befindet, sie irgendwann mal wieder das Erdgeschoss anfahren sollen. Das zweite Paar wird wie in Kapitel 4.2.1 beschrieben durch $e - 2$ Paare ersetzt, wobei k dabei die Werte 0 bis $e - 3$ annimmt. Mit diesem Paar wird für die Etagen zwischen dem Erdgeschoss und der höchsten Etage ausgedrückt, dass sie, wenn sie angefordert sind, auch angefahren werden müssen. Das Erdgeschoss und die höchste Etage sind nicht darin enthalten, da diese direkt angefahren werden sollen und dieses Verhalten in der Transitionsformel modelliert wird.

Die Formel der Spieler 0-Knoten ergibt sich zu:

```
!x[3*e] &
(((Ei{i<e}(x[i] & !x[i+(2*e)] & Ej{(j>e-1)&(j<2*e)&(j!=i+e)}(x[j] & !x[j+e] &
Ak{(k<2*e) & (k!=i) & (k!=j)} !x[k]))) &
(x[(3*e)+1]->(x[po]|x[e+po])) &
(Ei{(i>(2*e)-1)&(i<3*e)}(!x[i] & Ej{(j>(2*e)-1)&(j<3*e)&(j!=i)}(!x[j] &        5
Ek{(k>(2*e)-1)&(k<3*e)&(k!=i)&(k!=j)} !x[k])))) | (Ai{i<(3*e)+2} !x[i]))
```

Algorithmus 4.1: Formel Spieler 0-Zustände

Die einzelnen Zeilen der Formel bedeuten:

- In Zeile 1 wird das Flag des Spieler nicht gesetzt → ein Spieler 0-Knoten.
- Die Zeilen 2-3 bewirken, dass sich die beiden Aufzüge jeweils in genau einer Etage befinden und die Anforderungen dieser beiden Etagen, die unterschiedlich sein müssen, nicht gesetzt sind.
- Das Flag zum Verbleib in der Postetage wird in Zeile 4 nur gesetzt, wenn einer der beiden Aufzüge in der Postetage ist.
- Die Zeilen 5 und 6 stellen sicher, dass mindestens drei Etagen nicht angefordert werden können.

- Weiterhin ist der Zustand, in dem alle Variablen false sind, ein Spieler 0-Zustand. Dieser Zustand ist der bereits beschriebene Senkzustand.

Die Formel für die Spieler 1-Knoten ist analog aufgebaut, mit dem Unterschied, dass das Flag für Spieler 1 gesetzt ist und der Senkzustand herausfällt.

Die Transitionsformel ist im Algorithmus 4.2 angegeben. Die einzelnen Elemente der Formel bewirken:

```
((Ei{i<e}(x[i]&!x[i+(2*e)]& Ej{(j>e-1)&(j<2*e)&(j!=i+e)}(x[j]!&!x[j+e] &
  (Ak{(k<2*e) & (k!=i) & (k!=j)} !x[k]) &
 ((El{l<e}(x'[l]&!x'[(2*e)+l]& Em{(m>e-1)&(m<2*e)&(m!=l+e)}(x'[m]&!x'[m+e] &
  (An{(n<2*e) & (n!=l) & (n!=m)} !x'[n]) &
                                                                                5
  ((((Ap{((p>i)&(p<l))|((p>l)&(p<i))}!x'[p+(2*e)]) &
  (Aq{((q>j)&(q<m))|((q>m)&(q<j))} !x'[e+q])) &
    (Ao{(o>(2*e)-1)&(o<3*e)&(o!=l+(2*e))&(o!=m+e)}(x[o]=x'[o])) &
    !x[3*e] & x'[3*e] &
    (x[(3*e)-1]->(x'[e-1]|x'[(2*e)-1])) &                                      10
    (x[2*e]->(x'[0]|x'[e])) &
    ((x[(3*e)+1]&x[po])->x'[po])&((x[(3*e)+1]&x[e+po])->x'[e+po])&
    ((x'[po]|x'[e+po])->((!x[(3*e)+1]&x'[(3*e)+1])|(x[(3*e)+1]&!x'[(3*e)+1])))&
    ((!x'[po]&!x'[e+po]) -> !x'[(3*e)+1])) |
                                                                               15
   ((Ap{(p>(2*e)-1)&(p<3*e)}x[p]->x'[p]) & (Aq{(q<2*e)} x[q]=x'[q]) &
    x[3*e] & !x'[3*e] & (x[(3*e)+1]=x'[(3*e)+1])))     ))) |

  (!x[3*e] & (Al{l<(3*e)+2} !x'[l])) )))) &
(x[(3*e)+1]->(x[po]|x[po+e])) &                                                20
(Ei{(i>(2*e)-1)&(i<3*e)}(!x[i] & Ej{(j>(2*e)-1)&(j<3*e)&(j!=i)}(!x[j] &
   Ek{(k>(2*e)-1)&(k<3*e)&(k!=i)&(k!=j)} !x[k]))) &
(Ei{(i>(2*e)-1)&(i<3*e)}(!x'[i] & Ej{(j>(2*e)-1)&(j<3*e)&(j!=i)}(!x'[j] &
   Ek{(k>(2*e)-1)&(k<3*e)&(k!=i)&(k!=j)} !x'[k])))) |
(Ai{i<(3*e)+2} !x[i] & !x'[i])                                                 25
```

Algorithmus 4.2: Transitionsformel der Fallstudie

- Die Zeilen 1 und 2 stellen sicher, dass sich die beiden Aufzüge im Startzustand der Transition jeweils in genau einer Etage befinden und die Anforderungen dieser beiden Etagen, die unterschiedlich sein müssen, nicht gesetzt sind.
- Zeilen 3 und 4 analog für den Zielzustand
- Zeilen 6 bis 15 behandeln die von Spieler 0-Knoten ausgehenden Transitionen
  - Die Zeilen 6 und 7 stellen sicher, dass die Aufzüge an keiner angeforderten Etage vorbeifahren.
  - Die Zeile 8 übernimmt die Anforderungen ohne die beiden Etagen, in die die Aufzüge gefahren sind.

  - Die Zeile 9 sagt aus, dass die Transition von einem Spieler 0-Zustand zu einem Spieler 1-Zustand führt.
  - Die Zeilen 10 bis 11 garantieren, dass das Erdgeschoss und die höchste Etage direkt angefahren werden, wenn sie angefordert wurden.
  - Durch Zeile 12 ist sichergestellt, dass ein Aufzug, wenn er in der Postetage ist und das Warten-Flag gesetzt ist, auch dort bleibt.
  - Die Zeilen 13 und 14 setzen das Warten-Flag, wenn ein Aufzug die Postetage erreicht hat. Ansonsten wird es gelöscht.

- Die Zeilen 16 und 17 beschreiben die von Spieler 1-Zuständen ausgehende Transitionen. Dabei werden nur Anforderungen hinzugefügt und die Kontrolle an Spieler 0 übergeben.
- Zeile 19 fügt noch für alle Spieler 0-Knoten eine Transition zum Senkzustand hinzu.
- Zeile 20 stellt sicher, dass das Warten-Flag nur gesetzt ist, wenn auch ein Aufzug in der Postetage ist.
- Die Zeilen 21 und 22 stellen sicher, dass mindestens drei Etagen im Startzustand nicht angefordert sind. Die Zeilen 23 und 24 verlaufen analog für den Zielzustand der Transition.
- Zeile 25 fügt noch die Transition von Senkzustand zu sich selber hinzu.

4.3.2 Ergebnisse

Das Programm wurde bei diesem Spiel für kleine Werte der Etagenanzahl laufen gelassen. Hierbei ergab sich erwartungsgemäß, dass die Aufzugssteuerung ab einer gewissen Etagenanzahl, in diesem Fall ab fünf Etagen, nicht mehr alle Anforderungen erfüllen kann und somit der komplette Spielgraph Gewinnbereich der Umgebung ist. Bis zu dieser Etagenanzahl kann die Aufzugssteuerung von allen Zuständen außer dem Senkzustand die Anforderungen erfüllen. Tabelle 4.4 fasst die Ausgabe des Programmes zusammen. Dort finden sich unter anderem Informationen über die benötigte Zeit für das Erzeugen der Formeln und das Lösen des Spieles. Weiterhin sind die Mächtigkeiten der Gewinnbereiche der beiden Spieler angeben, in der Tabelle mit GWB abgekürzt.

Etagen	Variablen	RR-Paare	Zeit Parser	Zeit Spiel	GWB 0	GWB 1
3	11	2	2,32 s	0,42 s	20	1
4	14	3	10,59 s	3,28 s	108	1
5	17	4	237,79 s	43,78 s	0	393

Tabelle 4.4: Ergebnisse der Fallstudie

Die Ergebnisse wurden auf einem Rechner mit einem Intel P3-M Prozessor mit 1GHz und 256MB Ram unter Linux erzielt. Bei 6 Etagen nutzte der Rechner nach über 2 Stunden Laufzeit intensiv die Swap-Partition und hatte zu diesem Zeitpunkt noch nicht die Erzeugung

des JBDD-Objektes für die Transitionsformel abgeschlossen, so dass die Berechnung dann an dieser Stelle abgebrochen wurde.

Eine Analyse der Ergebnisse hat ergeben, dass eine Gewinnstrategie für Spieler 1 ab fünf Etagen existiert. In diesem Fall läßt sich kurz erläutern, wie die Strategie aussieht. Sie besteht darin, einen der beiden Aufzüge in die Postetage zu bekommen, ohne dass weitere Etagen angefordert sind. Dies kann Spieler 1 erreichen, da Spieler 0, um das Request-Response Spiel nicht zu verlieren, irgendwann die Postetage mit einem der beiden Aufzüge ansteuern muss. Danach kann Spieler 1 erreichen, dass Spieler 0 eine Anforderung nicht erfüllen kann, unabhängig davon, in welcher Etage sich der andere Aufzug befindet. Dies liegt daran, dass der Aufzug in der Postetage „gefangen“ genommen wurde und der andere nicht die beiden auflaufenden Anforderungen gleichzeitig erfüllen kann. Bei fünf Etagen fordert Spieler 1 die folgenden Etagen in Abhängigkeit des anderen Aufzuges an:

2. Aufzug	Anforderungen	Grund
Erdgeschoss	1. und 4. Etage	kann nicht an angeforderter Etage vorbeifahren
1. Etage	Erdgeschoss und 4. Etage	beide Expressetagen angefordert
3. Etage	Erdgeschoss und 4. Etage	beide Expressetagen angefordert
4. Etage	Erdgeschoss und 3. Etage	kann nicht an angeforderter Etage vorbeifahren

Tabelle 4.5: Gewinnstrategie des Spielers 1 bei fünf Etagen

Als Ergebnis kann man festhalten: Die anschauliche Gewinnstrategie der Umgebung bei fünf Etagen ist a posteriori aus dem Programmoutput extrahierbar. Für größere Beispiele dürfte selbst dies ein Problem sein.

Kapitel 5

Die Implementierung

5.1 Prozeßentscheidungen

Der Entwicklungsprozeß war nach [Som01] evolutionärer Natur. Ein evolutionärer Softwareentwicklungsprozeß ist dadurch gekennzeichnet, dass zuerst eine initiale Version entwickelt wird, die im weiteren Verlauf immer weiterentwickelt wird und an die Bedürfnisse des Kunden angepasst wird, bis schliesslich ein adäquates System fertiggestellt worden ist. Dieser Vorgang ist in Abbildung 5.1 illustriert. Ein weiteres Merkmal für einen solchen Prozeß ist, dass die Spezifikation, die Entwicklung und die Verifikation nicht als getrennte Aktivitäten durchgeführt werden, sondern parallel stattfinden. Dabei sind zuerst die Teile des Systems entwickelt worden, die vollständig verstanden waren, um das System schliesslich noch um die Merkmale zu erweitern, die noch gebraucht wurden. Dieser Ansatz wird nach [Som01] auch "exploratory development" genannt.

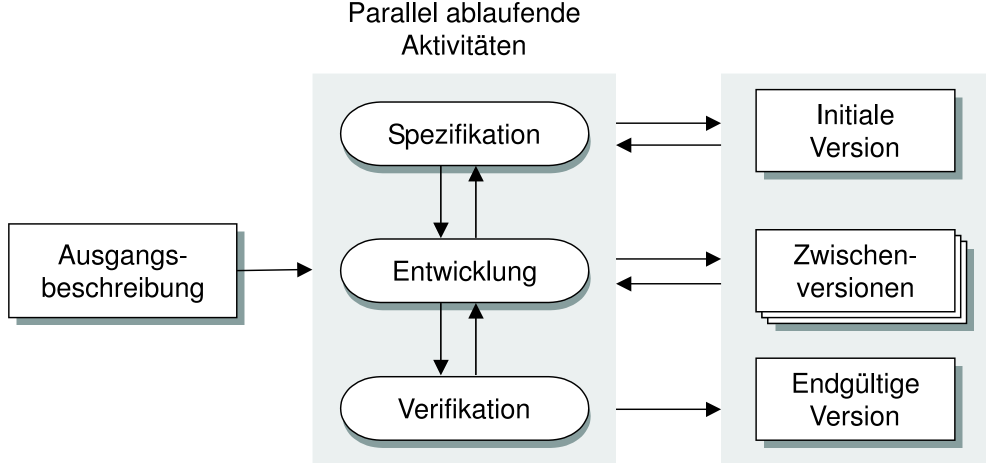


Abbildung 5.1: Prozessmodell "Evolutionary Development" nach [Som01]

Als Ausgangsbeschreibung diente das zweite Kapitel dieser Ausarbeitung. Die dort vorgestellten Verfahren für den abstrakten Zustandsraum wurden während der Bearbeitung auf den symbolischen Zustandsraum übertragen und dienten somit als Spezifikation für das Programm.

Für den kritischen Bereich der Umwandlung der Eingabesprache in ein BDD-Objekt wurde ein sogenannter "throw-away"-Prototyp (nach [Som01]) entwickelt, bei dem der Parser-Generator JavaCC eingesetzt worden ist. Die Analyse des Prototypen hat gezeigt, dass die Einbettung von Code zur weiteren Verarbeitung in die Grammatik ein Nachteil ist. Dies liegt darin begründet, dass durch die Vermischung von Code und Grammatik dieser Code mit in die durch JavaCC generierten Dateien einfliesst und somit nicht mehr davon ausgegangen werden kann, dass diese Klassen fehlerfrei sind. Als weiterer Nachteil hat sich herausgestellt, dass dieser Ansatz nicht die Mächtigkeit von SableCC mit dem erweiterten Visitor-Entwurfmuster (siehe dazu Kapitel 5.2.1) hat.

5.2 Architekturentscheidungen

In diesem Abschnitt werden die drei wichtigsten Entscheidungen vorgestellt, die die Architektur beeinflußt haben, welche im weiteren Verlauf dieses Kapitels noch genauer beschrieben wird. Dabei sind die wichtigsten Entscheidungen:

- Wahl des Parser-Generators für die Eingabesprache
- Wahl des passenden BDD-Paketes
- Wahl der Bibliothek für die grafische Benutzeroberfläche

5.2.1 Der Parser-Generator SableCC

Die Umwandlung eines Strings in einen BDD wird mittels eines Parsers realisiert, welcher mit dem Parser-Generator SableCC entwickelt wurde. SableCC erzeugt aus einer Grammatik, welche in der Extended-Backus-Naur-Form (EBNF) definiert wird, einen Parser. Genauere Informationen zur Parser-Erzeugung befinden sich in "Modern Compiler Implementation in Java" ([App97]).

Im Gegensatz zu dem bekannteren Parser-Generator JavaCC ist bei SableCC eine Einbettung von Quellcode in die Grammatik nicht möglich. Stattdessen ist in SableCC eine erweiterte Variante des Visitor-Entwurfsmusters ([GHJV96]) implementiert worden, welche in [Gag98] vorgestellt wurde. Das Visitor-Entwurfsmuster beschreibt allgemein eine Architektur für den Lauf durch einen Baum. SableCC generiert eine Klasse `DepthFirstAdapter`, welche eine Tiefensuche auf den Knoten des abstrakten Syntaxbaumes implementiert. Diese Klasse bietet für jeden Knotentyp Hook-In Methoden ([GHJV96]) für den Eintritt in und den Austritt aus diesen Knoten, siehe auch Abbildung 5.2.

Die Trennung von Code für die Weiterverarbeitung und Grammatik zusammen mit der eleganten Ausführung des Visitor-Entwurfsmusters sind die Vorteile der Nutzung von SableCC gegenüber JavaCC.

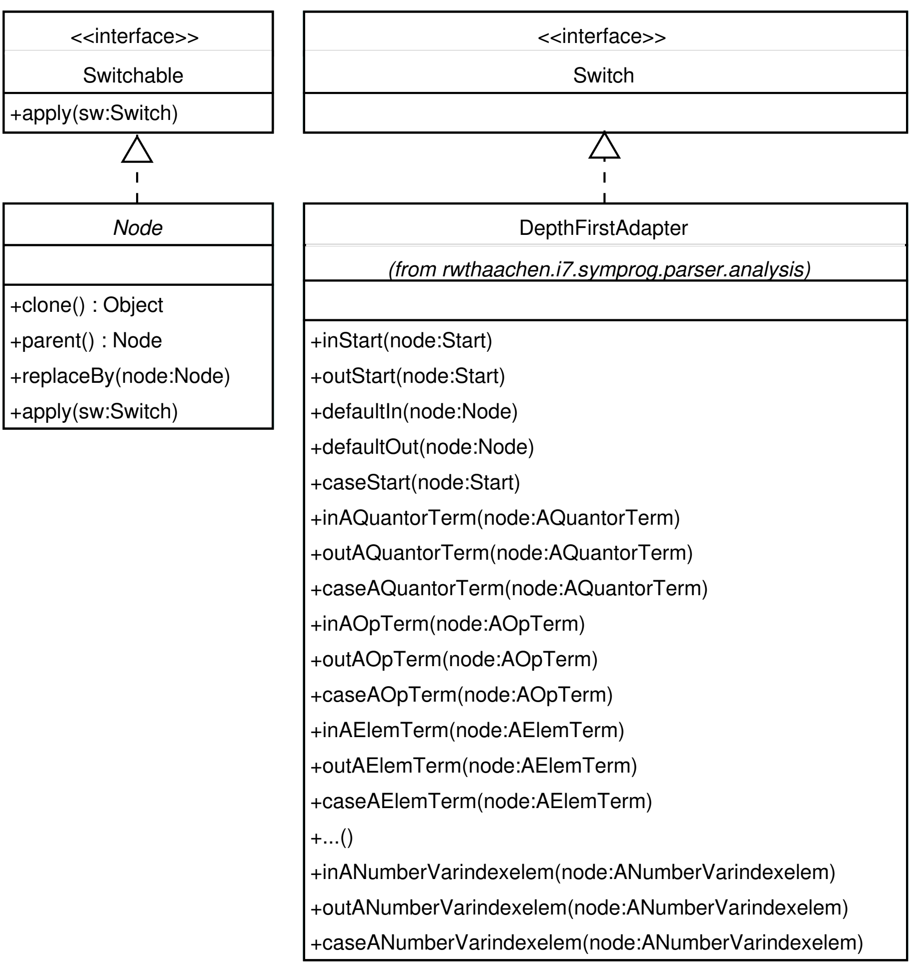


Abbildung 5.2: Knoten des Syntaxbaumes und das zugehörige Visitor-Entwurfsmuster

`Node`

Von dieser Klasse sind alle im Syntaxbaum vorhandenen Knoten abgeleitet worden.

`clone`	Klont den Knoten mitsamt seinen Kindern.
`parent`	Liefert den Vater des Knotens.
`replaceBy`	Diese Routine ersetzt den aktuellen Knoten durch den übergebenen.
`apply`	Die Methode `apply` erlaubt die Nutzung des Visitor-Entwurfsmusters. Sie ruft die Methode `case{Knotentyp}` von dem Knoten des übergebenen Syntaxbaumes auf.

`DepthFirstAdapter`

Die Klasse `DepthFirstAdapter` implementiert das erweiterte Visitor-Entwurfsmuster nach [Gag98]. Durch Aufrufen der Methode `apply` eines Syntaxbaums wird die Methode `case{Typ des Wurzelknotens}` für den Wurzelknoten des Baums aufgerufen und damit der Lauf durch den Baum gestartet.

`case{Knotentyp}`	Zuerst wird die Hook-In Methode `in{Knotentyp}` aufgerufen, dann für jeden Kindknoten die apply-Methode und zuletzt wird die Methode `out{Knotentyp}` ausgeführt.
`in{Knotentyp}`	Die Methode ruft standardmäßig die Routine `defaultIn` auf, solange nicht in einer abgeleiteten Klasse das Verhalten für den Knotentyp ersetzt worden ist.
`out{Knotentyp}`	Analog zu `in{Knotentyp}` wird diese Methode beim Verlassen des Knotens aufgerufen.
`defaultIn`	Diese Methode ist standardmäßig leer. In einer abgeleiteten Klasse kann das Standardverhalten definiert werden, wenn ein Knoten aufgesucht wird.
`defaultOut`	Analog für das Verlassens eines Knotens.

5.2.2 Das BuDDy-Paket

Da BDDs die zentrale Datenstruktur in diesem Programm sind, ist die Auswahl des zu verwendenden BDD-Paketes besonders wichtig. Die Suche nach einem geeigneten Paket, welches in Java implementiert ist, lieferte nur einen einzigen Treffer: JADE (JAVA DEcision diagram package).

- Schwerpunkt liegt auf der Visualisierung von BDDs, daher ansonsten nur rudimentärer Funktionsumfang.
- Weiterentwicklung von Projekt ROBDD, welches eine Studienarbeit war.
- Mittels JavaDoc dokumentiert.
- http://www.informatik.uni-bremen.de/agra/public/software/

Eine Suche nach C oder C++ Implementierungen lieferte dagegen gleich mehrere in Frage kommende Pakete:

- BuDDy - A Binary Decision Diagram Package
  - Von Jørn Lind-Nielsen
  - Wird bei verschiedenen Projekten eingesetzt.
  - Wird kontinuierlich weiterentwickelt.
  - Sehr gut und ausführlich dokumentiert (nicht nur Header-Files)
  - Das Paket enthält Beispiele.
  - http://www.it-c.dk/research/buddy/
- CU Decision Diagram Package (CUDD)
  - Von Fabio Somenzi

  - Neben BDDs werden auch ADDs[1] und ZDDs[2] unterstützt.
  - Beinhaltet auch eine ausführliche Dokumentation.
  - http://vlsi.colorado.edu/ fabio/

- The BDD Library (bbdlib)
  - Leider nur schlecht bis gar nicht dokumentiert.
  - http://www-2.cs.cmu.edu/afs/cs/project/modck/pub/www/bdd.html

Die Wahl fiel dann letztendlich auf das BuDDy-Paket von Jørn Lind-Nielsen. Einerseits lag es daran, dass es subjektiv auf mich den besten Eindruck machte, aber andererseits auch daran, dass es bereits mit Erfolg in den verschiedensten Projekten eingesetzt wird. Nach [LNAB+98] wurde BuDDy erfolgreich eingesetzt, um Systeme von endlichen Zustandsmaschinen mit bis zu 1400 nebenläufig arbeitenden Maschinen zu verifizieren.

Da somit ein C/C++-Paket für die BDDs zum Einsatz kommt, stellte sich die Frage, wie das Programm realisiert werden kann. Zur Auswahl standen die Möglichkeiten, den algorithmischen Part in C/C++ zu realisieren und eine grafische Benutzeroberfläche in Java zu programmieren oder das gesamte Programm in Java zu entwickeln. Um das BuDDy-Paket trotzdem benutzen zu können, stellt Sun mit Java Native Interface (JNI) eine Technik zur Verfügung, native Code aus Java-Programmen zu benutzen. Weitere Informationen zu JNI können beispielsweise in [Lia99] nachgelesen werden.

Da bei der ersten Variante das Problem des Datenaustausches zwischen C/C++-Programm und dem Java-Programm für die Benutzeroberfläche gegeben wäre, fiel die Wahl auf die zweite Variante, wo das BuDDy-Paket mittels JNI angebunden wird. Ein weiterer Vorteil dieser Variante ist, dass so der algorithmische Part des Programmes mittels des Testrahmenwerkes JUnit geprüft werden kann, siehe dazu Kapitel 5.4.

5.2.3 SWT - Standard Widget Toolkit

Bei der Auswahl der Bibliothek für die grafische Benutzerschnittstelle des Programmes fiel die Wahl auf SWT (Standard Widget Toolkit) von OTI[3]/IBM. Der Hauptvorteil von SWT im Vergleich zu AWT/Swing ist, dass bei SWT die vom jeweiligen Betriebssystem zur Verfügung gestellten grafischen Benutzerelemente zum Einsatz kommen. Daraus resultieren dann die weiteren Vorteile, die darin bestehen, dass SWT sehr schnell läuft und die Anwendung nicht wie ein „typisches“ Java-Programm aussieht, sondern so wirkt, als wäre es speziell für dieses Betriebssystem entwickelt worden.

Allerdings sollte auch der größte Nachteil von SWT nicht verschwiegen werden. Bedingt durch den Einsatz der native Widgets benötigt SWT für jedes Betriebssystem ein angepasstes JAR-Archiv mit dazu passender Bibliothek, um die SWT-API abzubilden. Von OTI gibt es diese für alle gängigen Systeme wie Windows, Linux, Mac OSX, Solaris. Dieser Nachteil wirkt im konkreten Fall nicht so schwer, da durch den Einsatz des BuDDy-Paketes bereits für jedes Betriebssystem eine eigene BDD-Bibliothek erstellt werden muss.

[1] Algebraic Decision Diagrams
[2] Zero-suppressed Binary Decision Diagrams
[3] Object Technology International Inc.

5.3 Die Architektur

In diesem Unterkapitel wird die statische Architektur des entstandenen Programmes beschrieben, d.h. welche Pakete beziehungsweise Klassen es gibt und in welcher Beziehung diese zueinander stehen. Dazu wird zuerst die Paketstruktur vorgestellt und anschliessend die Aufgaben und Klassen der einzelnen Pakete.

Abbildung 5.3 zeigt die vier verschiedenen Pakete des Programmes sowie deren Abhängigkeiten.

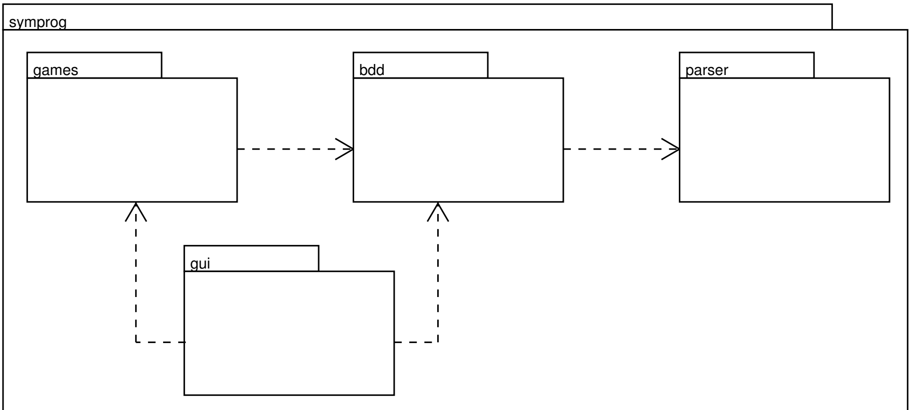


Abbildung 5.3: Pakete des Programms

Die Aufgaben der einzelnen Pakete sind:

bdd Das Paket bdd beinhaltet die zentrale Datenstruktur der BDDs und ist eine Kapselung des BuDDy-Pakets. Da das BuDDy-Paket nur als C bzw. C++-Version erhältlich ist, wurde es mittels JNI angebunden. Dies ist eine Technik, um aus Java-Programmen auf native Code zuzugreifen, der in Bibliotheken passend für das jeweilige Betriebssystem abgelegt ist. Damit ist schon der größte Nachteil von JNI genannt, durch den Einsatz von JNI geht die Plattformunabhängigkeit von Java verloren, da der native Code passend für das eingesetzte Betriebssystem kompiliert werden muss.

parser Das Paket parser realisiert einen Parser, der die in Kapitel 4.1.1 vorgestellte Grammatik in einen BDD umwandelt. Dieser Parser ist mit SableCC erstellt worden. Nachdem der Parser den String in einen abstrakten Syntaxbaum umgewandelt hat, modifizieren einige Visitoren zuerst den Baum, um letztendlich daraus den dazugehörenden BDD zu erzeugen.

games Das Paket games stellt soetwas wie das Aplication Programming Interface dar, da es den algorithmischen Teil der implementierten Spiele beinhaltet. An dieser Stelle ist darauf geachtet worden, die grafische Benutzeroberfläche und Anwendungslogik strikt zu trennen. Die grafische Benutzeroberfläche befindet sich ausschließlich im Paket gui.

gui Wie der Name schon vermuten lässt, befindet sich in diesem Paket alles, was zur grafischen Benutzeroberfläche des Programmes gehört.

5.3.1 Das Paket bdd: Kapselung der BDDs

Die vom BuDDy-Paket zur Verfügung gestellten Routinen sind auf vier Klassen des Pakets `bdd` verteilt worden, wie Abbildung 5.4 zeigt:

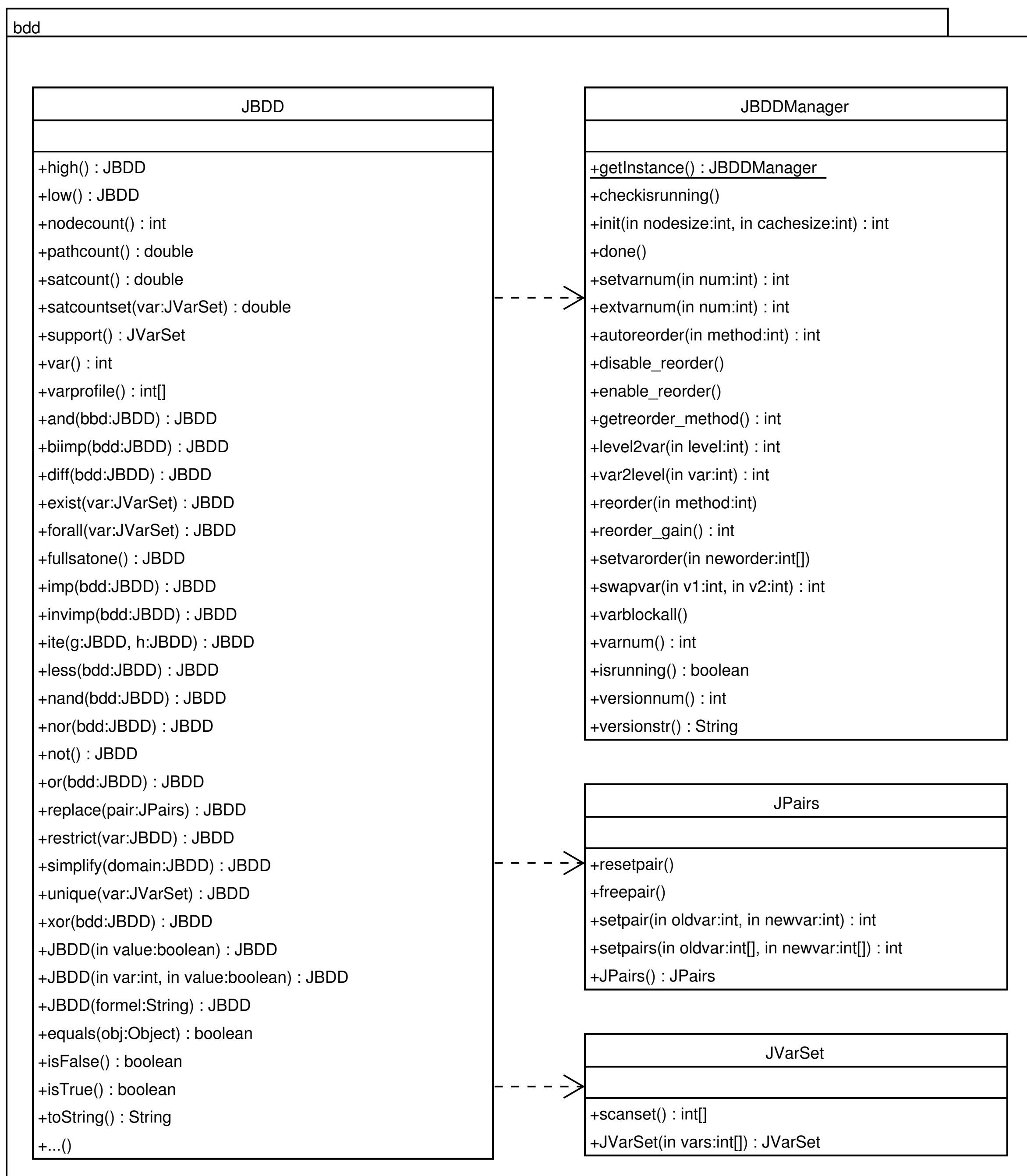


Abbildung 5.4: Paket bdd

Die in diesem Paket enthaltenen Klassen stellen bis auf wenige Ausnahmen nur die Funktionalität des BuDDy-Paketes zur Verfügung. Da die einzelnen Klassen teils sehr viele Methoden anbieten, sei an dieser Stelle auf die Dokumentation des BuDDy-Pakets verwiesen, da sich

an den Parametern der Routinen nichts verändert hat. Lediglich bei Namensgebung ist auf das Präfix bdd_ verzichtet worden.

Die Aufgaben der einzelnen Klassen sind:

JBDD Ein Objekt dieser Klasse repräsentiert einen BDD. Die enthaltenen Routinen lassen sich den folgenden Kategorien zuordnen:

- unäre oder binäre BDD-Operationen
- Durchlaufen des BDD-Baumes
- Variablenquantifizierung bzw. -ersetzung
- Statistik zum BDD (wie Knoten- oder Pfadanzahl)
- Lösungen des BDDs (Anzahl erfüllender Belegungen, konkrete erfüllende Belegung)

JBDDManager Diese Klasse beinhaltet alle Routinen, die BDD-übergreifend sind. Damit sichergestellt ist, dass nur eine Instanz dieser Klasse gebildet werden kann, wurde das Singleton-Entwurfsmuster ([GHJV96]) benutzt. Die Klasse bietet Routinen zu den Themen:

- globale Statistik
- Variablenordung
- Einstellungen zur Variablenanzahl, Knotenanzahl, Cacheanzahl

JPairs Die Klasse `JPairs` repräsentiert eine verkettete Liste von Variablenpaaren zwecks Ersetzung. Diese werden für die Routine `replace` der Klasse `JBDD` benötigt.

JVarSet Die Klasse `JVarSet` kapselt eine Variablenmenge, um sie beispielsweise in einem `JBDD`-Objekt zu quantifizieren.

Die Klasse `JBDD` ist noch um die Möglichkeit erweitert worden, einen Visitor auf den BDD-Baum anzuwenden (wie in [GHJV96] vorgestellt). Dies ist gemacht worden, da das BuDDy-Paket keine Möglichkeit vorsieht, alle erfüllenden Belegungen eines BDDs zu ermitteln. Das Ergebnis davon ist in Abbildung 5.5 dargestellt.

`JBDD`

Die bestehende Klasse ist um eine weitere Methode erweitert worden.

`applyvisitor`	Nimmt einen Visitor an, der dann auf den BDD angewendet wird

`Switch`

Das Interface definiert einen Visitor für einen BDD.

`caseJBDD`	Diese Routine wird für jeden Knoten des BDD-Baumes aufgerufen, anschliessend wird der Visitor für die Kinder des Knotens aufgerufen.

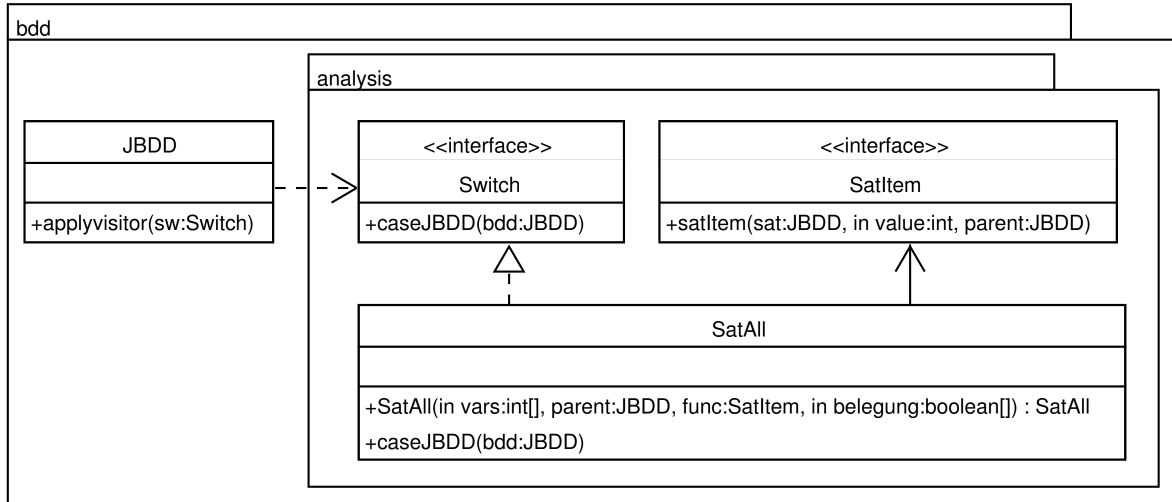


Abbildung 5.5: Paket bdd.analysis

<u>`SatItem`</u>
Das Interface definiert die Möglichkeit, eine gefundene, erfüllende Belegung eines BDDs weiterzuverarbeiten.

`satItem`	Diese Methode wird von der Klasse `SatAll` aufgerufen, wenn eine neue erfüllende Belegung des zu untersuchenden BDDs gefunden wurde.

<u>`SatAll`</u>
Der Visitor SatAll findet alle erfüllenden Belegungen eines BDDs.

5.3.2 Das Paket games: Aplication Programming Interface (API)

Abbildung 5.6 zeigt das Klassendiagramm des Paketes `games`. Dieses stellt die API des Programmes dar. Im folgenden werden alle Methoden der aufgeführten Interfaces und Klassen der API erklärt.

<u>`AbstractGame`</u>
Die abstrakte Klasse AbstractGame ist die Basis für alle implementierten Spiele.

`solve`	Diese Methode muss von jeder abgeleiteten Klasse implementiert werden. Ein Aufruf dieser Methode bestimmt die Gewinnbereiche und -strategien der beiden Spieler.
`attraktor`	Diese Methode berechnet den Attraktor zu einer gegebenen Menge.
`attraktorplus`	Diese Methode berechnet den Attraktor+ zu einer gegebenen Menge.
`recur`	Diese Methode berechnet die Recur-Menge zu einer gegebenen Menge.

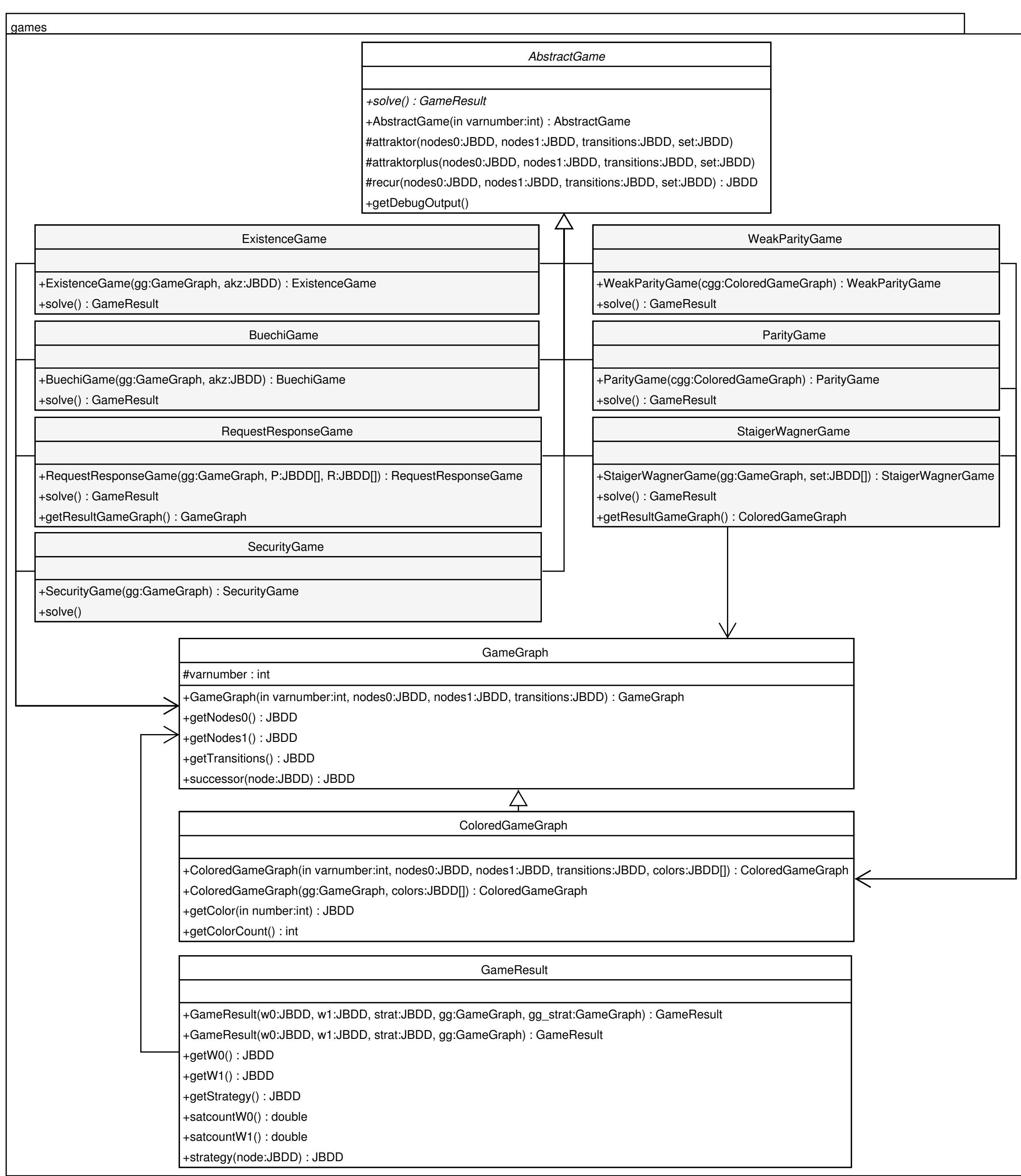


Abbildung 5.6: Paket games

`getDebugOutput`	Diese Routine wird von der GUI verwendet, um die Informationen für die Zwischenergebnis-Seite abzurufen.

Alle sieben davon abgeleiteten Spiele implementieren nur die `solve`-Methode der Klasse `AbstractGame`, so dass eine genauere Betrachtung dieser Klassen unnötig ist. Lediglich für die Staiger-Wagner-Spiele als auch für die Request-Response-Spiele wird noch eine weitere Methode implementiert:

`getResultGameGraph`	liefert den Spielgraphen des reduzierten Spieles

`GameGraph`

Diese Klasse repräsentiert einen Spielgraphen für die hier betrachteten Spiele.

`getNodesX`	Diese Methode liefert die Knoten des Spielers X zurück.
`getTransitions`	Liefert die Transitionsformel des Spielgraphens zurück.
`successor`	Diese Methode berechnet zu einem gegebenen Knoten die Nachfolger gemäß der Transitionsformel.

`ColoredGameGraph`

`ColoredGameGraph` ist von `GameGraph` abgeleitet und stellt den Spezialfall des gefärbten Spielgraphens dar. Als neue Methoden sind hinzugekommen:

`getColor`	Diese Methode liefert die Farbformel zu dem übergebenen Wert zurück.
`getColorCount`	Liefert die Anzahl der definierten Farben zurück.

`GameResult`

Diese Klasse repräsentiert die Lösung eines Spieles.

`getWX`	Diese Methode liefert den Gewinnbereich des Spielers X zurück.
`getStategy`	Liefert die Gewinnstrategien für beide Spieler zurück.
`getSatCountWX`	Liefert die Größe des Gewinnbereiches für Spieler zurück.
`strategy`	Liefert für den übergebenen Knoten die Nachfolgerknoten gemäß der berechneten Gewinnstrategie.

5.3.3 Das Paket gui: Das grafische Benutzerinterface

Dieses Paket beinhaltet die grafische Benutzeroberfläche des Programmes. Der Focus bei der Betrachtung dieses Paketes liegt auf der Einbindung der Spiele in die Oberfläche. Diese wird über die zwei Unterpakete `accept` und `game` realisiert, dargestellt in den Abbildungen 5.7 und 5.8.

Da sich die Spiele in der Akzeptierbedingung unterscheiden, muss die grafische Benutzeroberfläche in Abhängigkeit von dem ausgewählten Spiel angepasst werden. Da mehrere Spiele den selben Typ von Akzeptierbedingung haben, wie zum Beispiel Garantie- und Büchi-Spiel, sind die GUI-Elemente der Akzeptierbedingungen in eigene Klassen ausgelagert worden, damit diese für die unterschiedlichen Spiele wiederbenutzt werden können. Entstanden ist dabei ein Interface, welches von den Klassen für die Akzeptierbedingungen implementiert wird, wie die Abbildung 5.7 zeigt.

`Acceptance`

Dieses Interface definiert, welche Methoden eine Klasse implementieren muss, damit sie die Möglichkeit bietet, eine Akzeptierbedingung in der Benutzeroberfläche zur Verfügung zu stellen.

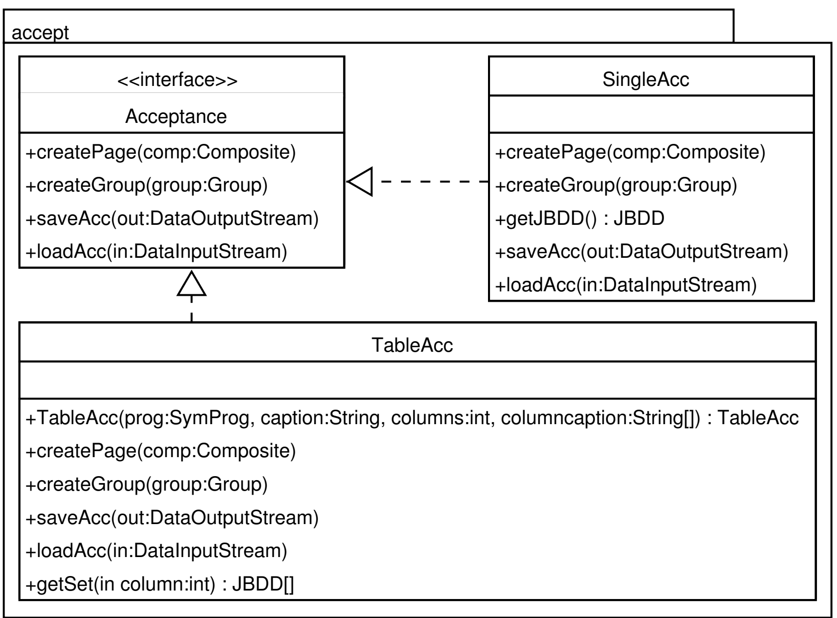


Abbildung 5.7: Paket gui.accept

`createPage`	Wenn diese Routine ausgeführt wird, wird eine Seite in der GUI erzeugt, um die Akzeptierbedingung eingeben zu können.
`createGroup`	Ein Aufruf diese Methode soll die GUI-Elemente anlegen, um auf der Parser-Seite der GUI die Ergebnisse des Parsers angewendet auf die Akzeptierbedingung darstellen zu können.
`saveAcc`	Diese Methode speichert die Akzeptierbedingung in dem übergebenen DataOutputStream.
`loadAcc`	Mit dieser Methode kann die Akzeptierbedingung aus dem DataInputStream rekonstruiert werden.

`SingleAcc`

Diese Klasse repräsentiert eine Akzeptierbedingung, die nur aus einer einzelnen booleschen Formel besteht, wie dies bei Garantie- bzw. Büchi-Spielen der Fall ist. Neben den vom Interface `Acceptance` geforderten Methoden hat die Klasse noch die folgende Routine:

`getJBDD`	Liefert ein `JBDD`-Objekt zurück, der der eingebenen Formel der Akzeptierbedingung entspricht.

`TableAcc`

Mit dieser Klasse sind alle die Akzeptierbedingungen abgedeckt, die sich in tabellarischer Form niederschreiben lassen. Dies ist zum Beispiel bei gefärbten Spielgraphen der Fall, die bei schwachen Paritätsspielen und Paritätsspielen zum Einsatz kommen. Um an die eingegebenen Daten zu kommen, ist die folgende Routine implementiert worden:

`getSet`	Diese Methode liefert die eingebenen Formeln der übergebenen Spalte als `JBDD`-Array zurück.

Für jedes implementierte Spiel gibt es eine eigene Klasse für die GUI. Analog zum `Abstract-`

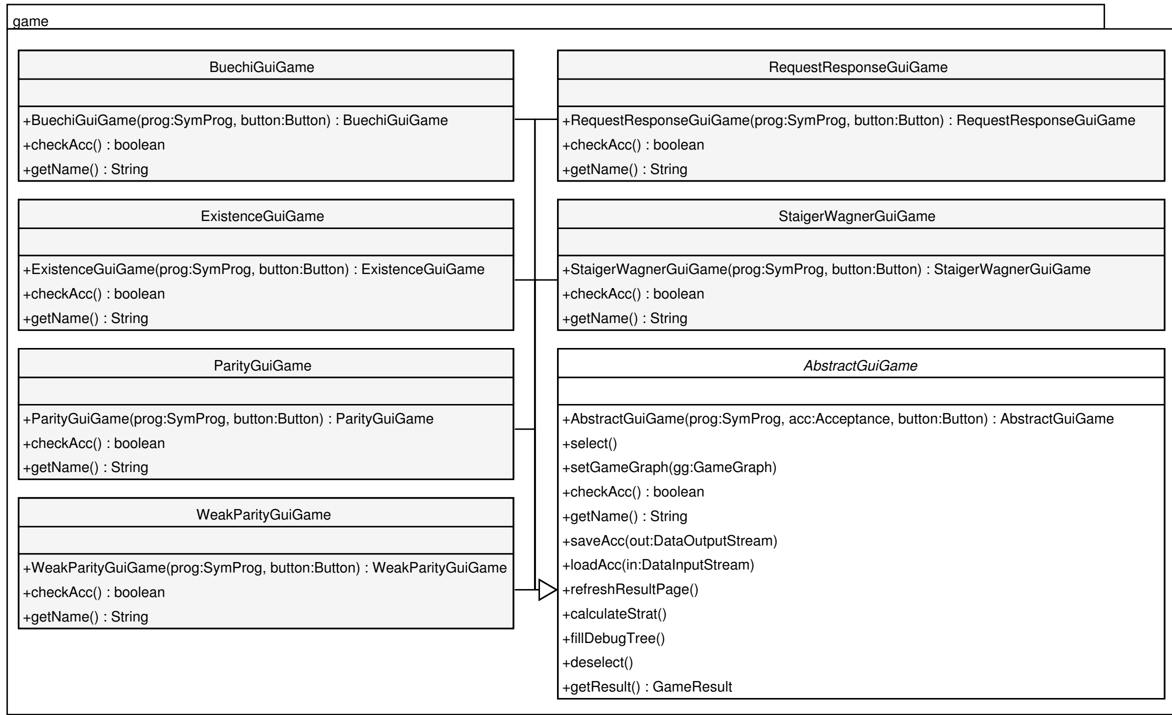


Abbildung 5.8: Paket gui.game

`Game` gibt es auch für die GUI-Klassen einen Ausgangspunkt. Dieser ist die Klasse `AbstractGuiGame`, von der alle Spieleklassen für die GUI abgeleitet werden müssen, wie auch Abbildung 5.8 zeigt.

<u>`AbstractGuiGame`</u>
Ziel dieser Klasse war, die Logik für die GUI soweit wie möglich in diese abstrakte Klasse zu implementieren, damit sich die davon abgeleiteten Klassen sehr einfach implementieren lassen. Das Ergebnis ist, dass die abgeleiteten Klassen nur zwei sehr einfach aufgebautete abstrakte Methoden implementieren müssen.

`select`	Diese Routine wählt das Spiel aus.
`deselect`	Diese Routine wählt ein Spiel wieder ab.
`setGameGraph`	Mittels dieser Routine kann der Spielgraph zugewiesen werden.
`checkAcc`	Diese Methode muss jede abgeleitete Klasse implementieren. Ein Aufruf dieser Methode soll dazu führen, dass das Spiel erzeugt wird und die Akzeptierbedingung überprüft wird.
`getName`	Liefert den Namen des Spieles zurück. Diese Information wird für zwei Punkte benötigt. Einerseits dient diese Info dazu, den Eintrag in die ComboBoxen für die Spielauswahl zu erstellen, und andererseits dient es zur Identifizierung des Spieles beim Speichern/Laden von Spielen. Deshalb ist es wichtig, dass jede abgeleitete Klasse einen eindeutigen String als Identifier zurück gibt.
`saveAcc / loadAcc`	Speichert bzw. lädt die Akzeptierbedingung des Spieles. Die eigentlich Arbeit wird an das `Acceptance`-Objekt des Spieles delegiert.

`refreshResultPage`	Diese Routine wird vom Hauptprogramm aufgerufen, um die Ergebnis-Seite zu aktualisieren.
`fillDebugTree`	Diese Routine aktualisiert die Zwischenergebnis-Seite.
`calculateStrat`	Liefert die Nachfolger des Knotens gemäß der Gewinnstrategie. Diese Aufgabe wird delegiert an das `GameResult`-Objekt des Spieles.
`getResult`	Nachdem die Routine `checkAcc` aufgerufen wurde, kann mittels `getResult` das Ergebnis des Spieles abgefragt werden.

Alle sechs hiervon abgeleiteten Klassen implementieren nur die beiden abstrakten Methoden `checkAcc` und `getName`.

5.3.4 Das Paket parser: String → BDD

Die Aufgabe dieses Paketes ist es, aus einem String, der eine boolesche Formel gemäß der Grammatik von Kapitel 4.1.1 enthält, ein `JBDD`-Objekt zu erzeugen. Diese Aufgabe ist mittels des Parser-Generators SableCC gelöst worden, wie in Kapitel 5.2.1 geschildert. Der Ablauf der Umwandlung einer booleschen Formel, gegeben als String, in ein `JBDD`-Objekt ist in Abbildung 5.9 dargestellt. Aus der Abbildung wird auch die zentrale Rolle des abstrakten Syntaxbaumes für diesen Vorgang deutlich.

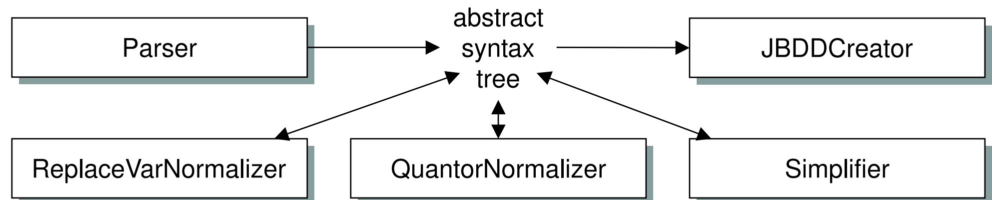


Abbildung 5.9: Umwandlung einer boolschen Formel in ein JBDD-Objekt

Die Umwandlung geschieht in den folgenden Schritten:

1. Der Parser erzeugt aus dem String, der die boolesche Formel beinhaltet, den abstrakten Syntaxbaum. Ausserdem wird, wie in Kapitel 5.2.1 beschrieben, die zur weiteren Bearbeitung des Baumes nötige Infrastruktur zur Verfügung gestellt, so dass Visitoren auf den Baum angewendet werden können.
2. Der Visitor `ReplaceVarNormalizer` ersetzt Variablen durch konkrete Werte. Er wird dazu benutzt, externe Parameter zu ersetzen. Ausserdem kommt er beim `QuantorNormalizer` zum Einsatz, um dort die Variable, die durch Quantor spezifiziert wurde, geeignet zu ersetzen. Im konkreten wird x[i] in x[3] umgewandelt, wenn das Paar (i,3) im Konstruktor übergeben wurde.
3. Wenn Quantoren in der booleschen Formel vorhanden sind, ersetzt der Visitor `QuantorNormalizer` die Quantoren geeignet, damit in den weiteren Schritten aus dem Baum ein `JBDD`-Objekt erzeugt werden kann. Somit stellt er neben dem `JBDDCreator` das Herzstück dieses Paketes dar.

4. In diesem Schritt sorgt der Visitor `Simplifier` dafür, dass in den Variablenindices keine arithmetischen Formeln mehr vorkommen. Durch die vorangegangenen Schritte ist sichergestellt, dass in den Variablenindices nur noch arithmetische Ausdrücke vorkommen. Somit kann der `Simplifier` einfach diese arithmetischen Formeln ausrechnen und ersetzen. So wird aus x[3+4] der Ausdruck x[7].
5. Durch die bisherigen Schritte ist sichergestellt, dass die boolesche Formel keine Quantoren mehr enthält und die Variablenindices nur aus einfachen Zahlenwerten bestehen. Somit wird es möglich, dass der Visitor `JBDDCreator` den Baum in ein `JBDD`-Objekt umwandeln kann, welches dem ursprünglich eingegebenen String entspricht.

5.3.5 Weitere Spiele hinzufügen

Das gesamte Programm ist so entwickelt worden, dass darauf geachtet wurde, dass das Einfügen weiterer Spiele sich möglichst einfach gestaltet. Dies ist schon aus eigenem Interesse geschehen, da alle hier implementierten Spiele der Reihe nach umgesetzt worden sind.

Um weitere Spiele dem Programm hinzuzufügen, sind mehrere Schritte notwendig:

1. Für den algorithmischen Part des Spieles wird eine Klasse im Paket `games` von `AbstractGame` abgeleitet. Im wesentlichen muss nur noch die Methode `solve` implementiert werden. Dafür bietet die abstrakte Klasse `AbstractGame` einige Algorithmen, wie Attraktor, Attraktor+ und Recur, als Unterstützung für das Implementieren der Methode `solve`, siehe auch Kapitel 5.3.2. Wird das Spiel mittels einer Reduktion auf ein bereits implementiertes Spiel gelöst, sollte auch eine Methode `getResultGameGraph` implementiert werden, die den Spielgraphen des reduzierten Spieles zurückgibt.
2. Bleibt nur noch der Teil übrig, der Voraussetzung dafür ist, dass das Spiel in die grafische Benutzeroberfläche eingehängt werden kann. Dazu muss für das Spiel eine Klasse von `AbstractGuiGame` im Paket `gui.game` abgeleitet werden, siehe auch 5.3.3. Für den Konstruktor muss eine passende Klasse gefunden werden, die das `Acceptance`-Interface implementiert. In der Regel sollte ein Objekt der Klasse `TableAcc` passend sein. Ansonsten müssen nur noch die beiden abstrakten Methoden `getName` und `checkAcc` implementiert werden. Für `getName` ist wichtig, dass ein eindeutiger String zurückgegeben wird, der sonst von keinem anderen Spiel zurückgegeben wird. Denn ansonsten gibt es beim Laden des Spieles Probleme. Bei einem Aufruf von `checkAcc` muss eine Instanz des Spieles angelegt werden.
3. Der letzte Schritt besteht darin, das neu erstellte Spiel in der grafischen Benutzeroberfläche anzumelden. Dafür muss die Methode `registerGames` in der Datei `SymProg.java` angepasst werden, die im Algorithmus 5.1 dargestellt ist.

   Dazu reicht es aus, einfach eine weitere Zeile hinzufügen, die gleich aufgebaut ist, nur dass der Klassenname des neu erstellen Spieles eingefügt wird.

5.4 Qualitätssicherung

Um die Qualität des entstandenen Programmes sicherzustellen, werden automatisierte Tests mittels des Testrahmenwerkes JUnit benutzt. Dieses Rahmenwerk ist von Erich Gamma und

```
public void registerGames(){
    games = new Vector();
    games.add(new ExistenceGuiGame(this));
    games.add(new SecurityGuiGame(this));
    games.add(new WeakParityGuiGame(this));                5
    games.add(new StaigerWagnerGuiGame(this));
    games.add(new RequestResponseGuiGame(this));
    games.add(new BuechiGuiGame(this));
    games.add(new ParityGuiGame(this));
}                                                          10
```

Algorithmus 5.1: registerGames

Kent Beck entwickelt worden (siehe [BG98]), um auf einfache Weise Regressionstests durchführen zu können. Darunter ist zu verstehen, dass bei Veränderungen am Programm diese Tests erneut ausgeführt werden, um sicherzustellen, dass die Änderungen am Programm keine negativen Auswirkungen haben. Somit ist der Aufwand für Tests neuer Programmversionen minimal, da die bereits existierenden, automatisierten Tests wieder ausgeführt werden können.

Erst automatisierte Tests ermöglichen Refactoring-Maßnahmen, da nur so sichergestellt werden kann, dass das beobachtbare Verhalten unverändert bleibt. Davon ist während der Entwicklung des Programmes reichlich Gebrauch gemacht worden.

Um sicherzustellen, dass die Testfälle gemäß der Spezifikation und nicht aufgrund des Quellcodes geschrieben werden, wurden weitestgehend zuerst die Testfälle entwickelt und erst anschliessend die dazugehörenden Klassen. Dies entspricht der Philosophie des "Extreme Programmings", nachzulesen in [Bec98, Bec00]. Gemäß dieser Testphilosophie gilt:

> Eine Eigenschaft ohne automatischen Test funktioniert nicht; eine Eigenschaft mit automatischem Test könnte funktionieren.

5.4.1 Testziele

Die Tests wurden gemäß [Lic00] in Test Groups, Test Suites und Test Cases strukturiert. Es wurden zwei verschiedene Gruppen von Tests mit unterschiedlichen Zielen entwickelt:

1. Test der BDDs
   a) des BuDDy-Paketes
   b) des Parsers zur Erzeugung der BDDs
2. Test der entwickelten Spiele

Beide Testgruppen wurden um weitere Testfälle erweitert, um eine möglichst hohe Zweigüberdeckung zu erreichen.

Für die Testfälle des BuDDy-Paketes lag die Dokumentation des Paketes als Spezifikation zugrunde. Die dabei entstandenen Tests sind gemäß [Som01] Blackbox-Tests, da die BDD-Klassen fast ausschliesslich aus Aufrufen des C++-Codes mittels JNI bestehen.

Die Grundlage für die Tests des Parsers bildete die Grammatik der Eingabesprache in Kapitel 4.1.1. Im Gegensatz zu den Tests des BuDDy-Paketes sind diese Test Whitebox-Tests.

Die Tests der BDDs sind mit äußerster Sorgfalt entwickelt worden, da die BDDs zentrale Datenstruktur für das Programm sind und somit sichergestellt sein muss, dass diese fehlerfrei arbeiten.

Für jedes implementierte Spiel aus dem dritten Kapitel existiert eine eigene Test Suite. Die darin enthaltenen Testfälle sind auf Grundlage der Kapitel 2.1 und 3 entstanden.

5.4.2 Überprüfung des Deckungsgrades

Zur Überprüfung, aber auch zur Verbesserung des Überdeckungsgrades der Tests wurde JCover 2 der Firma Man Machine Systems eingesetzt. Dieses Werkzeug wurde gewählt, da es das einzige ist, welches eine kostenlose Evaluierung zulässt und ausserdem noch eine komfortable Messung der Zweigüberdeckung bietet.

Gesamtanzahl der Dateien	29
Ausgeführte Dateien	26
Überdeckungsgrad der Dateien	89,66%
Gesamtanzahl der Klassen	26
Ausgeführte Klassen	26
Überdeckungsgrad der Klassen	100%
Gesamtanzahl der Methoden	129
Ausgeführte Methoden	129
Überdeckungsgrad der Methoden	100%
Gesamtanzahl der Zweige	234
Ausgeführte Zweige	230
Zweigüberdeckung	98,29%
Gesamtanzahl der Anweisungen	687
Ausgeführte Anweisungen	685
Anweisungsüberdeckung	99,71%

Tabelle 5.1: Überdeckungsgrade der automatisierten Tests

Wie Tabelle 5.1 zeigt, liegt die Zweigüberdeckung der eingesetzten Tests mit 98,29% nahe an den angestrebten 100%. Außerdem zeigt Abbildung 5.10, dass die Zweigüberdeckung in allen Methoden gleichermaßen gut ist. Diese Ergebnisse wurden erreicht, da für die Bestimmung der Überdeckungen das GUI-Paket für die grafische Oberfläche des Programm sowie die von SableCC generierten Klassen des Parser nicht in die Überdeckungsgradmessung miteinbezogen wurden.

Alle nicht überdeckten Teile des Quellcodes wurden ausgiebig inspiziert. Zumeist handelt es sich dabei um das Abfangen von Ausnahmen, deren Auftreten ein interner Fehler wäre. In diesen Fällen wird dann eine Instanz der Klasse `RunTimeException` geworfen.

Die Bestimmung des Überdeckungsgrades hat einige Passagen von sogenannten "Dead-Codes[4]"

[4] Code der nie ausgeführt wird

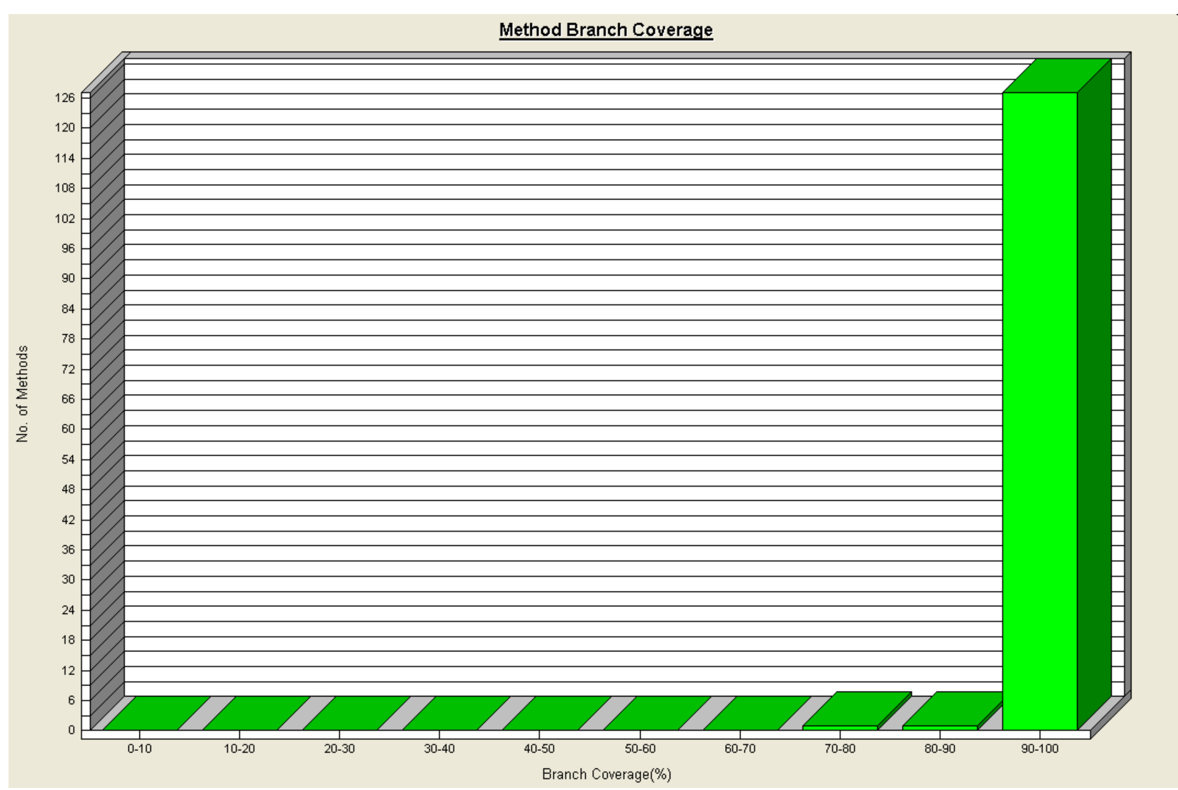


Abbildung 5.10: Überdeckungsgrade der automatisierten Tests pro Methode

aufgezeigt. Desweiteren hat die Analyse Schleifenbedingungen gefunden, die immer `true` oder `false` ergaben und somit überflüssig sind.

Kapitel 6

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurden Algorithmen aus der zustandsbasierten reaktiven Programmtheorie von dem abstrakten auf den symbolischen Zustandsraum übertragen. Dazu wurde als erstes die Attraktor-Konstruktion übertragen. Diese bildete die Ausgangsbasis für die Lösungsverfahren der Existenz- und der schwachen Paritätsspiele. Durch eine Modifikation hat sich daraus auch der Attraktor+ ableiten lassen, welcher zusammen mit der Recur-Konstruktion die Grundlage für die Lösung der Büchi-Spiele bildet. Für die Request-Response Spiele konnte die von Hütten ([Hüt03]) entwickelte Reduktion auf die Büchi-Spiele ins Symbolische überführt werden. Bei den Paritätsspielen ist eine Variante des McNaughtons-Algorithmus von Zielonka ([Zie98]) umgesetzt worden.

Für die Staiger-Wagner Spiele ist die Reduktion auf die schwachen Paritätsspiele nur durch eine Tiefensuche über den Spielgraphen ausgehend von jedem darin enthaltenen Knoten realisiert worden. Somit ist kein Vorteil in der Platz- wie auch Zeitkomplexität zum abstrakten Fall erreicht worden. Von einer algorithmischen Formulierung der Reduktionen von Muller-Spielen mittels LAR-Konstruktion und von Streett-Spielen mittels IAR-Konstruktion jeweils auf Paritätsspiele ist abgesehen worden, da kein Gewinn zu erwarten gewesen wäre. Dies liegt darin begründet, dass beide Reduktionen auf Einzelzustände zurückführen.

Insgesamt hat sich dabei herausgestellt, dass sich die direkten Lösungsverfahren, also die Verfahren für Spiele mit positionalen Gewinnstrategien, einfacher übertragen lassen als die Spielreduktionen. Dies liegt daran, dass bei den Reduktionen oftmals zusätzliche Informationen in dem erweiterten Spielgraphen codiert werden, die sich so nicht ins Symbolische übertragen lassen.

Im Vergleich zum symbolischen Model Checking ist der Stand bei den reaktiven Systemen dadurch gekennzeichnet, dass der Schritt zum symbolischen Zustandsraum in dieser Arbeit analog zum symbolischen Model Checking vollbracht worden ist. Allerdings fehlt noch eine Teilmenge von Gewinnbedingungen, bei denen sich die Gewinnbereiche und -strategien polynomial berechenen lassen. Beim symbolischen Model Checking ist dieser Schritt durch den Übergang zu der temporalen Logik CTL erreicht worden. In der algorithmischen Theorie unendlicher Spiele fehlt eine derartige Spezifikationslogik noch.

Für die Implementierung der Algorithmen wurde eine Eingabesprache konzipiert, die sich vor allem durch die dort eingeführten Existenz- und Universalquantoren auszeichnet. Die

Fallstudie hat jedoch gezeigt, dass diese Quantoren mit Bedacht eingesetzt werden sollten, da, wenn zuviele Quantoren geschachtelt werden, die Zeit für das Erzeugen des BDD-Objektes sehr groß werden kann.

Die Fallstudie hat weiterhin gezeigt, dass sich Zahlwerte, wie die Position eines Aufzuges, nur kompliziert codieren lassen. Da die Eingabesprache nur boolesche Variablen zur Verfügung stellt, müssen Zahlwerte mittels boolescher Variablen codiert werden. Die platzeffizientere Variante besteht in einer binären Codierung, die den Nachteil einer komplexen Transitionsformel nach sich zieht. Alternativ kann auch eine Variante einer unären Codierung benutzt werden, bei der so viele Variablen benutzt werden, wie der Zahlwert maximal annehmen wird. Dann wird ausschließlich die i-te Variable true gesetzt, wenn der Wert i codiert werden soll. Daraus resultiert auch, dass es keine Arithmetik gibt, um beispielsweise einfach ausdrücken zu können, dass sich der Aufzug nach dem Zug eine Etage höher befindet.

Bei der Implementierung ist darauf geachtet worden, dass sich weitere Spiele möglichst einfach hinzufügen lassen. Dies gilt nicht nur für den algorithmischen Part sondern auch für das Einbinden in die Benutzeroberfläche. Für beide Bereiche sind Ausgangspunkte in Form von abstrakten Klassen geschaffen worden, von denen ein neues Spiel abgeleitet werden kann (siehe dazu auch Kapitel 5.3.5). Die Implementierung ermöglicht auch die Kombination von zwei klassischen Gewinnbedingungen. Dabei wird das erste Spiel auf dem spezifizierten Spielgraphen gelöst und anschliessend das zweite Spiel auf einem eingeschränkten Spielgraphen. Dabei wird der Spielgraph für das zweite Spiel auf dem Gewinnbereich des Spielers 0 auf dem ersten Spiel eingeschränkt. Somit lassen sich beispielsweise Sicherheits- und Request-Response-Bedingungen kombinieren.

Ein weiterer Schwerpunkt bei der Implementierung lag bei der Qualitätssicherung. Dazu wurden für den algorithmischen Teil JUnit-Tests eingesetzt.

Neben dem Finden eines Lösungsverfahrens für die beiden fehlenden Spiele könnte eine mögliche Weiterentwicklung der hier erzielten Ergebnisse darin bestehen, die Effizienz des Parsers zu verbessern. Denn die Fallstudie hat gezeigt, dass die Umwandlung der Formeln in `JBDD`-Objekte ein Vielfaches der Zeit benötigt haben, die das anschliessende Lösen des Spieles gebraucht hat. Dies lag vor allem an dem umfangreichen Einsatz an Quantoren in der Transitionsformel. Um dieses Problem zu lösen, existieren drei mögliche Ansätze. Einer besteht darin, für die Spezifikation der Formeln ein Fragment von QBF (Quantifizierte Boolesche Formeln) herauszuarbeiten, um die Anzahl der Quantoren in den Formeln zu reduzieren. Ein weiterer in dem Finden einer geeigneten Variablenanordnung der BDDs, um somit die Platzkomplexität zu reduzieren. Der letzte Ansatz betrifft die Implementierung. Um eine Performancesteigerung zu erreichen, könnte der Parser in C/C++ implementiert werden. Die Anbindung an das Java-Programm würde dann, wie beim BDD-Paket, mittels der Technik „Java Native Interface“ realisiert.

Eine weitere Möglichkeit, die Effizienz der BDD-Erzeugung der Transitionsformel zu verbessern, könnte auch darin bestehen, dass hierarchische Eingabestufen für die Spezifikation der Transitionsformel geschaffen werden. Hierzu wäre eine Erweiterung bestehender Ansätze von Alur, Kannan und Yannakakis [?] bzw. von Benedikt, Godefroid und Reps [?] auf das spieltheoretische Paradigma nötig.

Anhang A

Inhalt der CD

Die CD-ROM enthält die folgenden Daten:

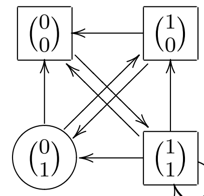
- Die PDF-Version bzw. Postscript-Version dieses Dokumentes
- Die Quelltexte des SymProg-Programms, sowie eine compilierte Version des Programmes.
- Die JavaDoc-Ausgabe der programmierten Klassen.
- Beispiele für das Programm. Diese entsprechenden den Testfällen für das Programm.
- Kompilierte Versionen der BDD-Bibliothek für die Betriebssysteme
  - Microsoft Windows
  - Linux
- Dokumentation und Source des BuDDy-Packages
- Java Development Kit in der Version 1.4.1_01 für die Betriebssysteme
  - Microsoft Windows
  - Linux

Anhang B

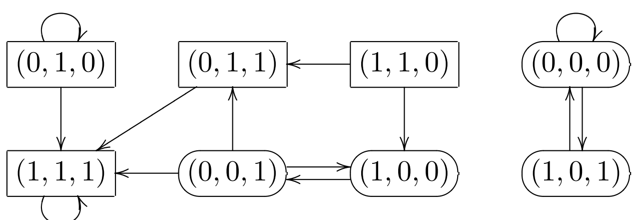
Beispiele auf der CD

Im folgenden werden die Spielgraphen der mitgelieferten Beispiele dargestellt, damit die Ergebnisse der Spiele einfacher nachvollziehbar sind.

- `existence.game`:

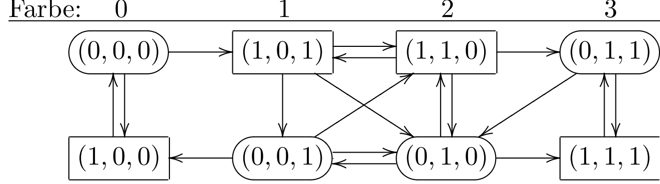


- `existence_2.game`:

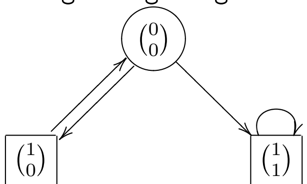


- `security.game`: entspricht dem Spielgraphen von `existence_2.game`

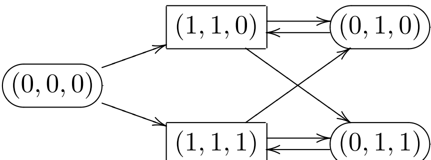
- `weakparity.game`:



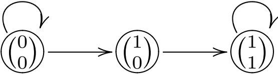
- `staiger-wagner.game`:



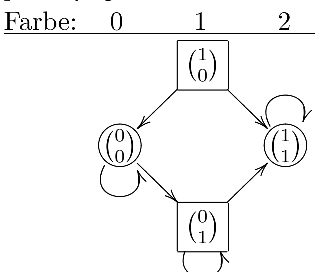
- request-response.game:



- request-response_2.game:



- request-response_3.game:



- buechi.game entspricht dem Spielgraphen von weakparity.game, wobei Spieler 0 und Spieler 1-Knoten vertauscht sind

- parity.game:

- parity_2.game entspricht dem Spielgraphen von weakparity.game

- fallstudie.game siehe Kapitel 4.3

Anhang C

Verwendete Software

Compiler

- Java Development Kit 1.4.1_01
  http://www.java.sun.com
- G++ 2.95
  http://gcc.gnu.org

Integrated Development Environment

- eclipse 2.0
  http://www.eclipse.org

Weitere Bibliotheken

- BuDDy - A Binary Decision Diagram Package
  http://www.it-c.dk/research/buddy/
- SWT - Standard Widget Toolkit
  http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-swt-home/index.html

Parser Generator

- JavaCC 2.1 (für einen Prototypen)
  http://www.webgain.com/products/java_cc
- SableCC 2.16.2
  http://www.sablecc.org

Pretty Printer

- Jalopy
  http://jalopy.sourceforge.net

Test-Framework

- JUnit
  http://www.junit.org

Coverage Analyzer

- JCover 2.0
  http://www.mmsindia.com/JCover.html

UML CASE Tool

- Poseidon for UML Community Edition 1.4
  http://www.gentleware.com

Versionsverwaltung

- Concurrent Version System
  http://www.cvshome.org

Literaturverzeichnis

[Ake78] AKERS, S. B.: *Binary Decision Diagrams.* IEEE Transactions on Computers, C-27(6), Juni 1978.

[AKY99] ALUR, RAJEEV, SAMPATH KANNAN und MIHALIS YANNAKAKIS: *Communicating Hierarchical State Machines.* Lecture Notes in Computer Science, 1644:169–??, 1999.

[App97] APPEL, ANDREW W.: *Modern Compiler Implementation in Java: Basic Techniques.* Cambridge University Press, Cambridge, UK, Februar 1997.

[Büc62] BÜCHI, J. R.: *On a Decision Method in Restricted Second Order Arithmetic.* In: *Proceedings of the International Congress on Logic, Method, and Philosophy of Science*, Seiten 1–12, Stanford, CA, USA, 1962. Stanford University Press.

[Büc83] BÜCHI, J. R.: *State-strategies for games in* $F_{\sigma\delta} \cap G_{\delta\sigma}$. J. Symb. Logic, 48:1171–1198, 1983.

[BCM+90] BURCH, J. R., E. M. CLARKE, K. L. MCMILLAN, D. L. DILL und L. J. HWANG: *Symbolic Model Checking:* 10^{20} *States and Beyond.* In: *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, Seiten 428–439, Philadelphia, Pennsylvania, 4–7 Juni 1990. IEEE Computer Society Press.

[Bec98] BECK, KENT: *Extreme Programming: A Humanistic Discipline of Software Development.* Lecture Notes in Computer Science, 1382:1–6, 1998.

[Bec00] BECK, KENT: *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[BG98] BECK, KENT und ERICH GAMMA: *Test Infected: Programmers Love Writing Tests.* Java Report, 3(7):37–50, 1998.

[BGR01] BENEDIKT, MICHAEL, PATRICE GODEFROID und THOMAS REPS: *Model Checking of Unrestricted Hierarchical State Machines.* Lecture Notes in Computer Science, 2076:652–??, 2001.

[BL69] BÜCHI, J. R. und L. H. LANDWEBER: *Solving sequential conditions finite-state strategies.* Trans. Ameri. Math. Soc., 138:295–311, 1969.

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

[Bry92] Bryant, Randal E.: *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams.* ACM Computing Surveys, 24(3):293–318, September 1992.

[CGP99] Clarke, Edmund M., Orna Grumberg und Doron A. Peled: *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.

[Chu62] Church, Alonzo: *Logic, Arithmetic and Automata.* In: *Proceedings of the International Congress of Mathematicians*, Seiten 23–35, 1962.

[CS01] Clarke, E. M. und H. Schlingloff: *Model Checking.* In: Robinson, A. und A. Voronkov (Herausgeber): *Handbook of Automated Reasoning*, Band II, Kapitel 24, Seiten 1635–1790. Elsevier Science, 2001.

[EJ91] Emerson, E. A. und C. S. Jutla: *Tree automata, μ-calculus and determinacy.* In: IEEE (Herausgeber): *Proceedings: 32nd annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, October 1–4, 1991*, Seiten 368–377, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1991. IEEE Computer Society Press.

[EJS93] Emerson, E. A., C. S. Jutla und A. P. Sistla: *On Model-Checking for Fragments of μ-Calculus.* In: Courcoubetis, C. (Herausgeber): *Proc. 5th International Computer-Aided Verification Conference*, Band 697 der Reihe *Lecture Notes in Computer Science*, Seiten 385–396. Springer, 1993.

[Gag98] Gagnon, E.: *Sablecc, an object-oriented compiler framework.* Diplomarbeit, School of Computer Science McGill University, Montreal, März 1998.

[GH82] Gurevich, Yuri und Leo Harrington: *Trees, Automata, and Games.* In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, Seiten 60–65, San Francisco, California, 5–7 Mai 1982.

[GHJV96] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.

[Hüt03] Hütten, Patrick: *Automatische Synthese optimaler Controller für request-response-Spezifikationen.* Diplomarbeit, RWTH Aachen, 2003.

[Jur00] Jurdzinski, Marcin: *Small Progress Measures for Solving Parity Games.* Lecture Notes in Computer Science, 1770:290–??, 2000.

[Kla94] Klarlund, Nils: *Progress measures, immediate determinacy, and a subset construction for tree automata.* Annals of Pure and Applied Logic, 69(2–3):243–268, 14 Oktober 1994.

[Lee59] Lee, C. Y.: *Representation of Switching Circuits by Binary-Decision Programs.* Bell System Technical Journal, 38:985–999, Juli 1959.

[Lia99] LIANG, SHENG: *Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, Reading, MA, USA, 1999.

[Lic00] LICHTER, HORST: *Vorlesung: Software-Qualitätssicherung*, 2000.

[LNAB+98] LIND-NIELSEN, JØRN, HENRIK REIF ANDERSEN, GERD BEHRMANN, HENRIK HULGAARD, KÅRE KRISTOFFERSEN und KIM G. LARSEN: *Verification of Large State/Event Systems Using Compositionality and Dependency Analysis.* Lecture Notes in Computer Science, 1384:201–??, 1998.

[Mar75] MARTIN, D.: *Borel determinacy.* Ann. Math., 102:363–371, 1975.

[McM93] MCMILLAN, K.: *Symbolic model checking: An approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[McN65] MCNAUGHTON, ROBERT: *Finite-State Infinite Games.* Technischer Bericht, Massachusetts Institute Of Technology, September 1965.

[McN93] MCNAUGHTON, ROBERT: *Infinite games played on finite graphs.* Annals of Pure and Applied Logic, 65(2):149–184, 1 Dezember 1993.

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

[Mor82] MORET, BERNARD M. E.: *Decision Trees and Diagrams.* ACM Computing Surveys, 14(4):593–623, Dezember 1982.

[Mos84] MOSTOWSKI, A. W.: *Regular expressions for infinite trees and a standard form of automata.* In: SKOWRON, ANDRZEJ (Herausgeber): *Proceedings of the 5th Symposium on Computation Theory*, Band 208 der Reihe *LNCS*, Seiten 157–168, Zaborów, Poland, Dezember 1984. Springer.

[MP92] MANNA, ZOHAR und AMIR PNUELI: *The Temporal Logic of Reactive and Concurrent Systems.* Springer, New York, 1992.

[MT98] MEINEL, C. und T. THEOBALD: *Algorithmen und Datenstrukturen im VLSI-Design.* Springer-Verlag, 1998.

[Mul63] MULLER, DAVID E.: *Infinite sequences and finite machines.* In: *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, Seiten 3–16, Chicago, Illinois, 28–30 Oktober 1963. IEEE.

[Rab69] RABIN, M.O.: *Decidability of second-order theories and automata on infinite trees.* Trans. Amer.Math.Soc., 141:1–35, 1969.

[Rab72] RABIN, M.O.: *Automata on Infinite Objects and Church's Problem.* Amer.Math.Soc.Providence, RI, 1972.

[Saf92] SAFRA, S.: *Exponential Determinization for ω-Automata with Strong Fairness Acceptance Condition.* In: ALON, N. (Herausgeber): *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, Seiten 275–282, Victoria, B.C., Canada, Mai 1992. ACM Press.

[SBB99] SCHOLL, C., B. BECKER und A. BROGLE: *Solving the Multiple Variable Order Problem for Binary Decision Diagrams by Use of Dynamic Reordering Techniques.* Technical Report report00130, Albert-Ludwigs-Universitaet Freiburg, Institut fuer Informatik, Oktober 4, 1999.

[Som01] SOMMERVILLE, IAN: *Software Engineering.* Addison-Wesley, 6. Auflage, 2001.

[Str82] STREETT, R. S.: *Propositional Dynamic Logic of Looping and Converse is Elementary Decidable.* IC, 54:121–141, 1982.

[SW74] STAIGER, L. und K.W. WAGNER: *Automatentheoretische Charakterisierungen topologischer Klassen regulärer Folgenmengen.* Elektron. Informationsverarb. Kybernet., 10:379–392, 1974.

[Tho95] THOMAS, WOLFGANG: *On the Synthesis of Strategies in Infinite Games.* Lecture Notes in Computer Science, 900:1–13, 1995.

[Tho96] THOMAS, WOLFGANG: *Languages, Automata, and Logic.* Technical Report 9607, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, Mai 1996.

[Tho00] THOMAS, WOLFGANG: *Vorlesung: Automaten und reaktive Systeme*, 2000.

[Val98] VALMARI, ANTTI: *The State Explosion Problem.* Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models, 1491:429–528, 1998.

[Vög00] VÖGE, JENS: *Strategiesynthese für Paritätsspiele auf endlichen Graphen.* Doktorarbeit, RWTH Aachen, 2000.

[Zie98] ZIELONKA, WIESŁAW: *Infinite games on finitely coloured graphs with applications to automata on infinite trees.* Theoretical Computer Science, 200(1–2):135–183, 28 Juni 1998. Fundamental Study.