Strategien in unendlichen Spielen mit Liveness-Gewinnbedingungen: Syntheseverfahren, Optimierung und Implementierung

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Nico Wallmeier
aus Hilden

Berichter: Univ.-Prof. Dr. Dr.h.c. Wolfgang Thomas
Univ.-Prof. Gerhard Lakemeyer, Ph.D.

Tag der mündlichen Prüfung: 06. September 2005

Zusammenfassung

Gegenstand dieser Arbeit sind Lösungsverfahren für unendliche Spiele und die Implementierung entsprechender Algorithmen im Rahmen einer Experimentierplattform.

Der Schwerpunkt liegt auf Spielen mit Gewinnbedingungen, welche gewisse Liveness-Eigenschaften fordern. Eine für Anwendungen (etwa in der Controller-Synthese) zentrale Liveness-Eigenschaft ist die "Request-Response-Bedingung". Sie ist eine Konjunktion von Bedingungen der folgenden Gestalt: Immer wenn ein "Request"-Zustand besucht wird, folgt irgendwann später auch der Besuch eines entsprechenden "Response"-Zustandes. Eng verwandt damit sind die sogenannten "Streett-Bedingungen", in denen für wiederholte Besuche gewisser Zustände wiederholte Besuche anderer Zustände gefordert werden.

Es werden verschiedene Lösungsverfahren für Request-Response-Spiele und Streett-Spiele vorgestellt, mit Anwendungen etwa in der Analyse von Live-Sequence-Charts. Der Hauptbeitrag besteht in einer quantitativen Analyse der Request-Response-Spiele. Ein natürlicher Ansatz zur quantitativen Abstufung von Gewinnstrategien berücksichtigt die Wartezeiten, die in einer unendlichen Partie zwischen den Besuchen von "Request'"- und nachfolgenden "Response"-Zuständen verstreichen.

Dazu werden verschiedene Qualitätsmaße für Partien in Request-Response-Spielen (über endlichen Spielgraphen) eingeführt und diskutiert. Für Maße, in denen die "Strafe" mehr als linear in den Wartezeiten steigt, wird eine algorithmische Berechnung optimaler Gewinnstrategien vorgestellt. Kernpunkt ist eine Reduktion auf Mean-Payoff-Spiele über endlichen Zustandsräumen, mit der zugleich gezeigt wird, dass optimale Strategien durch endliche Automaten implementierbar sind.

Die Experimentierplattform GaSt ("Games, Automata & Strategies") integriert zahlreiche Algorithmen zur Theorie der ω-Automaten und zur Lösung unendlicher Spiele.

Abstract

In this thesis we develop methods for the solution of infinite games and present implementations of corresponding algorithms in the framework of a platform for the experimental study of automata theoretic algorithms.

Our focus is on games with winning conditions that express certain liveness properties. A central type of liveness requirement in applications (e.g., in controller synthesis) is the "request-response condition". It has the form of a conjunction of conditions "Whenever a "request"-state is visited, sometime later a corresponding "response"-state is visited". A closely related winning condition is the "Streett condition" in which for repeated visits of certain states the repeated visits of other states is required.

We present methods for the solution of request-response games and Streett games, the latter with an application in the analysis of live-sequence-charts. The main contribution is a quantitative analysis of request-response games. We pursue a natural approach for the quantitative evaluation of winning strategies by taking into account the waiting times that elapse between visits of "request"-states and subsequent visits of "response"-states in an infinite play.

We introduce and discuss several related measures of plays in request-response games (over finite game arenas). For measures that induce a "penalty" which grows more than linearly in the waiting times, we present an algorithm to compute optimal winning strategies. The core of the argument is a reduction to mean-payoff games over finite arenas; it also shows that optimal strategies are implementable by finite-state machines.

The experimental platform GaSt ("Games, Automata & Strategies") offers numerous algorithms of the theory of ω-automata and for the solution of infinite games.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Hintergrund und Motivation

Die Konstruktion korrekter und effizienter Hardware- und Softwaresysteme ist eine der zentralen Herausforderungen der Informatik. Die Vergangenheit hat gezeigt, dass in diesem Bereich noch immer Nachholbedarf besteht. Stellvertretend seien hier drei prominente Vorfälle genannt, bei denen Fehler in der Software gravierende Auswirkungen hatten:

- 1962: Die Raumsonde Mariner I weicht aufgrund eines Softwarefehlers vom vorausberechneten Kurs ab und muss schließlich über dem Atlantik gesprengt werden.
- 1993: Intels neu erschienener Pentium-Prozessor liefert bei der Division von Komma-Zahlen falsche Ergebnisse.
- 1996: Teile der Programmierung der Ariane-4-Rakete werden für den Nachfolger Ariane 5 übernommen, unter anderem ein Modul, das dazu dient, Zahlenwerte zu konvertieren. Allerdings sind die Motoren der neuen Ariane schneller – das führt dazu, dass das Modul die Berechnungen nicht mehr bewältigen kann und den Bordcomputer zum Absturz bringt. 40 Sekunden nach dem Start bricht die Rakete auseinander.

Das Problem gewinnt auch dadurch an Bedeutung, dass Computer-Komponenten und Software sich inzwischen in fast allen Lebensbereichen befinden – und damit auch Probleme, die durch falsch programmierte Software ausgelöst werden.

Mögliche Ansätze zur Lösung des Problems können im Testen und Simulieren von Software 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 *Model Checking* ([CGP99]). In [CS01] wird Model Checking definiert als:

> 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 ein Modell des Systems erzeugt – in den meisten Fällen ein Transitionsgraph oder Kripke-Struktur. Die Spezifikation wird häufig als Formel der temporalen Logik formuliert und zusammen mit der Kripke-Struktur dem Model-Checker übergeben. Erfüllt das Modell die Formel nicht, wird ein Fehlerszenario zurückgegeben.

Ein darüber hinaus gehendes Ziel sollte es sein, Verfahren der algorithmischen Synthese von Hardware bzw. Software aus einer vorliegenden Spezifikation zu finden. Synthese bedeutet hierbei, dass die Spezifikation automatisch in eine Implementierung transformiert wird, die bedingt durch ihre Konstruktion korrekt ist. Diese anspruchsvollere Zielsetzung gegenüber der Verifikation bedeutet eine erhöhte algorithmische Komplexität und zuweilen sogar den Verlust eines automatisierbaren Vorgehens. Während Methoden der Softwarevalidierung und -verifikation inzwischen gut etabliert sind, abgesichert durch adäquate formale Modelle und auch erprobt in der praktischen Anwendung, ist der Ansatz der automatischen Synthese von Software bzw. Hardware noch nicht so weit entwickelt. Der Ansatz der Programmsynthese ist allerdings nur in eingeschränkten Szenarien realistisch, vor allem in reaktiven (Multi-Agenten-)Systemen.

Unendliche Zwei-Personen-Spiele

Reaktive Systeme sind dadurch gekennzeichnet, dass es fortwährende Interaktion mehrerer Systemkomponenten gibt. Beispiele für zustandsbasierte reaktive Systeme sind u.a. Kommunikationsprotokolle in Netzwerken, digitale Schaltungen und die industrielle Regelungstechnik. Die Entwicklung aktueller Systeme geht immer mehr in diese Richtung und wird von Milner mit der Kurzformel „computing is interaction“ ([Mil89]) bezeichnet. Der einfachste Fall besteht darin, dass die Komponenten „Controller“ und „Umgebung“ unterschieden werden. Eine Spezifikation des Systems legt dann fest, welche Systemläufe der beiden Komponenten zulässig sein sollen. Ein adäquates Modell für diesen Fall sind die unendlichen Zwei-Personen-Spiele.

Aufbauend auf den Arbeiten der sechziger Jahre von Church [Chu62], Büchi [Büc62], Rabin [Rab69] und weiteren sind die unendlichen Zwei-Personen-Spiele auf endlichen Spielgraphen konzipiert worden. In den Arbeiten von Church, Büchi und Rabin wird

eine Logik-Beschreibung in ein deterministisches Transitionssystem und eine Akzeptierbedingung (ω-Automat) umgesetzt. McNaughton ([McN65, McN93]) sowie Gurevich und Harrington ([GH82]) haben damit begonnen, Spielgraph und Gewinnbedingung zu trennen.

Spielgraph

Wir betrachten Zwei-Personen-Spiele mit den Spielern 0 und 1. Die Knoten des Spielgraphen sind die möglichen Zustände des betrachteten reaktiven Systems. Jeder Knoten ist genau einem Spieler zugeordnet. Es entscheidet genau der Spieler über den Folgezustand des Systems, dem der Knoten des aktuellen Zustands gehört. Eine Partie in einem Zwei-Personen-Spiel ist eine unendliche Folge von Zuständen im Spielgraphen. Aus diesem Grund wird meistens gefordert, dass jeder Knoten mindestens eine ausgehende Kante hat. Spieler 0 gewinnt eine Partie genau dann, wenn eine vorher vorgegebene Gewinnbedingung erfüllt worden ist.

Gewinnbedingung

Dabei kann für die Gewinnbedingung entweder direkt eine automatentheoretische Akzeptierbedingung angegeben werden, oder sie wird mittels einer Logik spezifiziert, aus der wiederum häufig eine Akzeptierbedingung abgeleitet werden kann. Dies entspricht der heute üblichen Form unendlicher Zwei-Personen-Spiele auf endlichen Spielgraphen. Diese werden spezifiziert durch ihren Spielgraphen sowie eine Gewinnbedingung an die unendlichen Läufe auf dem Spielgraphen. In der Praxis müssen Spielgraph und Gewinnbedingung gemeinsam behandelt werden, da sich die Anforderungen an ein System nur mittels beider Komponenten des Spiels zusammen festlegen lassen.

Request-Response-Bedingung

Eine wichtige Gewinnbedingung für unendliche Zwei-Personen-Spiele, die in der vorliegenden Arbeit eine zentrale Bedeutung hat, ist die "Request-Response"-Bedingung, in der Literatur auch manchmal Assume-Guarantee-Bedingung genannt. Die Gewinnbedingung ist dabei durch eine endliche Menge von Tupeln (P_i, R_i) gegeben, wobei P_i für die Anforderungsmenge einer Bedingung steht und R_i für die dazu passende Antwortmenge. Spieler 0 gewinnt eine Partie in einem Request-Response-Spiel genau dann, wenn (für jedes i) nach jedem Besuch eines Zustands der P_i-Menge irgendwann später auch ein Zustand der passenden Antwortmenge R_i aufgesucht wird. Die Bedeutung der Request-Response-Bedingung liegt darin begründet, dass sie eine einfache Form der Liveness-Bedingung darstellt, die eine wichtige und somit häufig vorkommende Anforderung an reaktive Systeme ist. Somit ist es möglich, eine große Anzahl praxisbezogener Spezifikationen von reaktiven Systemen durch Request-Response-Spiele, kurz RR-Spiele, zu kodieren.

ω-Automaten

Das Modell der ω-Automaten ist für unendliche Zwei-Personen-Spiele insofern relevant, als dass es für die Spezifikation der Gewinnbedingungen benötigt wird. Spieler 0 gewinnt genau die Partien, die der der Gewinnbedingung zugrundeliegende deterministische Automat akzeptiert. Büchi hat mit seiner Arbeit ([Büc62]) Automaten mit unendlichen Läufen eingeführt. Weitere ω-Automaten wurden unter anderem eingeführt von Muller ([Mul63]), Rabin ([Rab72]), Streett ([Str82]), Staiger und Wagner ([SW74]). Eine Sonderrolle nimmt die Paritätsbedingung ([Mos84, EJ91]) ein, da sich alle genannten Gewinnbedingungen darauf zurückführen lassen. Dazu ist aber bei den meisten Gewinnbedingungen eine Vergrößerung des Zustandsraumes der betrachteten Automaten notwendig. Zu einem Muller-Automaten kann der konstruierte Paritätsautomat exponentiell größer sein als der Gegebene.

Ein zentrales Ergebnis in der Theorie der ω-Automaten ist der Satz von McNaughton [McN66]. Dieser besagt in seiner Originalformulierung, dass ein nicht-deterministischer Büchi-Automat in einen deterministischen Muller-Automaten transformiert werden kann. Diese Transformation ist für die Spielspezifikation eine Grundvoraussetzung (vgl. Einführung zu Kapitel 2). Bei der Elimination des Nichtdeterminismus ist der "Blow-up" bezogen auf die Zustandsanzahl größer als bei der klassischen Potenzmengenkonstruktion [HU79]. Die Algorithmen von Safra [Saf88] bzw. Muller/Schupp [MS95] erreichen beide die untere Schranke für die Determinisierung eines Büchi-Automaten mit n Zuständen, die bei $2^{O(n \log n)}$ Zuständen liegt. In dieser Arbeit werden beide Konstruktionen genauer untersucht und für die Muller/Schupp-Variante noch eine Verbesserung vorgestellt, die in vielen Fällen deutlich Zustände einsparen kann. Die Determinisierung von ω-Automaten ist eine notwendige Voraussetzung bei der Spezifikation von unendlichen Zwei-Personen-Spielen.

Gewinnbereiche und -strategien

Das Syntheseproblem für reaktive Systeme ist die Frage nach der Gewinnstrategie für das Controller-Programm in dem korrespondierenden unendlichen Zwei-Personen-Spiel. Dabei versteht man unter einer Gewinnstrategie eine Vorschrift zur Zug-Auswahl, mit der ein Spieler den Gewinn erzwingen kann – unabhängig davon, wie sich der andere verhält. Der Gewinnbereich eines Spielers umfasst alle die Knoten, von denen aus er eine Gewinnstrategie besitzt. Bei allen im weiteren Verlauf betrachteten Spielen hat für jeden Knoten des Spielgraphen genau einer der beiden Spieler eine Gewinnstrategie, d. h. die Spiele sind determiniert. Dies folgt aus der Tatsache, dass die hier betrachteten automatentheoretischen Gewinnbedingungen zu Spielen führen, die in der deskriptiven Mengenlehre zu den ersten beiden Stufen der Borel-Hierarchie gehören. Nach Martin [Mar75] folgt somit, dass alle betrachteten Spiele determiniert sind.

Ergebnisse dieser Arbeit

Die Ergebnisse dieser Arbeit gliedern sich in drei zusammenhängende Teile. Im ersten Teil werden die Algorithmen zur Determinisierung von Büchi-Automaten von Safra [Saf88] und Muller/Schupp [MS95] genauer analysiert. Dabei wird erstmals auch eine experimentelle Untersuchung durchgeführt. Es wird auch gegenüber den in der Literatur dokumentierten Verfahren eine deutliche Verbesserung in Bezug auf die Zustandsanzahl des resultierenden deterministischen Automaten erreicht.

Der zweite Teil beschäftigt sich mit den unendlichen Zwei-Personen-Spielen, die die Grundlage für die automatische Controllersynthese bilden. Dabei wird die Request-Response-Gewinnbedingung genauer analysiert. Neben der Optimierung der Speichergröße des resultierenden Strategieautomaten bei Request-Response- und verallgemeinerten Büchi-Spielen wird auch eine Anwendung der unendlichen Zwei-Personen-Spiele betrachtet: die Strategiesynthese für Live-Sequence-Charts-Spezifikationen und die daraus auf natürliche Weise entstehenden einfachen Streett-Spiele. Diese sind mächtiger als Büchi-Spiele, jedoch einfacher als allgemeine Streett-/Rabin-Spiele. Zur Lösung solcher einfachen Streett-Spiele wird ein neuer Algorithmus vorgestellt, der diese Spiele direkt (ohne eine Spielreduktion) löst.

Die klassischen Algorithmen der Spieltheorie liefern nur eine qualitative Analyse eines RR-Spiels – nämlich nur die Gewinnbereiche und -strategien für beide Spieler. Das Hauptresultat dieser Arbeit zielt auf eine quantitative Verschärfung, nämlich die Optimalität der Gewinnstrategien in Request-Response-Spielen. Dabei ist aber die Optimalität nicht bezogen auf die Zustandsanzahl des Automaten, der die Gewinnstrategie implementiert, sondern auf die Güte des Gewinnens bzw. Verlierens. Wir ergänzen damit Ergebnisse der Literatur zu optimalen Strategien für Payoff-Gewinnbedingungen, z.B. von Gimbert und Zielonka [GZ05, GZ06] oder von de Alfaro, Henzinger und Majumdar [dAHM03].

Der letzte Teil beschreibt die Implementierung einer Experimentierplattform für unendliche Spiele mit dem Namen GASt (Games, Automata & Strategies). Diese umfasst drei Komponenten: die Determinisierung von Büchi-Automaten, Algorithmen zur Lösung unendlicher Zwei-Personen-Spiele und als Anwendung davon die Strategiesynthese für Live-Sequence-Charts-Spezifikationen. In der Komponente der Strategiesynthese unendlicher Zwei-Personen-Spiele sind wichtige Algorithmen aus der Literatur und selbstentwickelte sowohl enumerativ als auch symbolisch umgesetzt, sofern dies möglich ist (siehe dazu auch die Tabellen 5.1 sowie 5.2 auf Seite 99). Die in der Literatur erwähnten Implementierungen lösen jeweils lediglich einen Spieltyp und lassen sich nicht einfach um weitere Gewinnbedingungen erweitern, worauf bei der Experimentierplattform GASt besonderer Wert gelegt worden ist. Dieser Vorteil zeigt sich z.B. im Vergleich

mit der Implementierung von Harding, Ryan und Schobbens [HRS05] zur Lösung von LTL-Gewinnbedingungen oder mit der Implementierung von Melcher [Mel01], bei der eine stark vereinfachende Annahme über die Gewinnbedingung (sie wird als elementare Streett-Bedingung, d. h. Konjunktion mit nur einem Glied, vorgegeben) zugrunde liegt.

Gliederung

Im folgenden Abschnitt werden die Determinisierungsalgorithmen für Büchi-Automaten von Safra und Muller/Schupp genauer analysiert. Weiterhin wird eine verbesserte Variante der Muller/Schupp Konstruktion vorgestellt, die zum Teil eine deutliche Einsparung von Zuständen im resultierenden Automaten ermöglicht.

Im dritten Kapitel werden die theoretischen Grundlagen von unendlichen Zwei-Personen-Spielen beschrieben sowie die notwendigen Notationen eingeführt. Weitere Themen in diesem Abschnitt sind die Request-Response-Spiele und die Speicheroptimierung bei der Spielreduktion von RR-Spielen und verallgemeinerten Büchi-Spielen. Den Abschluss des Kapitels bildet die praktische Anwendung unendlicher Zwei-Personen-Spiele in der Strategiesynthese für Live-Sequence-Charts-Spezifikationen.

Das vierte Kapitel enthält das Hauptergebnis dieser Arbeit. Optimale Strategien in Request-Response-Spielen werden definiert und analysiert. Anschließend wird ein Algorithmus zur Berechnung optimaler Strategien in RR-Spielen vorgestellt.

Zum Abschluss der Arbeit geht das fünfte Kapitel auf die Implementierung der Experimentierplattform GASt genauer ein. Neben der Vorstellung der einzelnen Komponenten dieser Implementierung werden softwaretechnische Aspekte zur Entwicklung und Anwendung der Plattform diskutiert.

Dank

Ich möchte Wolfgang Thomas danken, der mir die Gelegenheit zu dieser Arbeit gegeben und mich mit gutem Rat und vielfältigen Anregungen unterstützt hat. Viele wichtige Anregungen verdanke ich auch den Gesprächen mit Florian Horn während seiner Aufenthalte in Aachen. Herrn Rossmanith danke ich für die Unterstützung bei der Abschätzung der Funktion in Bemerkung 4.31.

Kapitel 2

Determinisierung von ω-Automaten

Die Determinisierung von ω-Automaten ist eine Grundvoraussetzung für die Spezifikation von unendlichen Zwei-Personen-Spielen. In vielen Anwendungen wird die Menge der möglichen Gewinnpartien eines unendlichen Zwei-Personen-Spiels zunächst durch einen nicht-deterministischen Büchi-Automaten $\mathcal{A}$ gegeben. Diese Spezifikationsform ist für die Konstruktion von Gewinnstrategien ungeeignet: Während einer Partie würde Spieler 0 versuchen, einen akzeptierenden Lauf des Automaten $\mathcal{A}$ nachzubilden, der die Büchi-Bedingung erfüllt. Das Problem dabei ist aber, dass eine Gewinnstrategie den nächsten Zug nur anhand des aktuellen Partiepräfixes bestimmen kann und der akzeptierende Lauf des Büchi-Automaten von der Zukunft eines Partieverlaufs abhängen kann.

Beim Model-Checking kann man nicht-deterministische Automaten für die Logik-Spezifikation benutzen (da der Nicht-Leerheitstest genügt). Für die Spezifikation von unendlichen Zwei-Personen-Spielen muss die Gewinnbedingung in einen deterministischen Automaten umgewandelt werden.

Eine von Büchi [Büc83] angeregte Sicht auf das Problem der Determinisierung liefert einen anderen Aspekt: Ein ω-Automat kann als ein unendliches Ein-Personen-Spiel aufgefasst werden, bei dem die Zustandsbesuche des Automaten als Aktionen des Spielers in dem unendlichen Ein-Personen-Spiel angesehen werden. Dann bedeutet die Determinisierung eines solchen ω-Automaten $\mathcal{A}$, dass der Spieler genau dann eine Gewinnstrategie in dem Spiel hat, wenn der Automat $\mathcal{A}$ das Eingabewort akzeptiert. Somit entspricht die Determinisierung von ω-Automaten dem Lösen von unendlichen Ein-Personen-Spielen.

In der Theorie der ω-Automaten wird das Problem der Determinisierung durch den Satz von McNaughton [McN66] gelöst. In der Originalformulierung besagt er, dass nicht-deterministische Büchi-Automaten in deterministische Muller-Automaten transformiert werden können. Viele Konstruktionen sind vorgeschlagen worden, um dieses

Determinisierungsergebnis zu zeigen (siehe [Tho90] bzw. [PP04] für weitere Literaturhinweise). In den meisten Fällen wird als Zielautomat ein deterministischer Rabin-Automat verwendet, der als Spezialfall eines Muller-Automaten angesehen werden kann. Es ist wohl bekannt, dass der Blow-up bezogen auf die Anzahl der Zustände größer ist als bei der klassischen Potenzmengenkonstruktion. Die untere Schranke für die Zustandsanzahl bei der Minimierung eines Büchi-Automaten mit n Zuständen liegt bei $2^{O(n \log n)}$ Zuständen für einen deterministischen Rabin-Automaten ([Mic88, Löd99]).

Safra [Saf88] war der erste, der mit seiner Konstruktion diese untere Schranke erreicht hat. Dies ist mittlerweile in der Literatur die Standardkonstruktion, um einen Büchi-Automaten zu determinisieren. Aber es gibt noch eine zweite Konstruktion, die auch die untere Schranke erfüllt: die Konstruktion von Muller und Schupp [MS95]. Erfahrungen in der Lehre haben gezeigt, dass sich das Verfahren von Muller und Schupp einfacher erklären lässt. Die beiden Verfahren sind ausreichend unterschiedlich, so dass sich ein genauerer Vergleich der beiden lohnt.

Unsere Studien basieren auf der Implementierung der beiden Algorithmen, die Teil der Experimentierplattform GASt sind (siehe dazu Kapitel 5). Wie sich herausgestellt hat, ist solch eine Implementierung nicht nur notwendig, um auf praktischer Ebene die beiden Algorithmen zu vergleichen, sondern auch, um das Verhalten der beiden Algorithmen konzeptionell zu verstehen. Die beiden Algorithmen sind zu kompliziert, um mehrere Dutzend Beispiele manuell zu analysieren. Im Weiteren sollen einige Beobachtungen der Untersuchungen der beiden Algorithmen vorgestellt werden, um ein besseres Verständnis der Eigenschaften (und Gemeinsamkeiten) beider Algorithmen zu vermitteln. Dabei wurde natürlich auch die Güte der Algorithmen (in Bezug auf die Größe des Zielautomaten) berücksichtigt. Festzustellen ist (abgesehen von wenigen Ausnahmen), dass der Safra-Algorithmus eine stärkere Abstraktion als der Muller-Schupp benutzt, was zu einem kleineren deterministischen Automaten führt. (Vielleicht ist dies auch die Ursache dafür, dass sich der Safra-Algorithmus in der Lehre schwieriger vermitteln lässt.) Die praktischen Experimente mit den beiden Algorithmen haben zu einer verbesserten Version des Muller-Schupp-Algorithmus geführt, die die Zustandsanzahl des Zielautomaten zum Teil deutlich reduziert.

Im folgenden Unterabschnitt werden die beiden Determinisierungsverfahren noch genauer vorgestellt. Hierbei liegt der Schwerpunkt darauf, die Gemeinsamkeiten und Unterschiede der beiden aufzuzeigen. Es werden die Erkenntnisse aus den Experimenten ausgenutzt; begonnen wird mit einer Erklärung des Muller-Schupp-Algorithmus und einer kürzeren Diskussion der Safra-Konstruktion. Auf einen Korrektheitsbeweis für die Safra-Konstruktion wird an dieser Stelle verzichtet. Weiterhin wird die verbesserte Version des Muller-Schupp-Algorithmus vorgestellt. In Kapitel 5 wird dann die Experimentierplattform GASt noch genauer vorgestellt, dessen Komponente zur Deter-

minisierung von ω-Automaten auch eine Variante der Potenzmengenkonstruktion von Hayashi und Miyano [MH84] für co-Büchi-Automaten umfasst.

2.1 Die Algorithmen von Safra und Muller-Schupp

Definition 2.1 (ω-Automat)**.** Ein nicht-deterministischer ω-Automat hat die Form $\mathcal{A} = (Q, \Sigma, q_0, \Delta, Acc)$ mit:

- Σ als endliches Eingabealphabet,
- Q als endlicher Menge von Zuständen,
- $q_0 \in Q$ als Anfangszustand,
- $\Delta \subseteq Q \times \Sigma \times Q$ als Transitionsrelation und
- Acc als Akzeptierkomponente.

Ein deterministischer ω-Automat hat statt einer Transitionsrelation eine Transitionsfunktion und somit die Form $\mathcal{A} = (Q, \Sigma, q_0, \delta, Acc)$ mit:

- Q, Σ, q_0 und Acc wie beim nicht-deterministischen ω-Automaten und
- $\delta : Q \times \Sigma \to Q$ als Transitionsfunktion.

Der Lauf eines ω-Automaten $\mathcal{A}$ auf einem gegebenen ω-Wort ist im üblichen Sinne definiert.

Definition 2.2 (Büchi-Automat)**.** Bei einem Büchi-Automaten $\mathcal{A}$ ist die Akzeptierkomponente durch eine Menge $F \subseteq Q$, die auch "Endzustände" genannt werden, gegeben. Ein Büchi-Automat $\mathcal{A}$ akzeptiert ein ω-Wort $\alpha \in \Sigma^\omega$ genau dann, wenn ein Lauf $\rho \in Q^\omega$ existiert, bei dem unendlich oft Zustände aus F auftreten.

Definition 2.3 (Rabin-Automat)**.** Bei einem Rabin-Automat $\mathcal{A}$ ist die Akzeptierkomponente $\Omega = ((E_1, F_1), \ldots, (E_k, F_k))$ als eine Menge von "akzeptierenden Paaren" mit $E_j, F_j \subseteq Q$ gegeben. Der Rabin-Automat $\mathcal{A}$ akzeptiert ein Eingabewort $\alpha \in \Sigma^\omega$ genau dann, wenn es für einen Lauf $\rho \in Q^\omega$ von $\mathcal{A}$ auf α einen Index j gibt, so dass F_j-Zustände unendlich oft besucht werden, aber jeder Zustand aus E_j nur endlich oft.

Wird im Weiteren von einem Büchi-Automaten gesprochen, ist immer die nicht-deterministische Variante gemeint. Ist hingegen die Rede von einem Rabin-Automaten, ist immer die deterministische Variante gemeint. Ausgehend von einem ω-Wort α lässt sich der *Berechnungsbaum* eines Büchi-Automaten $\mathcal{A}$ wie folgt definieren:

Definition 2.4 (Berechnungsbaum t_α)**.** Der Berechnungsbaum t_α eines Büchi-Automaten $\mathcal{A}$ für das ω-Wort α hat als Wurzel einen Knoten, der mit q_0 beschriftet ist (bezeichnet als "Ebene 0") und auf allen Ebenen $i = 0, 1, 2, \ldots$ mit den erreichbaren Zuständen des α-Präfixes $\alpha(0) \ldots \alpha(i-1)$. Für einen Knoten auf der Ebene i, der mit p beschriftet ist, sind die Nachfolgerknoten beschriftet mit $q_1, \ldots, q_k$, wenn für den Zustand p und den Buchstaben $a = \alpha(i)$ die Transitionen $(p, a, q_1), \ldots, (p, a, q_k)$ anwendbar sind. Es wird vorausgesetzt, dass Geschwisterknoten untereinander geordnet sind, in Abhängigkeit von einer Ordnung auf den Zuständen.

Aus dem Berechnungsbaum t_α lässt sich der *Laufgraph* r_α ableiten:

Definition 2.5 (Laufgraph r_α)**.** Der Laufgraph r_α von $\mathcal{A}$ auf α wird induktiv, Ebene für Ebene, aus t_α durch das Löschen von Knoten v erzeugt, die mit q beschriftet sind und zu denen jeweils ein Knoten u auf gleicher Ebene existiert, der ebenfalls mit q beschriftet ist und weiter links angeordnet ist. In diesem Fall wird eine Kante von dem Vorgänger von v nach u hinzugefügt.

Offensichtlich wird das Eingabewort α von einem Büchi-Automaten $\mathcal{A}$ genau dann akzeptiert, wenn in dem Laufgraphen r_α ein unendlicher Pfad von der Wurzel existiert, auf dem immer wieder Zustände aus F auftreten.

Der Ausgangspunkt für die Transformation eines Büchi-Automaten $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ in einen äquivalenten Rabin-Automaten (der dieselbe ω-Sprache akzeptiert) ist die Benutzung einer endlichen Abstraktion für die unendlich vielen endlichen Präfixe der Berechnungsbäume. Wenn der Büchi-Automat die Präfixe $\alpha(0) \ldots \alpha(i-1)$ der Eingabe verarbeitet hat, kann der Berechnungsbaum bis zur Ebene i erzeugt werden. Nimmt man stattdessen den Laufgraphen, stellt man fest, dass eine Struktur mit endlicher Breite ausreicht, da jeder Knoten auf einer Ebene nur einmal vorkommen kann. Die Hauptaufgabe der Transformation von $\mathcal{A}$ in einen deterministischen endlichen ω-Automaten besteht darin, eine endliche Anzahl von Repräsentationen der unendlich vielen möglichen Präfixe von Laufgraphen zu finden, so dass es trotzdem noch möglich ist, die Existenz eines Pfades mit unendlich vielen F-Zuständen zu erkennen. Deshalb ist es notwendig, verschiedene Zweige des Berechnungsbaums (oder Laufgraphen) für die Erkennung von Endzuständen zu unterscheiden. Beide Algorithmen, die Verfahren von Safra und Muller-Schupp, benutzen Baumstrukturen für diesen Zweck. Ein Knoten in solch einem Baum beinhaltet die Informationen, welche Zustände aktuell in einem Zweig des Berechnungsbaums besucht worden sind; im Besonderen wird im Wurzelknoten gespeichert, welche Knoten überhaupt erreichbar sind (wie in der klassischen Potenzmengenkonstruktion). Beide Verfahren benutzen den Ansatz der Laufgraphen, der besagt, dass von einem Zustand nur das am weitesten links stehende Vorkommen behalten wird.

2.1.1 Muller-Schupp-Bäume

Zunächst werden die Baumstrukturen eingeführt, die von Muller und Schupp benutzt werden. Diese werden im Weiteren *Muller-Schupp-Bäume* genannt. Ein Muller-Schupp-Baum ist ein geordneter, strikter Binärbaum (alle Knoten mit Ausnahme der Blätter haben genau zwei Nachfolger), dessen Knoten mit natürlichen Zahlen benannt sind und die mit zwei Werten beschriftet sind: einerseits einer Teilmenge von Q und andererseits einer Farbe aus der Menge {rot, gelb, grün}. Durch die Konstruktion ist sichergestellt, dass die Menge der Zustände eines Knotens gleich der disjunkten Vereinigung der Zustandsmengen von beiden Nachfolgern ist. Dadurch würde es ausreichen, nur die Blätter mit Zuständen zu beschriften. Für eine bessere Lesbarkeit der Bäume werden alle Knoten mit den zugehörigen Zuständen beschriftet.

Die Muller-Schupp-Bäume können in drei Schritten eingeführt werden, beginnend mit dem Berechnungsbaum t_α des gegebenen Büchi-Automaten auf dem Eingabewort α.

Definition 2.6 (Akzeptierbaum t^1_α)**.** Der *Akzeptierbaum* t^1_α entsteht dadurch aus dem Berechnungsbaum t_α, dass die Nachfolger eines Knotens v in zwei Klassen aufgeteilt werden. Davon beinhaltet die eine die Endzustände und die andere die restlichen Zustände. Die ersteren werden in einer Menge zusammengefasst und ergeben die Beschriftung des linken Nachfolgers von v, die anderen bilden die Beschriftung des rechten Nachfolgers (natürlich kann einer der beiden Nachfolger auch verschwinden).

Der Akzeptierbaum t^1_α hat somit höchstens binäre Verzweigungen.

Lemma 2.7. *Der Berechnungsbaum t_α hat einen akzeptierenden Pfad genau dann, wenn der Akzeptierbaum t^1_α einen Pfad hat, der unendlich oft nach links verzweigt.*

Beweis.

$\Rightarrow$ Sei ρ ein akzeptierender Pfad in t_α. Wir verfolgen ρ von der Wurzel des Akzeptierbaums t^1_α aus. Ist ein Folgezustand Endzustand, so verzweigt der Pfad nach links, ansonsten nach rechts. Da ρ akzeptierend ist, werden unendlich oft Endzustände besucht und somit verzweigt der entsprechende Pfad in t^1_α auch unendlich oft nach links.

$\Leftarrow$ Der Akzeptierbaum t^1_α hat einen unendlichen, unendlich oft nach links verzweigenden Pfad ρ'. Wir betrachten den Teilbaum des Berechnungsbaums t_α, der zu diesem Pfad ρ' zusammengefasst worden ist. Dieser bildet einen unendlichen, aber endlich verzweigten Baum. Nach dem Lemma von König existiert in diesem Teilbaum ein unendlicher Pfad. Dieser ist aufgrund der Wahl des Teilbaums akzeptierend. □

Ausgehend von dem Akzeptierbaum t^1_α kann der *reduzierte Berechnungsbaum* t^2_α konstruiert werden:

Definition 2.8 (Reduzierter Berechnungsbaum t^2_α)**.** Der reduzierte Berechnungsbaum t^2_α entsteht aus dem Akzeptierbaum t^1_α, indem auf einer Ebene nur das Vorkommen eines Zustands q behalten wird, welches am weitesten links auf dieser Ebene des Baums t^1_α ist.

Der Baum t^2_α wächst in deterministischer Weise Ebene für Ebene mit der Eingabe α. Jede Ebene hat höchstens so viele Knoten, wie es Zustände im Automaten $\mathcal{A}$ gibt, also ist t^2_α von beschränkter Breite und hat weiterhin höchstens eine binäre Verzweigung.

Das Lemma 2.7 lässt sich auf den reduzierten Berechnungsbaum t^2_α übertragen:

Lemma 2.9. *Der Berechnungsbaum t_α hat einen akzeptierenden Pfad genau dann, wenn der Baum t^2_α einen Pfad hat, der unendlich oft nach links verzweigt.*

Beweis.

$\Leftarrow$ Angenommen, ein Pfad ρ ist nicht akzeptierend in t_α. Somit kann es auch keinen unendlich oft nach links verzweigenden Pfad im Akzeptierbaum t^1_α geben. Da t^2_α ein Teilbaum von t^1_α ist, gibt es auch hier keinen unendlich oft nach links verzweigenden Pfad.

$\Rightarrow$ Wenn der Berechnungsbaum t_α einen akzeptierenden Pfad hat, dann gibt es auch einen am weitesten links vorkommenden akzeptierenden Pfad. Somit gibt es auch einen am weitesten links vorkommenden akzeptierenden Pfad in dem nach dem ersten Schritt entstandenen Akzeptierbaum t^1_α. Dieser Pfad führt automatisch in jeder Ebene durch das linkeste Vorkommen der im zugehörigen Lauf des Automaten an dieser Stelle erreichten Zustände, da sonst ein noch weiter links liegender Pfad existieren würde. Somit bleibt der akzeptierende Pfad auch nach dem zweiten Schritt, der Konstruktion von t^2_α, erhalten. □

Die Idee besteht darin, komprimierte Versionen von den t^2_α-Präfixen als Zustände für den deterministischen Rabin-Automaten zu verwenden: ein Pfadsegment von einem linken Nachfolger (bzw. einem rechten) oder von der Wurzel bis zum nächsten Verzweigungspunkt v wird in dem Knoten v zusammengefasst. Hierdurch wird ein strikter Binärbaum erzeugt. Die Zustände auf einem solchen Pfadsegment werden vergessen, das Vorhandensein von Endzuständen auf diesem Pfadsegment wird hingegen über die Farbe des Knotens v gespeichert. Mögliche Farbwerte sind rot, gelb und grün. Es ist klar, dass die Anzahl solcher Bäume endlich ist.

Der Updateschritt, bei dem ein Eingabebuchstabe a verarbeitet wird (dies entspricht der Erweiterung von t_α^2 um eine Ebene), wird durchgeführt, indem Nachfolger gemäß der Potenzmengenkonstruktion zu den Blättern hinzugefügt werden, ausgehend von den Zustandsmengen, mit denen die Blätter beschriftet sind. Es wird kein Nachfolger hinzugefügt, wenn von der Zustandsmenge an dem Blatt und dem Buchstaben a kein neuer Zustand erreicht werden kann. Dieser Fall führt zu einer Löschung des Pfadsegments bis zum nächsten Verzweigungspunkt. Knotenbezeichnungen, die durch solch einen Vorgang wieder zur Verfügung stehen, können erst ab dem nächsten Updateschritt wieder benutzt werden. Im anderen Fall wird ein linker Nachfolger, ein rechter Nachfolger oder beides eingefügt, in Abhängigkeit davon, ob nur Endzustände, nur Nicht-Endzustände oder beide Arten in der resultierenden Zustandsmenge enthalten sind. Wenn ein Endzustand auftritt, wird der Knoten, der ihn beinhaltet, grün gefärbt. Durch die Strategie des Löschens (nur das am weitesten links vorkommende Auftreten eines Zustands wird behalten) und die Pfadkomprimierung kann es passieren, dass Pfadsegmente (genauer gesagt Baumknoten) zu einem einzigen Knoten zusammengefasst werden. Auch hier können freigewordene Namen im nächsten Schritt wieder benutzt werden. In diesem Fall kann ein Knoten Endzustände von einem Nachfolger erhalten, mit dem er zusammengefasst worden ist.

Ein Knoten wird rot gefärbt, wenn das Pfadsegment, für das er steht, keine Endzustände enthält. Er bekommt die Farbe gelb, wenn er Endzustände enthält, diese aber nicht im letzten Schritt bekommen hat. Grün wird er eingefärbt, wenn er einen Endzustand im letzten Schritt bekommen hat, sei es über die Potenzmengenkonstruktion oder beim Zusammenfassen mit z.B. einem Knoten der vorher gelb markiert war.

Die so entstehenden Bäume nennen wir Muller-Schupp-Bäume (kurz MS-Bäume).

Wird dieses Updateverfahren benutzt, gilt folgendes Lemma:

Lemma 2.10. *Der Baum t_α^2 hat genau dann einen unendlich oft nach links verzweigenden Pfad, wenn in der dazugehörigen Sequenz der Muller-Schupp-Bäume von einem Zeitpunkt an ein Knoten v immer enthalten und dieser Knoten unendlich oft grün gefärbt ist.*

Beweis.

$\Leftarrow$ Bleibt ein Knoten v in der Sequenz der Muller-Schupp-Bäume ab einem Zeitpunkt bestehen, beschreibt dieser Knoten einen Teil genau eines Pfades in dem Baum t_α^2. Ein Knoten wird grün gefärbt, wenn er im letzten Schritt einen Endzustand erhalten hat, sei es über die Potenzmengenkonstruktion (grün markierter Knoten) oder die Pfadkomprimierung (gelb markierter Nachfolger). Die grüne Farbmarkierung eines Knotens bedeutet, dass ein Endzustand besucht wurde und somit

der Pfad im Baum t^2_α links verzweigt. Da der Knoten v unendlich oft grün markiert ist, heißt dies, dass der dazugehörige Teil des Pfades im Baum t^2_α unendlich oft nach links verzweigt.

$\Rightarrow$ Sei Π ein Pfad in t^2_α, der unendlich oft nach links verzweigt. Gesucht wird ein Knoten v_Π in der Folge der Muller-Schupp-Bäume zu t^2_α, der ab einem Zeitpunkt (d. h. von einer t^2_α-Ebene an) immer vorhanden ist und unendlich oft grün gefärbt ist. Sei U die Menge der Knotennamen v, die schließlich immer vorkommen. Sei $i \in \mathbb{N}$ der Zeitpunkt, von dem an alle Knoten $v \in U$ immer vorhanden sind. Sei weiterhin v_Π ein Knoten aus U, dessen Anfangsknoten des repräsentierten Pfadsegments im reduzierten Berechnungsbaum t^2_α auf dem Pfad Π liegt und so dass für alle Nachfahren von v_Π im i-ten MS-Baum gilt, dass ihre repräsentierten Pfadsegmente nicht auf dem Pfad Π beginnen.

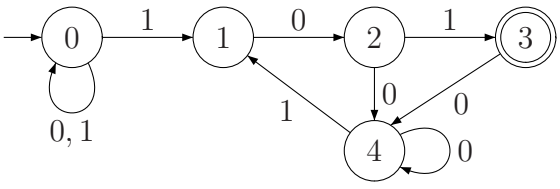
Annahme: v_Π wird nur endlich oft mit seinem Nachfolger zusammengefasst und ist somit nach einem Zeitpunkt $i' > i$ nicht mehr grün gefärbt. Da der Pfad Π unendlich oft nach links verzweigt, gibt es ein $i'' \geq i'$ mit $\Pi(i'') \in F$. Sei v der Knoten, dessen Pfadsegment mit $\Pi(i'')$ endet. v ist ein Nachfahre von v_Π im i''-ten Muller-Schupp-Baum. Sei v' der Nachfolger von v_Π auf dem Pfad von v_Π nach v. Sein Pfadsegment beginnt auch auf dem Pfad Π. Da v' nicht mit seinem Vorgänger zusammengefasst wurde, gilt, dass $v' \in U$ ist. Dies steht im Widerspruch zur Wahl von v_Π. □

Satz 2.11. *Ein (nicht-deterministischer) Büchi-Automat kann durch die Muller-Schupp-Konstruktion in einen deterministischen Rabin-Automaten transformiert werden.*

Beweis. Die Lemmas 2.7, 2.9 und 2.10 ergeben zusammen, dass ein Büchi-Automat genau dann ein Wort akzeptiert, wenn in der dazugehörigen Sequenz der Muller-Schupp-Bäume von einem Zeitpunkt an ein Knoten v immer enthalten und dieser Knoten unendlich oft grün gefärbt ist. Dies kann mit der Rabin-Akzeptierbedingung mit den Paaren (E_i, F_i) erfasst werden, wobei i über die Werte der endlichen Knotennamen läuft: E_i beinhaltet die Bäume, in denen der Knoten i fehlt, und F_i genau die Bäume, in denen i grün gefärbt ist. □

Formal ist der Updateschritt für einen Muller-Schupp-Baum t und einen Eingabebuchstaben a im Algorithmus 2.1 definiert.

Beispiel 2.12. Sei $\mathcal{A}_0$ folgender Büchi-Automat:



1. Kopiere den gegebenen Baum t, färbe dabei alle grün gefärbten Knoten gelb.
2. Wende Potenzmengenkonstruktion (mit Buchstaben a) auf jedes Blatt an, füge linken Nachfolger mit erreichten Endzuständen (grün) und rechten Nachfolger mit Nicht-Endzuständen (rot) hinzu.
3. Behalte nur das am weitesten links vorkommende Auftreten eines Zustands.
4. Lösche die Knoten auf einem Pfad, der nur zu einem Blatt mit leerer Menge führt.
5. Solange Knoten mit genau einem Nachfolger existieren, fasse sie mit ihrem Nachfolger zusammen, erbe die Farbe Grün, wenn der Nachfolger grün oder gelb gefärbt war.
6. Erzeuge Bottom-Up die Beschriftungen der Knoten im Baum. Die Beschriftung eines Knotens ist die Vereinigung der Beschriftungen (entspricht Zustandsmengen) von beiden Nachfolgern.

Algorithmus 2.1: Updateschritt für Muller-Schupp-Bäume

$\mathcal{A}_0$ akzeptiert die 0-1 Folgen, die das Segment 11 nur endlich oft enthalten, aber 101 unendlich oft. Der Laufgraph von $\mathcal{A}_0$ auf dem Eingabewort 1010101 . . . ist auf der linken Seite der Abbildung 2.2 dargestellt. In Abbildung 2.1 sind die Muller-Schupp-Bäume (MS-Bäume) rechts für dieses Eingabewort dargestellt, mit Zwischenschritten für die Berechnung links davor. Jeder Knoten wird mit einer Zahl bezeichnet, nach einem senkrechten Strich folgen die Zustände, die zu diesem Knoten gehören. In einem Muller-Schupp-Baum sind rot gefärbte Knoten durch ein gestricheltes Rechteck dargestellt, die gelben Knoten durch ein normales und die grünen durch ein doppeltes Rechteck.

2.1.2 Safra-Bäume

Die Safra-Bäume kann man als komprimierte Versionen von Muller-Schupp-Bäumen ansehen, in denen die Speicherung von Nicht-Endzuständen vermieden wird. Ausgehend vom Updateschritt des Muller-Schupp-Algorithmus führt die Safra-Konstruktion eine technische Vereinfachung ein, wenn die Potenzmengenkonstruktion angewendet wird: nur der linke Nachfolger eines Knotens v, der die Endzustände beinhaltet, verbleibt in dem Baum, es wird also kein Knoten für die Nicht-Endzustände eingeführt. Wird von diesen Nicht-Endzuständen später ein Endzustand erreicht, wird ein neuer Nachfolger von v in den Safra-Baum aufgenommen. In diesem Fall können mehr als binäre Verzweigungen auftreten. In einem Muller-Schupp-Baum ergeben die dazwischen-

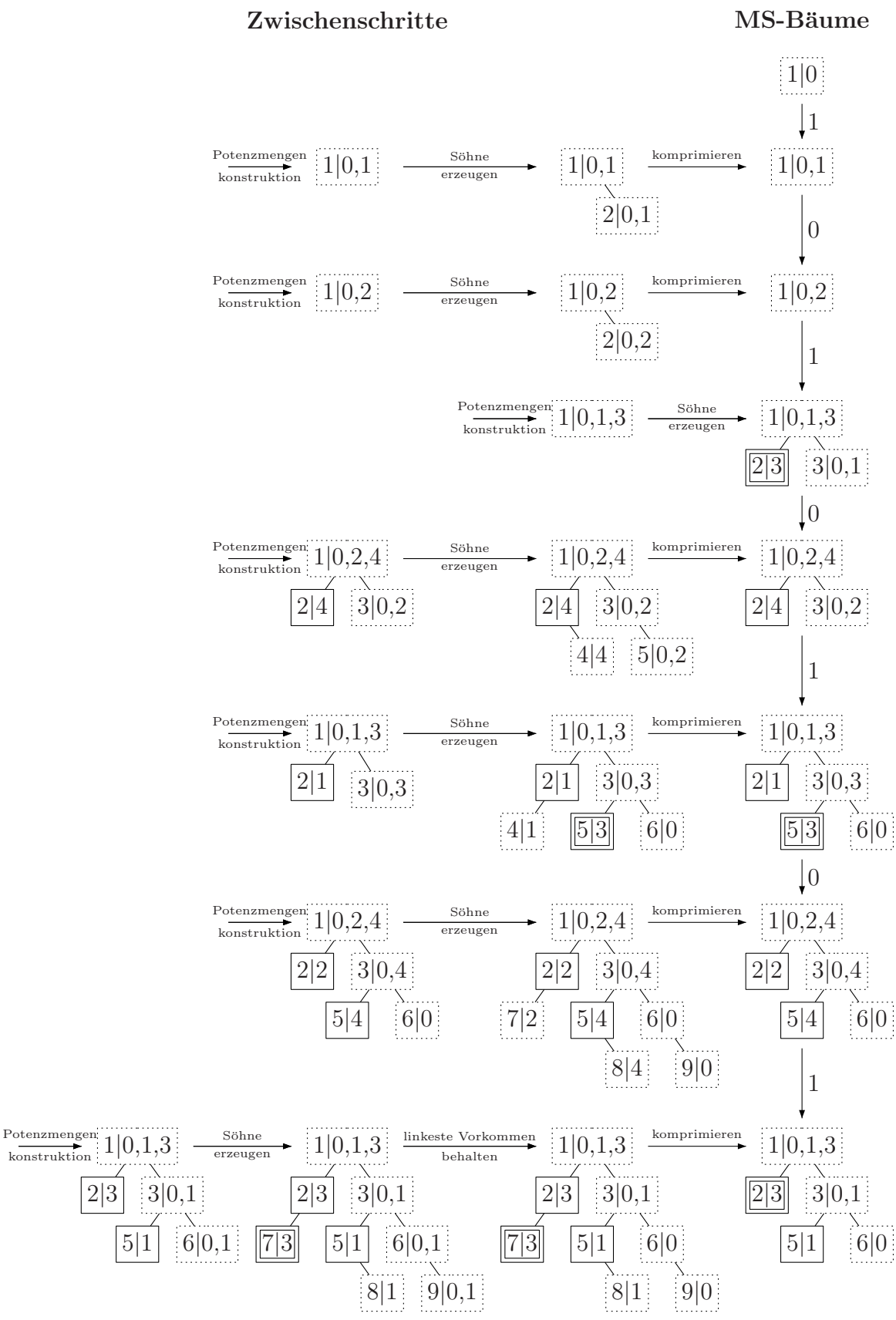


Abbildung 2.1: MS-Bäume (mit Zwischenschritten) des Automaten $\mathcal{A}_0$ für die Eingabe 1010101

liegenden Knoten mit Nicht-Endzuständen eine binäre Kodierung der Safra-Bäume. Allerdings kann die Einbettung der Safra-Bäume in die korrespondierenden Muller-Schupp-Bäume aufgrund der unterschiedlichen Farbkodierung in der Regel nicht auf eine Einbettung des Safra-Zustandsraums in den Muller-Schupp-Zustandsraums fortgesetzt werden. Dies folgt bereits daraus, dass ein Safra-Automat größer werden kann als der entsprechende Muller-Schupp-Automat (siehe dazu auch die Anmerkung am Ende von Abschnitt 2.3). Ein didaktischer Vorteil der Muller-Schupp-Bäume ist, dass sie mehr der Struktur des Berechnungsbaums des gegebenen Büchi-Automaten entsprechen.

Der Unterschied in der Färbung spiegelt sich in der unterschiedlichen Speicherung von Endzustandsbesuchen wider. Der Muller-Schupp-Algorithmus nutzt die Färbungsstrategie, um den Besuch von neuen Endzuständen zu signalisieren. Der Safra-Algorithmus hingegen markiert einen Knoten grün gemäß der so genannten Breakpoint-Konstruktion, wenn alle Zustände eines Knotens über Endzustände in der Vergangenheit erreicht werden können. Analog zu den Muller-Schupp-Bäumen ist ein Lauf akzeptierend, wenn ein Knoten von einem Zeitpunkt an immer bestehen bleibt und dabei von Zeit zu Zeit grün gefärbt ist. Formal ist die Aktualisierung eines Safra-Baums t für einen Eingabebuchstabens a gemäß [Saf88] im Algorithmus 2.2 definiert.

1. Kopiere den Baum t, entferne alle grünen Markierungen.
2. Für alle Knoten v des Baums erzeuge einen jüngsten Nachfolger, der alle Endzustände von v enthält, wenn solche existieren.
3. Wende Potenzmengenkonstruktion (mittels Buchstaben a) auf alle Knoten an.
4. Behalte nur das am weitesten links vorkommende Auftreten eines Zustands.
5. Entferne alle Knoten mit leerer Beschriftung.
6. Für jeden Knoten v, dessen Beschriftung gleich der Vereinigung der Beschriftungen der Nachfolger ist, entferne alle Nachkommen von v und färbe v grün.

Algorithmus 2.2: Updateschritt für Safra-Bäume

Beispiel 2.13. Sei $\mathcal{A}_0$ wieder der Büchi-Automat von Beispiel 2.12. In Abbildung 2.2 auf der nächsten Seite ist der Laufgraph für die Eingabe 10101010 . . . dargestellt, rechts daneben der Lauf der Safra-Bäume und dann der Lauf der Muller-Schupp-Bäume. Zwischenschritte in der Konstruktion zwischen zwei aufeinander folgenden Bäumen sind hier weggelassen worden. Benutzt wird dieselbe Notation wie beim Beispiel 2.12.

Analog dazu sind die Knoten eines Safra-Baums, die grün markiert sind, mit einem doppelten Rechteck dargestellt.

2.2 Optimierung des Muller-Schupp-Algorithmus

In der experimentellen Erprobung der beiden Konstruktionen ergibt sich, dass Muller-Schupp-Bäume aufgrund der Einbeziehung von Knoten, deren Beschriftungen aus Nicht-Endzuständen bestehen, dazu tendieren, größer als die entsprechenden Safra-Bäume zu werden. Einen noch gravierenderen Effekt allerdings hat die Art und Weise des Einfügens von neuen Nachfolgern und wie diese benannt werden. So werden viele Bäume erzeugt, die dieselbe Struktur in den Knotenbeschriftungen (und Farben) haben, sich aber in den Knotenbezeichnungen unterscheiden. Eine Idee, um Knotenbezeichnungen zu sparen, besteht darin, neue Nachfolger für Blätter nur dann hinzuzufügen, wenn sowohl Endzustände als auch Nicht-Endzustände in der Knotenbeschriftung nach der Potenzmengenkonstruktion existieren. Dies führt zu einer restriktiveren Art der Einführung von Knotenbezeichnungen. Formal ist diese Modifikation des Muller-Schupp-Algorithmus im Algorithmus 2.3 definiert.

1. Kopiere den gegebenen Baum t, färbe dabei alle grün gefärbten Knoten gelb.
2. Wende die Potenzmengenkonstruktion (mit Buchstaben a) auf jedes Blatt an.
3. Behalte nur das am weitesten links vorkommende Auftreten eines Zustands.
4. Für alle Blätter, die sowohl Endzuständen als auch Nicht-Endzustände beinhalten, füge einen linken Nachfolger mit den Endzuständen und rechten Nachfolger mit den Nicht-Endzuständen hinzu; färbe diese Nachfolger grün bzw. rot.
5. Färbe alle Blätter, die nur Endzustände enthalten, grün.
6. Schritte 4., 5. und 6. des Originalalgorithmus

Algorithmus 2.3: Optimierter Updateschritt eines Muller-Schupp-Baums

Satz 2.14. *Der Muller-Schupp-Algorithmus gemäß Algorithmus 2.3 erzeugt einen zum gegebenen Büchi-Automaten sprachäquivalenten Rabin-Automaten.*

Beweis. Auch für die Version 2.3 des Muller-Schupp-Algorithmus lässt sich das Lemma 2.10 zeigen. Die neue Variante der Aktualisierung eines Muller-Schupp-Baums unterscheidet sich von der ursprünglichen nur in den Namen der neu hinzugefügten Knoten,

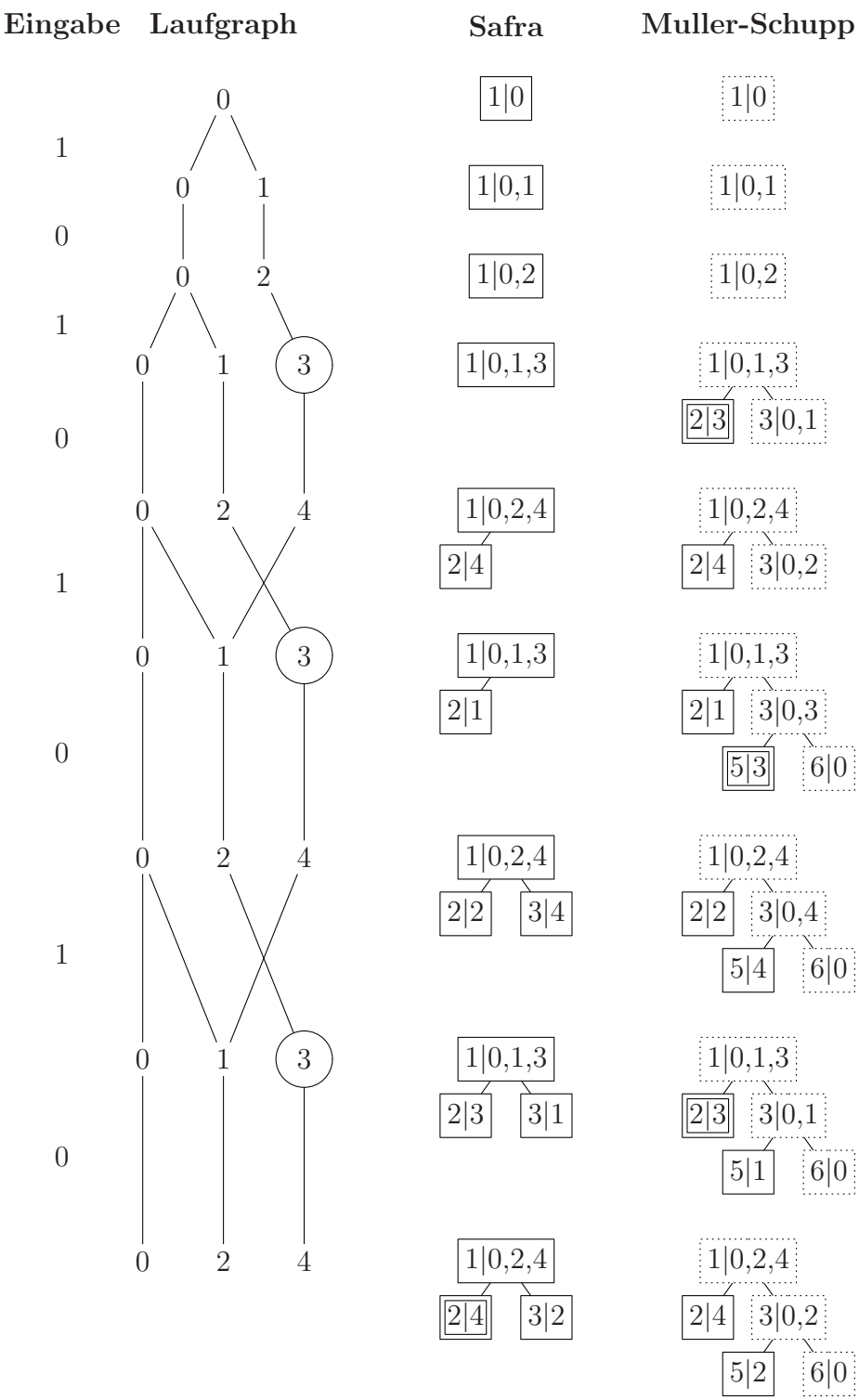
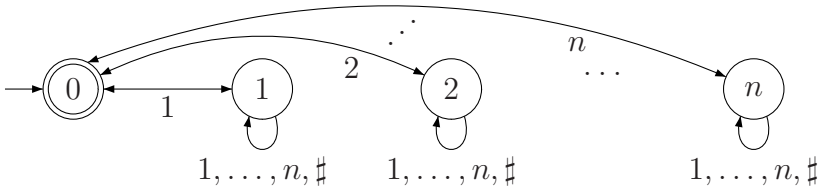


Abbildung 2.2: Vergleich zwischen Laufgraph und der Sequenz der Safra- und Muller-Schupp-Bäume für die Eingabe 10101010

bedingt durch die restriktivere Art des Hinzufügens neuer Knoten. Der Unterschied kommt dann zum Tragen, wenn die Potenzmengenkonstruktion entweder nur Endzustände oder nur Nicht-Endzustände liefert. In diesem Fall werden bei der neuen Variante keine neuen Knoten im Baum hinzugefügt, die im fünften Schritt des ursprünglichen Algorithmus direkt wieder entfernt worden wären. Diese so gesparten Knotennamen können dann für andere hinzuzufügende Knoten benutzt werden. Da diese eingesparten Knotennamen nicht im ursprünglichen Baum vorhanden sind, lässt sich der Beweis des Lemmas 2.10 übernehmen. Somit kann auch der Beweis von Satz 2.11 übernommen werden, um zu zeigen, dass der optimierte Muller-Schupp-Algorithmus zu einem gegebenen Büchi-Automaten einen sprachäquivalenten Rabin-Automaten erzeugt. □

2.3 Vergleich der Algorithmen

Im Rahmen der Experimentierplattform GASt (siehe auch Kapitel 5) sind die vorgestellten Algorithmen auf verschiedene Büchi-Automaten angesetzt worden, darunter auch auf die Automaten, die von Michel [Mic88] vorgeschlagen worden sind, um die untere Schranke für die Komplementierung von Büchi-Automaten zu zeigen (siehe auch [PP04] und [Tho97]). Die Büchi-Automaten $\mathcal{M}_n$, die Michel in [Mic88] vorgeschlagen hat, haben die Zustände $0, \ldots, n$, jeweils das Eingabealphabet $\{1, \ldots, n, \sharp\}$ und den folgenden Transitionsgraphen:



Während der entsprechende Rabin-Automat mit der Safra-Konstruktion problemlos bis zum Automaten $\mathcal{M}_5$ (bei dem eine Millionen Zustände zum ersten Mal erreicht werden) berechnet werden kann, hat der Rabin-Automat gemäß der Muller-Schupp-Konstruktion bereits über fünf Millionen Zustände für $\mathcal{M}_3$ – der verwendete Computer hatte nicht genug Speicher zur Berechnung von $\mathcal{M}_4$ (siehe auch Tabelle 2.1).

Die optimierte Variante des Muller-Schupp-Algorithmus kann viele Zustände im deterministischen Rabin-Automaten einsparen, wie die Tabelle 2.1 zeigt. Trotzdem wächst die Anzahl der Muller-Schupp-Bäume deutlich schneller als die Anzahl der Safra-Bäume.

Weitere Fallstudien sind in der Diplomarbeit von Christoph Schulte Althoff ([Alt05]) zu finden. Es ist nicht möglich, einen einfachen Zusammenhang bezüglich der Zustandsanzahl zwischen beiden Algorithmen herzustellen, da es viele Fälle gibt, in de-

Automat		Safra		Muller-Schupp		Opt. Muller-Schupp	
Name	Zus.	Zustände	Paare	Zustände	Paare	Zustände	Paare
$\mathcal{M}_1$	2	7	1	9	5	9	5
$\mathcal{M}_2$	3	33	2	4,058	8	262	7
$\mathcal{M}_3$	4	385	5	4,823,543	11	23,225	9
$\mathcal{M}_4$	5	13,601	7	kein Speicher mehr		3,656,802	11
$\mathcal{M}_5$	6	1,059,057	9	kein Speicher mehr		kein Speicher mehr	

Tabelle 2.1: Vergleich der Algorithmen von Safra, Muller-Schupp und des optimierten Muller-Schupp-Algorithmus

nen der Safra-Algorithmus deutlich kleinere Automaten liefert als die Muller-Schupp-Konstruktion. Auf der anderen Seite gibt es auch Beispiele, bei denen der Muller-Schupp-Algorithmus kleinere Automaten konstruiert. Dies tritt zum Beispiel für den Automaten $\mathcal{A}_1$ auf, der zur Vorstellung der Experimentierplattform GASt in Abschnitt 5.1.1 auf Seite 93 benutzt wird, bei dem die Safra-Konstruktion vier Zustände benötigt und die Muller-Schupp-Konstruktion (sowohl die normale als auch die optimierte) mit zwei Zuständen auskommt. Die verzögerte Signalisierung des "Erfolges" (durch eine verlängerte Initialisierung) führt dazu, dass unterschiedliche Safra-Bäume, die sich nur in der Färbung unterscheiden, für einen einzelnen Muller-Schupp-Baum erzeugt werden. Eine Möglichkeit, Zustände beim Safra-Algorithmus einzusparen, besteht darin, die Anwendung der Potenzmengenkonstruktion und die Erzeugung der Söhne zu vertauschen; dieser Ansatz wird in [PP04] verfolgt. In dem gerade erwähnten Beispiel wird dadurch ein Zustand eingespart. Allerdings führt dies immer noch zu einem größeren Automaten als bei der Muller-Schupp-Konstruktion. Die Erfahrungen haben insgesamt gezeigt, dass für praktische Anwendungen die Konstruktion von Safra zu bevorzugen ist.

Kapitel 3

Unendliche Zwei-Personen-Spiele

Nachdem im letzten Kapitel die Grundlage für die Spezifikation von Gewinnbedingungen für unendliche Zwei-Personen-Spiele durch die Determinisierung von ω-Automaten (im konkreten Fall Büchi-Automaten) geschaffen worden ist, sollen in diesem Kapitel die Grundlagen dieser Spiele eingeführt und im Weiteren insbesondere die Request-Response-Spiele genauer betrachtet werden. Dabei wird die Terminologie von unendlichen Zwei-Personen-Spielen benutzt, wie sie in [Tho95, PP04] zu finden ist.

3.1 Einführung

Unendliche Zwei-Personen-Spiele stellen ein geeignetes Rahmenwerk zur Spezifikation von reaktiven Systemen [PR89] dar, da sich ein reaktives System in ständiger Interaktion mit seiner Umgebung [HP85] befindet. Um die Interaktion zwischen dem Controller und der Umgebung zu modellieren, wird das gesamte System als Spielgraph eines unendlichen Zwei-Personen-Spiels betrachtet. Ein Spielgraph ist dabei ein Transitionsgraph, wobei jeder Zustand genau einem der beiden Spieler zugeordnet ist – in diesem Punkt ist das Modell des Spielgraphen eine Erweiterung des üblichen Automatenmodells. Das Programm des Controllers ist dabei Spieler 0 und die Umgebung ist Spieler 1 zugeordnet. Während einer Partie wählt immer der Spieler den Nachfolgezustand aus, zu dem der aktuelle Zustand der Partie gehört.

Definition 3.1 (Spielgraph)**.** Ein Spielgraph G ist ein Tupel (Q, E) mit Zustandsmenge $Q = Q_0 \dot\cup Q_1$, Kantenrelation $E \subseteq Q \times Q$, so dass für alle $q \in Q$ gilt $qE \neq \emptyset$ (d. h. jeder Knoten $q \in Q$ hat eine ausgehende Kante).

Solange nichts anderes explizit erwähnt wird, werden in dieser Arbeit immer Spielgraphen betrachtet, bei denen die Zustandsmenge Q endlich ist.
Eine Partie ρ ist eine Folge von Zuständen $\rho = \rho_0\rho_1\rho_2 \dots$ mit $(\rho_i, \rho_{i+1}) \in E$. Für eine unendliche Partie ρ definiert $Oc(\rho)$ die Menge der in ρ vorkommenden Zustände und $In(\rho)$ die Menge der unendlich oft besuchten Zustände.

Die Spezifikation eines unendlichen Zwei-Personen-Spiels wird durch die Gewinnbedingung vervollständigt, die Spieler 0 erfüllen muss, um zu gewinnen. Typische Gewinnbedingungen sind die Erreichbarkeits-, Sicherheits-, Staiger-Wagner- [SW74], Büchi- [Büc62], Muller- [Mul63], Paritäts- [Mos84], Rabin- [Rab69, Rab72] oder die Streett-Bedingung [Str82]. In Tabelle 3.1 sind die Bedeutungen dieser Bedingungen aufgeführt.

Name	**gegeben**	**Spieler 0 gewinnt, wenn**
Erreichbarkeit	$F \subseteq Q$	$Oc(\rho) \cap F \neq \emptyset$
Sicherheit	$F \subseteq Q$	$Oc(\rho) \subseteq F$
Staiger-Wagner	$\mathcal{F} = \{F_1, \ldots, F_n\}$	$Oc(\rho) \in \mathcal{F}$
Büchi	$F \subseteq Q$	$In(\rho) \cap F \neq \emptyset$
Parität	$c : Q \to \{0, \ldots, k\}$	$\max(In(c(\rho)))$ ist gerade
Muller	$\mathcal{F} = \{F_1, \ldots, F_n\}$	$In(\rho) \in \mathcal{F}$
Rabin	$\Omega = ((E_1, F_1), \ldots, (E_r, F_r))$	$\bigvee_{i=1}^{r} (In(\rho) \cap E_i = \emptyset \wedge In(\rho) \cap F_i \neq \emptyset)$
Streett	$\Omega = ((E_1, F_1), \ldots, (E_r, F_r))$	$\bigwedge_{i=1}^{r} (In(\rho) \cap F_i \neq \emptyset \to In(\rho) \cap E_i \neq \emptyset)$

Tabelle 3.1: Typische Gewinnbedingungen für unendliche Zwei-Personen-Spiele

Die Gewinnbedingung spezifiziert also, wann Spieler 0 eine Partie ρ gewinnt. Wenn Spieler 0 erzwingen kann, dass er von einem Startzustand q aus gewinnt, wird gesagt, dass Spieler 0 von q aus gewinnt. Eine Strategie von einem Spieler bestimmt den folgenden Zug ausgehend von einem seiner Zustände anhand der bisher gespielten Partie.

Definition 3.2. (a) Eine *Strategie* für Spieler 0 von Knoten q ist eine Funktion $f : Q^*Q_0 \to Q$ die jedem Partiepräfix $\rho_0 \ldots \rho_k$ mit $\rho_0 = q$ und $\rho_k \in Q_0$ einen Knoten $r \in Q$ zuordnet mit $(\rho_k, r) \in E$.

(b) Eine Partie $\rho = \rho_0\rho_1 \ldots$ von $\rho_0 = q$ heißt gemäß der Strategie f gespielt, wenn für jedes $\rho_i \in Q_0$ gilt $\rho_{i+1} = f(\rho_0 \ldots \rho_i)$.

(c) Die Strategie f ist eine *Gewinnstrategie* für Spieler 0 von q aus, falls von q aus jede gemäß f gespielte Partie von Spieler 0 gewonnen wird. Analog gilt diese Definition für Spieler 1.

Eine Gewinnstrategie ist *positional*, wenn die Entscheidung für den nächsten Zug nur von dem aktuellen Zustand abhängt. Eine andere Möglichkeit ist, dass ein endlicher Speicher zur Realisierung einer Strategie benötigt wird. Solche Strategien lassen sich

durch endliche Automaten mit Ausgabe darstellen. Dabei entspricht die Ausgabe der Knotenauswahl des Spielers. Derartige Gewinnstrategien werden auch *Automatengewinnstrategien* genannt.

Definition 3.3 (Strategieautomat)**.** Ein *Strategieautomat* für Spieler 0 im Spiel über $G = (Q, E)$ sei ein endlicher Automat mit Ausgabe der Form $\mathcal{A} = (S, Q, s_0, \sigma, \tau)$ mit:

- S als endlicher Speicher,
- Q als endliches Eingabealphabet,
- $s_0 \in S$ als Anfangszustand,
- σ als Speicherupdate $\sigma : S \times Q \to S$,
- τ als Transitionsauswahl $\tau : S \times Q_0 \to Q$.

Mit $\tau : S \times Q_1 \to W$ kann der Strategieautomat analog für Spieler 1 definiert werden.

Die Lösung eines Spiels besteht darin, die Zustände zu bestimmen, von denen aus Spieler 0 eine Gewinnstrategie hat und diese Strategie auch konstruiert wird.

Definition 3.4 (Gewinnbereiche)**.** Zu einem Spielgraphen G und einer Gewinnmenge Win_0 ($Win_0 \subseteq Q^\omega$ - Gewinnpartien von Spieler 0) sei

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

Analog ist W_1 für Spieler 1 definiert.

Es gibt keinen Zustand $q \in Q$, von dem aus beide Spieler eine Gewinnstrategie haben. Wähle entsprechende Gewinnstrategien f, g von q aus für beide Spieler. Da f eine Gewinnstrategie für Spieler 0 ist, folgt für die entstehende Partie ρ, dass $\rho \in Win_0$. Da g eine Gewinnstrategie für Spieler 1 ist, gilt $\rho \notin Win_0$, Widerspruch. Daraus folgt die Eigenschaft, dass $W_0 \cap W_1 = \emptyset$ gilt (vgl. [Tho97]).

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 Arbeit betrachteten Gewinnbedingungen beschreiben reguläre ω-Sprachen, also Borelmengen, und sind somit nach [Mar75] determiniert.

Im Fall der reaktiven Systeme legt die Spezifikation fest, welche Systemläufe, die durch die beiden Komponenten Controller (Spieler 0) und Umgebung (Spieler 1) aufgebaut werden, "korrekt" sind. Ein adäquates abstraktes Modell für diese Situation sind die unendlichen Zwei-Personen-Spiele, deren Partien als unendliche Wörter interpretiert

werden und deren Gewinnbedingung (für den Spieler "Controller") durch eine Menge Win_0 unendlicher Wörter beschrieben wird. Das Syntheseproblem für solch ein reaktives System ist die Frage nach einer Gewinnstrategie für den Controller. Die Theorie der reaktiven Systeme ist effektiv; denn wenn die Spezifikation implementierbar ist, kann ein Zeuge zurückgegeben werden, der eine Gewinnstrategie für alle Zustände ist, von denen Spieler 0 die Spezifikation garantieren kann. Mit anderen Worten ausgedrückt: Die Konstruktion einer Gewinnstrategie für den Controller ist gleichbedeutend mit der Synthese des Controllers.

Die einfachste Gewinnbedingung ist die der Erreichbarkeit. Spieler 0 gewinnt eine Partie ρ in einem Erreichbarkeitsspiel, wenn die Partie ρ einen Zustand aus der gegebenen Menge F aufsucht. Ein solches Spiel kann mit der Attraktor-Berechnung gelöst werden. In dieser werden induktiv über i die Mengen $Attr_0^i(F)$ bestimmt, die jeweils diejenigen Zustände enthalten, von denen Spieler 0 den Besuch eines Zustands $q \in F$ in maximal i Zügen erzwingen kann.

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

Der Gewinnbereiche der beiden Spieler sind $W_0 = Attr_0(F)$ und $W_1 = Q \setminus W_0$. In der Literatur findet sich für $Attr_0(F)$ auch manchmal die Bezeichnung $Reach_0(F)$. $Attr_1(F)$ ist analog zu $Attr_0(F)$ mit dem Unterschied definiert, dass jeweils die Mengen Q_0 und Q_1 vertauscht sind.

Beide Spieler haben auf ihrem jeweiligen Gewinnbereich eine positionale Gewinnstrategie. Diese so genannte (positionale) *Attraktorstrategie* [GTW02] basiert auf der Distanz zur Menge F, die wie folgt formal definiert ist:

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

Die Attraktorstrategie für Spieler 0 besteht darin, die Distanz zur Menge F zu reduzieren und für Spieler 1 darin, die Zustände aus $Attr_0(F)$ zu vermeiden:

Spieler 0 : Für $q \in Attr_0^{i+1}(F) \setminus Attr_0^i(F)$ wähle Transition nach $Attr_0^i(F)$

Spieler 1 : Für $q \in Q \setminus Attr_0(F)$ wähle Transition nach $Q \setminus Attr_0(F)$

Eine weitere für diese Arbeit wichtige Gewinnbedingung ist die Büchi-Bedingung. Spieler 0 gewinnt eine Partie ρ in einem Büchi-Spiel, wenn die Partie ρ unendlich oft Zustände aus der gegebenen Menge F besucht. Die Idee zur Lösung eines Büchi-Spiels besteht darin, Mengen $Recur_0^i$ zu bestimmen, die genau die Zustände $q \in F$ beinhalten, von denen Spieler 0 mindestens i Wiederbesuche in F erzwingen kann. Zur Vorbereitung wird dazu der Attraktor$^+$ als eine Abwandlung des Attraktors definiert:

$$\begin{aligned}
Attr_0^+(F) &:= \{q \in Q \mid \text{Spieler 0 kann von } q \text{ Besuch in F in } \geq 1 \text{ Schritten erzwingen}\} \\
A_0^0 &:= \emptyset \\
A_0^1 &:= \{q \in Q_0 \mid \exists (q,r) \in E : r \in F\} \cup \{q \in Q_1 \mid \forall (q,r) \in E : r \in F\} \\
i \geq 1 : A_0^{i+1} &:= A_0^i \cup \{q \in Q_0 \mid \exists (q,r) \in E : r \in A_0^i\} \cup \\
&\quad \{q \in Q_1 \mid \forall (q,r) \in E : r \in (A_0^i \cup F)\} \\
Attr_0^+(F) &:= \bigcup_{i=0}^{|Q|} A_0^i
\end{aligned}$$

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

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

Die Gewinnbereiche der beiden Spieler in einem Büchi-Spiel sind $W_0 = Attr_0(Recur_0(F))$ und $W_1 = Q \setminus W_0$. Beide Spieler haben eine positionale Gewinnstrategie auf ihrem jeweiligen Gewinnbereich. Auf $Recur_0(F) \cap Q_0$ ist eine Kantenauswahl zurück nach $Recur_0(F)$ gemäß der Attraktorstrategie möglich. Allerdings ist die Attraktorstrategie keine Gewinnstrategie für Spieler 1, da Endzustände in der Menge $Q \setminus Attr_0(Recur_0(F))$ enthalten sein können. Somit reicht es nicht aus die Menge $Attr_0(Recur_0(F))$ zu vermeiden. Um die positionale Gewinnstrategie von Spieler 1 berechnen zu können, muss zunächst ein Sicherheitsspiel (komplementär zum Erreichbarkeitsspiel) von Spieler 1 für die Menge $Q \setminus F$ gelöst werden. Spieler 1 gewinnt dieses Spiel genau dann, wenn die Zustandsmenge $Q \setminus F$ nicht verlassen wird. Sei W_1' der Gewinnbereich von Spieler 1 in diesem Sicherheitsspiel. Dann ist $Attr_1(W_1')$ gerade genau der Gewinnbereich W_1 im gegebenen Büchi-Spiel. Die dazugehörige positionale Gewinnstrategie entspricht der Attraktorstrategie für Zustände $q \in Attr_1(W_1') \setminus W_1'$ und Spieler 1 spielt die positionale Gewinnstrategie des Sicherheitsspiels für Zustände $q \in W_1'$.

3.1.1 Symbolischer Zustandsraum

Bisher sind die unendlichen Zwei-Personen-Spiele über einem enumerativen Zustandsraum betrachtet worden, bei dem jeder Zustand separat betrachtet worden ist. Bei dem enumerativen Zustandsraum besteht das Problem, dass die Anzahl der Zustände, welche zur Modellierung eines realen Systems benötigt werden, und der dadurch benötigte Speicherplatz oftmals größer ist, als dass ein Computer damit umgehen könnte – auch wenn sich diese Grenze durch immer leistungsfähigere Computer immer weiter verschiebt. Dieses Problem wird auch das "State Explosion Problem" genannt. Eine wichtige Technik zur Reduktion der Auswertungsaufgabe im Model-Checking ist die "Partial Order Reduction" oder die Verwendung von Symmetrien ([CGP99]). Der Ansatz des symbolischen Zustandsraums stellt eine Möglichkeit dar, dieses Problem zu verringern. Der Grundgedanke ist vergleichbar mit dem symbolischen Model-Checking im Gegensatz zum klassischen Model-Checking, wie in [BCM$^+$90] beschrieben. Im symbolischen Ansatz wird der Zustandsraum durch eine Anzahl boolescher Aussagevariablen gegeben. Zustände sind dann konkrete Belegungen dieser Variablen. Die Transitionen im Zustandsgraphen sowie Mengen von Zuständen werden durch boolesche Funktionen über diesen Variablen definiert. Der Vorteil besteht darin, dass nicht mehr jeder Zustand bzw. jede Transition separat betrachtet werden und dadurch Speicher benötigen, sondern dass auf Mengen von Zuständen bzw. Transitionen gearbeitet wird, die effizienter im Speicher des Computers angelegt werden können. Als Repräsentation für booleschen Funktionen werden aus Effizienzgründen nicht boolesche Ausdrücke, sondern binäre Entscheidungsgraphen, kurz BDDs (Binary decision diagrams) benutzt. Diese wurden erstmals 1959 von Lee [Lee59] eingeführt und später von Akers [Ake78] und Moret [Mor82] weiter entwickelt. Durch die Hinzunahme von zusätzlichen Ordnungsbedingungen sowie Reduktionsmechanismen verbesserte Bryant [Bry86, Bry92] diesen Ansatz zur Darstellung boolescher Funktionen. Durch die BDDs ist eine kompakte Darstellung und effiziente Möglichkeit zur Manipulation boolescher Funktionen gegeben (vgl. [CGP99]).

3.2 Request-Response-Spiele

Ein wichtige Gewinnbedingung für unendliche Zwei-Personen-Spiele ist die Request-Response-Bedingung, die in der Literatur auch manchmal Assume-Guarantee-Bedingung genannt wird. Sie ähnelt der im letzten Abschnitt erwähnten Streett-Bedingung, die eine Konjunktion folgender Form ist (mit Zustandsmengen P_i, R_i):

$\bigwedge_i$ wird ein P_i-Zustand immer wieder besucht, so auch ein R_i-Zustand.

Diese "Fairness-Bedingung" spart den in der Praxis wichtigen Fall aus, dass ein P_i-Zustand nur endlich oft, z.B. einmal, besucht wird. Die Request-Response-Bedingung berücksichtigt diesen Fall und fordert für jeden Besuch in P_i einen nachfolgenden Besuch in R_i. Damit entspricht sie einem Standardfall in Spezifikationen, in denen ein System etwa auf bestimmte Signale reagieren soll.

Definition 3.5 (Request-Response-Spiel - [WHT03])**.** 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

$$\rho \in \mathit{Win}_0 :\Leftrightarrow \bigwedge_{i=1}^{r} (\forall j\, (\rho(j) \in P_i \Rightarrow \exists j'\, (j' \geq j \ \wedge \rho(j') \in R_i)))$$

In LTL[1] ausgedrückt ergibt sich die kürzere Formulierung (mit Zustandsprädikaten P_i, R_i):

$$\rho \in \mathit{Win}_0 :\Leftrightarrow \bigwedge_{i=1}^{r} \mathrm{G}(P_i \rightarrow \mathrm{F}\ R_i)$$

In Zukunft kürzen wir "Request-Response" häufig durch "RR" ab. Eine RR-Partie wird von Spieler 0 gewonnen, wenn gilt, dass nach einem Besuch der Anforderungsmenge (P_i) irgendwann später ein Zustand aus der passenden Antwortmenge (R_i) gesehen wird (wobei "später" auch die jeweilige Gegenwart einschließt). Dabei heißt eine RR-Bedingung zu einem Zeitpunkt t einer Partie offen oder aktiv, wenn in der Partie ein Zustand aus der Anforderungsmenge besucht worden ist, aber anschließend (bis t) noch keiner aus der Antwortmenge. Der Besuch der P_i-Menge wird auch Aktivierung, und der anschließende Besuch der R_i-Menge Erfüllung der RR-Bedingung genannt.

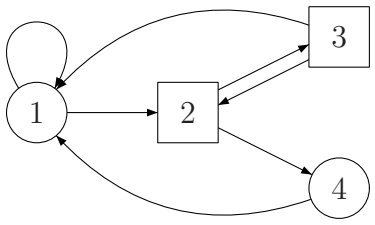
Die Bedeutung der Request-Response-Bedingung liegt darin begründet, dass sie eine einfache Form der Liveness-Bedingung darstellt, die eine wichtige und somit häufig vorkommende Anforderung an reaktive Systeme ist. Somit ist es möglich, eine große

[1] Linear Temporal Logic [Eme90]

Anzahl praxisbezogener Spezifikationen von reaktiven Systemen durch RR-Spiele zu kodieren. Als ein solches Beispiel wird eine Aufzugsteuerung im Abschnitt 3.2.2 noch genauer vorgestellt.

Den Unterschied zwischen der Streett-Bedingung und der RR-Bedingung soll das folgende einfache Beispiel aufzeigen:

Beispiel 3.6. Gegeben sei der folgende Spielgraph:



Die Knoten von Spieler 0 sind durch Kreise dargestellt und die von Spieler 1 als Quadrate. Für das RR-Spiel sei $P_1 = \{1\}$, $P_2 = \{2\}$, $R_1 = \{4\}$ und $R_2 = \{3\}$. Analog verfahren wir für das entsprechende Streett-Spiel.

Betrachtet man die Partie $\rho = 11123232323\ldots$ werden die Zustände 2 und 3 unendlich oft besucht und somit auch die Mengen P_2, R_2. Der Zustand 1 hingegen wird nur endlich oft besucht und somit auch die Menge P_1. Somit erfüllt die Partie ρ die Streett-Bedingung aber nicht die RR-Bedingung.

Um bei Request-Response-Spielen die Gewinnbereiche für Spieler 0 bzw. 1 sowie deren Gewinnstrategien bestimmen zu können, werden diese im Folgenden auf Büchi-Spiele reduziert. Die Spielreduktion wandelt ein gegebenes Spiel durch eine Erweiterung des Spielgraphen in ein vereinfachtes um. Durch die Bestimmung der Gewinnbereiche und -strategien im vereinfachten Spiel können dann die Gewinnbereiche und Automatengewinnstrategien im gegebenen Spiel bestimmt werden.

Definition 3.7 (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 eindeutig in eine Partie ρ' über G' überführt durch
   (a) eine Funktion $f : Q \to Q \times S$ ($f(q)$ ist der Anfang von ρ'),

(b) $\forall (q,s) \in Q \times S,\ \forall p : (q,p) \in E \Rightarrow$ ex. genau ein s' mit $((q,s),(p,s')) \in E'$,

(c) $\forall ((q,s),(p,s')) \in E'$, dann gilt $(q,p) \in E$.

3. Für ρ und ρ' gemäß 2. gilt: $\rho \in \mathit{Win}_0 \Leftrightarrow \rho' \in \mathit{Win}_0{}'$.

Die Motivation für diese Definition ist folgender offensichtlicher Sachverhalt: Gilt $(G, \mathit{Win}_0) \leq (G', \mathit{Win}_0{}')$ und ist das Spiel $(G', \mathit{Win}_0{}')$ mit positionalen Gewinnstrategien lösbar, so ist (G, Win_0) mit Automatenstrategien (über dem Zustandsraum S der Definition) lösbar. Diese Aussage verwenden wir in folgendem Satz:

Satz 3.8 ([WHT03])**.** *Request-Response-Spiele lassen sich auf Büchi-Spiele reduzieren und somit lassen sich die Gewinnbereiche und entsprechende Automatengewinnstrategien berechnen.*

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

Konstruktion: 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 ein zusätzliches RR-Paar (P_2, R_2) mit $P_2 = Q_2 = \emptyset$ hinzu. Wir geben ein Büchi-Spiel $(G', \mathit{Win}_0{}')$ an mit $(G, \mathit{Win}_0) \leq (G', \mathit{Win}_0{}')$. Definiere $G' = (Q', E')$ und Endzustandsmenge F folgendermaßen:

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

Die Initialisierungsfunktion $g : Q \to Q'$ ist definiert durch: $g : q \mapsto (q, \{i \mid q \in P_i\}, 1, 0)$

Der Korrektheitsbeweis (vgl. [Hüt03]) ergibt sich sofort aus den Definitionen.

Korollar 3.9 (Laufzeit [Hüt03])**.** *Die Gewinnbereiche und -strategien in einem Request-Response-Spiel $G = (Q, E)$ mit r RR-Paaren können in Zeit $\mathcal{O}(4^r \cdot r^2 \cdot |Q| \cdot |E|)$ berechnet werden.*

Beweis. Der dominierende Faktor für die Laufzeit ist das Bestimmen der Gewinnbereiche und -strategien des Büchi-Spiels. Ein Büchi-Spiel kann in Zeit $\mathcal{O}(|E| \cdot |F|)$ gelöst werden. Im Fall der RR-Reduktion bekommt man $O(|E'| \cdot |Q|)$ mit $|Q'| = 2^r \cdot r \cdot |Q|$ und $|E'| = |E| \cdot \frac{|Q'|}{|Q|}$. Insgesamt ergibt sich die Laufzeit von $O(4^r \cdot r^2 \cdot |Q| \cdot |E|)$. □

Korollar 3.10. *Für jede Gewinnstrategie eines Request-Response-Spiels mit n Zuständen und r Bedingungen, die durch die vorgestellte Reduktion auf Büchi-Spiele erzeugt wurde, gilt, dass jede Bedingung nach spätestens $n \cdot r$ Zügen erfüllt wird (für alle Partien, die im Gewinnbereichs von Spieler 0 beginnen).*

Beweis. Durch den Zähler bei der Reduktion auf Büchi-Spiele werden die offenen Bedingungen in der Reihenfolge ihrer Nummerierung erfüllt. Um eine einzelne Bedingung zu erfüllen, benötigt man maximal n Züge mittels der Attraktorstrategie. Somit kann es im ungünstigsten Fall maximal $n \cdot r$ Schritte dauern, eine offene Bedingung zu erfüllen. □

3.2.1 Einfache Request-Response-Spiele

Die im letzten Abschnitt vorgestellte Reduktion von Request-Response-Spielen auf Büchi-Spiele ist in Bezug auf den Speicherbedarf nicht optimal, wenn nur ein RR-Paar in der Spieldefinition benutzt wird. Dies liegt daran, dass bei der Reduktion ein zusätzliches RR-Paar eingefügt werden muss (siehe Konstruktion zum Satz 3.8). Das eingefügte Paar wurde nur dazu benötigt, um eine Erfüllung des ersten Paares zu signalisieren. Request-Response-Spiele mit nur genau einem RR-Paar werden im weiteren Verlauf als einfache RR-Spiele bezeichnet. In diesem Unterabschnitt wird eine direkte Reduktion von einfachen RR-Spielen auf Büchi-Spiele vorgestellt, die ohne ein zusätzliches RR-Paar auskommt und somit den Speicherbedarf senkt.

Satz 3.11. *Einfache Request-Response-Spiele lassen sich auf Büchi-Spiele reduzieren. Somit lassen sich die Gewinnbereiche und entsprechende Automatengewinnstrategien berechnen, die mit zwei Zuständen auskommen.*

Beweis. Sei (G, Win_0) ein RR-Spiel mit genau einem RR-Paar, definiert durch die Mengen $P_1, R_1 \subseteq Q$. Dann kann ein Büchi-Spiel $(G', Win_0{}')$ angegeben werden mit $(G, Win_0) \leq (G', Win_0{}')$. Wir arbeiten mit dem Speicher $S = \{0, 1\}$. Dieser wird dazu genutzt, um zu signalisieren, ob die Bedingung aktiv ist (Wert 1 entspricht aktiv). Definiere $G' = (Q', E')$ und Endzustandsmenge F folgendermaßen:

- $Q' := Q \times \{0, 1\}$
- $F := Q \times \{0\}$
- $f(q) := \begin{cases} (q, 0), & \text{falls } q \notin P_1 \\ (q, 1), & \text{sonst} \end{cases}$
- $((q, m), (q', m')) \in E' \Leftrightarrow$
  - $(q, q') \in E$
  - $m' = 1 \Leftrightarrow q' \notin R_1 \wedge ((m = 1) \vee (q' \in P_1))$

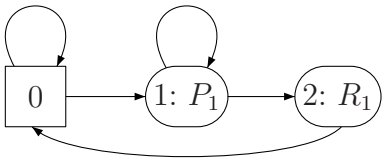
Behauptung: $\rho \in Win_0 \Leftrightarrow \rho' \in Win_0{}'$

$$\begin{aligned} \rho' \notin Win_0{}' &\Leftrightarrow \exists j_0 \in \mathbb{N} : \forall j > j_0 \text{ gilt: } \rho'(j) \notin F \\ &\Leftrightarrow \exists j_0 \in \mathbb{N} : \forall j > j_0 \text{ gilt: } \rho'(j) = (q, 1) \\ &\Leftrightarrow \exists j_1 \leq j_0 \text{ mit } \rho(j_1) \in P_1 \text{ und } \forall j \geq j_1 \text{ gilt: } \rho(j) \notin R_1 \\ &\Leftrightarrow \rho \notin Win_0 \end{aligned}$$

□

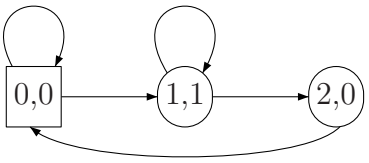
Betrachtet wird noch ein einfaches Beispiel, welches die Komplexität für einfache Request-Response-Spiele mit der gerade vorgestellten Reduktion verglichen mit der Reduktion für allgemeine Request-Response-Spiele verdeutlicht.

Beispiel 3.12. Gegeben sei das folgende einfache Request-Response-Spiel.

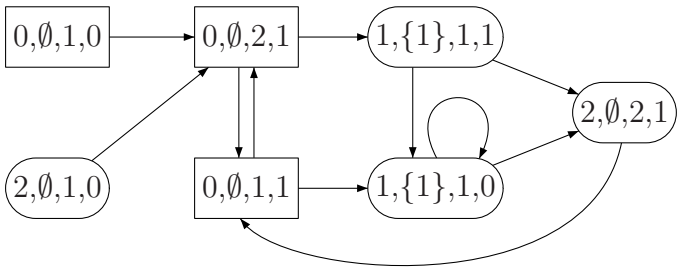


Die rechteckigen Zustände gehören Spieler 1 und die ellipsenförmigen Knoten entsprechend Spieler 0. Welche Zustände zur Menge P_1 bzw. R_1 gehören, ist direkt in den Spielgraphen eingetragen.

Die hier vorgestellte Reduktion von einfachen Request-Response-Spielen liefert für das gegebene Spiel formal einen Spielgraphen mit sechs Zuständen. Schränkt man den Spielgraphen aber auf die von transformierten Zuständen des Originalspiels erreichbaren Zustände ein, ergibt sich der folgende erweiterte Spielgraph:



Vergleicht man dies mit der Reduktion für allgemeine Büchi-Spiele, sieht man einen deutlichen Komplexitätsunterschied. Formal liefert die Reduktion einen Spielgraphen mit 48 Zuständen, aber auch hier werden wieder nur die von transformierten Zuständen erreichbaren Zustände angegeben.



Die in Kapitel 5 vorgestellte Experimentierplattform GASt nutzt automatisch für RR-Spiele mit genau einem RR-Paar die Reduktion für einfache RR-Spiele aus diesem Abschnitt.

3.2.2 Beziehung zwischen verallgemeinerten Büchi-Spielen und RR-Spielen

Verallgemeinerte Büchi-Spiele sind durch eine Gewinnbedingung der Form $\mathcal{F} = \{F_1, \ldots, F_n\}$ gegeben. Spieler 0 gewinnt eine Partie in einem solchen verallgemeinerten Büchi-Spiel, wenn jede dieser F_i-Mengen immer wieder besucht wird. Somit ist ein verallgemeinertes Büchi-Spiel ein Spezialfall eines Request-Response-Spiels.

Korollar 3.13. *Zu einem gegebenen verallgemeinerten Büchi-Spiel $G = (Q, E)$ mit $\mathcal{F} = \{F_1, \ldots, F_n\}$ gibt es ein äquivalentes RR-Spiel $G' = (Q, E)$ mit $\Omega = \{(P_1, R_1), \ldots, (P_n, R_n)\}$, wobei $P_i = Q \setminus F_i$ und $R_i = F_i$ für $1 \leq i \leq n$.*

Hier wird deutlich, dass Spieler 0 das in Korollar 3.13 angegebene RR-Spiel genau dann gewinnt, wenn jede Menge F_i immer wieder besucht wird (jeder Zustand, der nicht zur Menge F_i gehört, aktiviert die i-te RR-Bedingung). Somit ist das RR-Spiel äquivalent zum ursprünglichen verallgemeinerten Büchi-Spiel.

Auf der anderen Seite kann unter gewissen Umständen ein RR-Spiel in ein effizienter zu lösendes, verallgemeinertes Büchi-Spiel umgewandelt werden (siehe dazu auch Abschnitt 3.3.2). Dazu wird der Begriff der P-Abgeschlossenheit benötigt:

Definition 3.14 (P-abgeschlossen)**.** Ein Request-Response-Spiel $G = (Q, E)$ heißt *P-abgeschlossen* genau dann, wenn für alle möglichen Partien des Spiels G und alle RR-Paare gilt, dass jeder endliche Pfad von einem P_i-Zustand aus in Zuständen aus P_i verbleibt; es sei denn, es wird ein R_i-Zustand erreicht.

Satz 3.15. *Ist ein Request-Response-Spiel $G = (Q, E)$ P-abgeschlossen, gibt es ein äquivalentes verallgemeinertes Büchi-Spiel auf demselben Spielgraphen $G = (Q, E)$.*

Beweis. Als Akzeptierkomponente $\mathcal{F} = \{F_1, \ldots, F_r\}$ werden die Mengen F_i gewählt als $F_i = R_i \cup (Q \setminus P_i)$.

Man kann leicht erkennen, dass dieses verallgemeinerte Büchi-Spiel äquivalent zum gegebenen RR-Spiel ist:

Spieler 0 verliert die Partie im gegebenen RR-Spiel
$\Leftrightarrow \exists i : 1 \leq i \leq r$, Bedingung i wird in der Partie aktiviert und nicht mehr erfüllt
$\Leftrightarrow$ im äquivalenten Büchi-Spiel werden schließlich keine Zustände aus der Menge F_i mehr besucht
$\Leftrightarrow$ Spieler 0 verliert die Partie im äquivalenten Büchi-Spiel. □

Dies ist also eine Eigenschaft des zu Grunde liegenden Spielgraphen $G = (Q, E)$. Schauen wir uns dazu als Beispiel eine Aufzugsteuerung an, die erst als Request-Response-Spezifikation aufgefasst worden ist und anschließend als verallgemeinerte Büchi-Spezifikation, da der zu Grunde liegende Spielgraph, die Bedingung aus Satz 3.15 erfüllt. Das Beispiel ist eine Weiterentwicklung aus [WHT03].

Beispiel 3.16. Betrachtet wird eine Aufzugsteuerung für zwei Aufzüge in einem Haus mit e Etagen – dargestellt in Abbildung 3.1. Es gibt zwei Spieler: die Aufzugsteuerung, die die beiden Aufzüge koordiniert, und die Umgebung, bei der Personen die Aufzüge anfordern bzw. Ziele auswählen. In einem Zug der Aufzugsteuerung können beide Aufzüge bewegt werden, in einem Zug der Umgebung können die Benutzer auf den Etagen und in den beiden Aufzüge neue Anforderungen stellen. Dabei werden die Anforderungen von den Etagen (dass ein Aufzug kommen soll), die von beiden Aufzügen erfüllt werden können, und Anforderungen aus den Aufzügen, die von den betreffenden Aufzügen erfüllt werden müssen, unterschieden. Die folgende Liste zeigt die Anforderungen, die die Aufzugsteuerung erfüllen soll:

1. Nachdem eine Etage angefordert worden ist, kommt irgendwann ein Aufzug dorthin (wenn die Anforderung aus einem Aufzug kam, muss genau dieser Aufzug die Bedingung auch erfüllen).

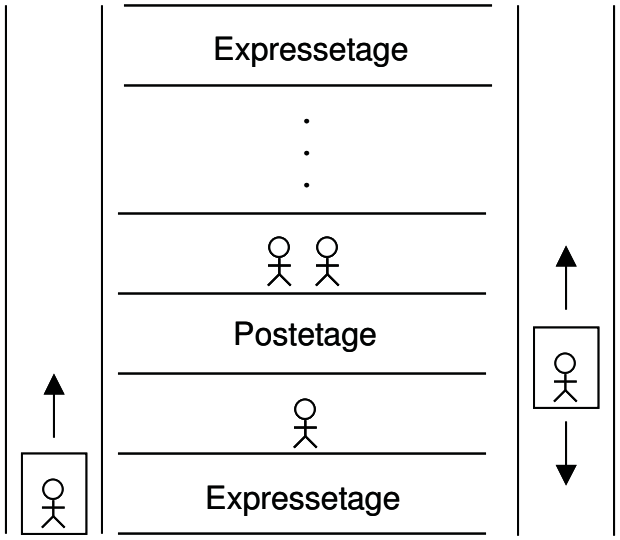


Abbildung 3.1: Schema der Aufzugsteuerung

2. Die höchste und die niedrigste Etage werden immer sofort bedient ("Expressetagen").

3. In der zweiten Etage befindet sich die Poststelle und ein Aufzug, der diese Etage anfährt, muss einen Zug der Aufzugsteuerung aussetzen, damit die Post ein- und ausgeladen werden kann.

4. Die beiden Aufzüge stoppen niemals gleichzeitig in der zweiten Etage.

5. Kein Aufzug fährt an einer angeforderten Etage vorbei.

6. Ein Aufzug fährt eine Etage nur dann unaufgefordert an, wenn keine Anforderungen vorliegen.

Die nächsten beiden Bedingungen sind Annahmen über das Verhalten der Umgebung. Sie sind aufgenommen worden, um der Aufzugsteuerung eine bessere Gewinnchance zu geben.

7. Höchstens eine Person steigt in den Aufzug, wenn er auf einer Etage hält (somit ist das Ziel eindeutig).

8. Es werden insgesamt immer mindestens drei Etagen nicht angefordert.

In Tabelle 3.2 sieht man einen Vergleich zwischen den Modellierungen durch ein verallgemeinertes Büchi-Spiel und ein Request-Response-Spiel für die Aufzugsteuerung. Die

Simulation ist mit der Experimentierplattform GASt aus Kapitel 5 auf einem Pentium IV mit einem Gigabyte RAM erfolgt. In der Tabelle ist für beide Spieltypen und für drei bis fünf Etagen (Parameter e) die Größe des Spielgraphen, die Größe des erweiterten Spielgraphen und die Zeit, die benötigt wurde, um das Spiel zu lösen, angegeben.

Typ	e	**Spielgraph**		**Erweiterter Spielgraph**		**Zeit**
		Zustände	Transit.	Zustände	Transitionen	
RR	3	25	85	1.200	4.080	11,99
	4	673	2.711	516.864	2.082.048	19,06
	5	12.913	71.371	119.006.208	657.755.136	372,48
V. Büchi	3	25	85	150	510	10,62
	4	673	2.711	8.076	32.532	13,48
	5	12.913	71.371	232.434	1.284.678	36,32

Tabelle 3.2: Aufzugsteuerung: Vergleich RR-Bedingung und verallgemeinerte Büchi-Bedingung

Dieses Beispiel zeigt, dass es sinnvoll ist, bei einem RR-Spiel den gegebenen Spielgraphen zu überprüfen, ob die Bedingung aus Satz 3.15 erfüllt ist, damit man zu dem effizienter zu lösenden verallgemeinerten Büchi-Spiel übergehen kann.

3.3 Speicheroptimierung bei Spielreduktionen

In diesem Abschnitt wollen wir uns mit der klassischen Frage der Speicher-Optimierung von Gewinnstrategien befassen. Hierbei werden wir die beiden Gewinnbedingungen “Request-Response” und “verallgemeinert Büchi” betrachten. Die Beobachtungen werden dabei keine neuen asymptotischen Komplexitätsschranken für die bekannten Reduktionen liefern, sondern dazu beitragen, dass konstante Faktoren eingespart werden können. Dies kann durchaus dazu führen, dass Berechnungen durchgeführt werden können, die ohne Optimierung an Speichermangel scheitern. Wir diskutieren auch, wie sich die Optimierungen auf den enumerativen bzw. den symbolischen Zustandsraum auswirken.

3.3.1 Request-Response-Bedingung

Bei der in Abschnitt 3.2 vorgestellten Reduktion von Request-Response-Spielen auf Büchi-Spiele kann der Zustandsraum des erweiterten Spielgraphen noch optimiert werden. Dies sollte sich vor allem für den abstrakten Zustandsraum auszahlen.

Der Zustandsraum des erweiterten Spielgraphen war dort definiert worden als:

$$Q' := Q \times 2^{\{1,\dots,r\}} \times \{1,\dots,r\} \times \{0,1\}$$

Darin sind viele Zustände enthalten, die keine eingehende Kante haben und nicht das Ergebnis einer Transformation eines Zustands des ursprünglichen Spielgraphen sind. Diese werden nicht benötigt und können somit weggelassen werden. Formal lässt sich der Zustandsraum dann definieren als:

$$Q' := Q \times 2^{\{1,\dots,r\}} \times \{1,\dots,r\} \times \{0,1\} \setminus (A \cup B \cup C \cup D)$$

Mit:

- $A := \{(q, M, m, f) \mid \exists i : q \in R_i \wedge i \in M\}$
  Das sind genau die Zustände $q \in Q$, die in einer Antwortmenge R_i enthalten sind (also $q \in R_i$), deren Anforderungsflag im erweiterten Spielgraphen aber nicht zurückgesetzt wurde.

- $B := \{(q, M, m, f) \mid \exists i : q \in (P_i \setminus R_i) \wedge i \notin M\}$
  Das sind genau die Zustände $q \in Q$, die in einer Anforderungsmenge P_i enthalten sind (also $q \in P_i$), deren Anforderungsflag im erweiterten Spielgraphen aber nicht gesetzt wurde.

- $C := \{(q, M, m, f) \mid \exists i : i \notin M \wedge m = i \wedge f = 0\}$
  Die Menge C beschreibt die Zustände, bei denen die Bedingung i nicht offen ist, aber die Variable m nicht auf die nächste Bedingung gesetzt worden ist (vgl. die Definitionen von m' und f' bei der Reduktion von Request-Response auf Büchi).

- $D := \{(q, M, m, f) \mid \exists i : i \in M \wedge m = i + 1 \wedge f = 1\}$
  Die Menge D ist analog zur Menge C definiert mit dem Unterschied, dass hier die Variable m erhöht worden ist, obwohl die Bedingung i noch aktiv ist.

Dann muss noch die Initialisierungsfunktion $g : Q \to Q'$ angepasst werden, damit keine Zustände in ihrem Bild sind, die nicht mehr im Spielgraphen enthalten sind. Eine geeignete Definition für g lautet dann

$$g : q \mapsto \begin{cases} (q, \{i \mid q \in P_i\}, 1, 0), & \text{falls } q \in P_i \\ (q, \{i \mid q \in P_i\}, 2, 1), & \text{sonst} \end{cases}$$

Bemerkung 3.17. Das Ergebnis dieser Optimierung der Zustandsanzahl wird beim Beispiel 3.16 deutlich. Die Ergebnisse für die optimierte und nicht optimierte Variante sind in Tabelle 3.3 sowohl für den abstrakten als auch den symbolischen Zustandsraum dargestellt. Aus den knapp 120 Millionen Zuständen für fünf Etagen werden etwa 5 Millionen. Dies ist für den abstrakten Zustandsraum von Vorteil. Allerdings hat diese Optimierung auf den symbolischen Zustandsraum eher negative Effekte. Dies liegt daran, dass zwar die Zustandsanzahl deutlich nach unten korrigiert wurde, die Variablenanzahl allerdings unverändert bleibt. Dies führt in der Regel zu eher komplexeren BDDs, so dass die booleschen Operationen auf diesen mehr Rechenzeit benötigen (siehe dazu auch Tabelle 3.3).

Opt.	e	**Spielgraph**		**Erweiterter Spielgraph**		**Zeit**	
		Zustände	Transit.	Zustände	Transitionen	symb.	enum.
nicht	3	25	85	1.200	4.080	11,99	11,09
optimiert	4	673	2.711	516.864	2.082.048	19,06	2258,63
	5	12.913	71.371	119.006.208	657.755.136	372,48	—
optimiert	3	25	85	330	1.083	16,46	10,95
	4	673	2.711	59.172	283.272	42,44	92,57
	5	12.913	71.371	5.303.700	44.025.228	34.308,16	—

Tabelle 3.3: Aufzugsteuerung: Auswirkung der Optimierung bei Request-Response-Spielen

3.3.2 Verallgemeinerte Büchi-Bedingung

RR-Spiele können unter gewissen Voraussetzungen als verallgemeinerte Büchi-Spiele aufgefasst werden, wie Satz 3.15 gezeigt hat. Aus diesem Grund wird dieser Spieltyp im Weiteren genauer untersucht. Als erstes wird gezeigt, dass für die aus der Literatur bekannte Reduktion auf "einfache" Büchi-Spiele die Hälfte der Zustände des erweiterten Spielgraphen weggelassen werden können. Dies senkt somit den Speicherbedarf für die Gewinnstrategien. Da die Anordnung der F_i-Mengen bei dieser Reduktion einen Einfluss auf die Laufzeit für das Lösen des reduzierten Büchi-Spiels hat, wird in einem weiteren Unterabschnitt eine Reduktion mit exponentiellem Platzverbrauch vorgestellt, bei der die Anordnung der F_i-Mengen keinen Einfluss auf die Laufzeit hat.

3.3.2.1 Reduktion mit linearem Platz

In der Literatur finden sich zahlreiche Varianten, um einen verallgemeinerten Büchi-Automaten in einen Büchi-Automaten zu transformieren, wenn es um die Transformation von LTL-Formeln in Büchi-Automaten geht – z.B. [GO01]. Diese lassen sich problemlos auf verallgemeinerte Büchi-Spiele übertragen. Dabei wird der Spielgraph des verallgemeinerten Büchi-Spiels um einen Zähler und ein Flag erweitert. Der Zähler durchläuft die Werte 1 bis n, wenn $\mathcal{F} = \{F_1, \ldots, F_n\}$, (siehe auch die Reduktion von Request-Response auf Büchi in Abschnitt 3.2). Die Idee der Reduktion ist, dass die F_i-Mengen zyklisch durchlaufen werden und beim Voranschreiten ein Endzustand besucht wird. Kann der Zyklus einmal nicht fortgesetzt werden, werden von dort ab keine Endzustände mehr besucht und Spieler 1 gewinnt die Partie. Dies wird durch einen Zähler erreicht, der zyklisch weiterläuft, wenn die passende F_i-Menge besucht wurde, und dabei jeweils einen Endzustand durchläuft. Formal lässt sich der erweiterte Spielgraph für ein verallgemeinertes Büchi-Spiel $G = (Q, E)$ mit $\mathcal{F} = \{F_1, \ldots, F_n\}$ wie folgt definieren:

- $Q' := Q \times \{1, \ldots, n\} \times \{0, 1\}$
- $((q, m, f), (q', m', f')) \in E' \Leftrightarrow$
  - $(q, q') \in E$
  - $m' := \begin{cases} m, & \text{falls } q' \notin F_m \\ (m \mod n) + 1, & \text{sonst} \end{cases}$
  - $f' := \begin{cases} 0, & \text{falls } m = m' \\ 1, & \text{sonst} \end{cases}$
- Endzustände $F := \{(q, m, f) \in Q' \mid f = 1\}$
- Initialisierungsfunktion $g : Q \to Q'$ mit $g(q) = (q, 1, 0)$

Dabei sind jedoch etliche Zustände überflüssig. Deshalb kann die Zustandsmenge des erweiterten Spielgraphen definiert werden als

$$Q' := Q \times \{1, \ldots, n\} \times \{0, 1\} \setminus (\bigcup_{i=1}^{n} \underbrace{((Q \setminus F_i) \times i + 1 \times 1)}_{A} \cup \underbrace{(F_i \times i \times 0)}_{B})$$

Die mit A bezeichnete Menge von Zuständen ist überflüssig, da das Flag gesetzt ist, aber kein Zustand aus der Menge F_i besucht worden ist. Analog bezeichnet B die Mengen von Zuständen mit $q \in F_i$, bei der aber das Flag nicht gesetzt worden ist.

Weiterhin muss dann noch die Funktion g, die einen Zustand aus Q in einen Zustand aus Q' überführt, angepasst werden:

$$g(q) = \begin{cases} (q, 1, 0), & \text{falls } q' \notin F_1 \\ (q, 2, 1), & \text{sonst} \end{cases}$$

Bemerkung 3.18. Erläuterungen zur Definition von Q':

1. Das Herausnehmen von Zuständen in der Definition von Q' reduziert die Anzahl der Zustände des erweiterten Spielgraphen auf $|Q| \cdot |\mathcal{F}|$; der naive Ansatz führt auf die doppelte Anzahl.

2. Wir nutzen den Vorteil, dass die letzte Komponente nur dort eine 1 enthält, wo es sinnvoll ist. Dies erspart unnötige Iterationen bei der Recur-Berechnung beim Lösen des Büchi-Spiels.

3. Weiterhin haben die herausgenommenen Zustände alle keine eingehenden Transitionen. Das Herausnehmen kann u. U. eine Iteration bei der $Attr^+$- bzw. $Attr$-Berechnung ersparen (wieder bezogen auf das Lösen des Büchi-Spiels).

Bei der Aufzugsteuerung aus Beispiel 3.16 zeigt sich das in Tabelle 3.4 dargestellte Ergebnis mit der hier vorgestellten Optimierung für verallgemeinerte Büchi-Spiele.

Opt.	e	**Spielgraph**		**Erweiterter Spielgraph**		**Zeit**	
		Zustände	Transit.	Zustände	Transitionen	symb.	enum.
nicht	3	25	85	150	510	10,62	10,20
optimiert	4	673	2.711	8.076	32.532	13,48	36,72
	5	12.913	71.371	232.434	1.284.678	36,32	1692,61
optimiert	3	25	85	75	255	10,53	10,28
	4	673	2.711	4.038	16.266	13,41	36,15
	5	12.913	71.371	116.217	642.339	74,51	1599,84

Tabelle 3.4: Aufzugsteuerung: Auswirkung der Optimierung bei verallgemeinertem Büchi-Spiel

3.3.2.2 Reduktion mit exponentiellem Platz

Die folgende Reduktion eines verallgemeinerten Büchi-Spiels auf ein Büchi-Spiel benötigt eine exponentielle Vergrößerung des Zustandsraums in Bezug auf die Menge $\mathcal{F}$. Sie wird der Vollständigkeit halber betrachtet, da die Reihenfolge der Mengen F_i im System $\mathcal{F}$ eines verallgemeinerten Büchi-Spiels einen Einfluss auf die Laufzeit für das Lösen des reduzierten Büchi-Spiels hat und dieser Einfluss bei dieser Reduktion nicht vorhanden ist. Zu einem gegebenen verallgemeinerten Büchi-Spiel $G = (Q, E)$ mit $\mathcal{F} = \{F_1, \ldots, F_n\}$ kann der erweiterte Spielgraph $G' = (Q', E')$ wie folgt konstruiert werden:

- $Q' := Q \times 2^{\{1,\ldots,n\}}$
- $((q, M), (q', M')) \in E' \Leftrightarrow$
  - $(q, q') \in E$
  - $M' := \begin{cases} M \cup \{i \mid q' \in F_i\}, & \text{falls } M \text{ nicht alle Elemente umfasst} \\ \{i \mid q' \in F_i\}, & \text{sonst} \end{cases}$
- Endzustände $F := \{(q, M) \in Q' \mid M \text{ umfasst alle Elemente}\}$
- Initialisierungsfunktion $f : Q \to Q'$ mit $f(q) = (q, \{i \mid q \in F_i\})$

Wir verifizieren die Behauptung: $\rho \in \mathit{Win}_0 \Leftrightarrow \rho' \in \mathit{Win}_0{}'$:

$$\begin{aligned} \rho' \notin \mathit{Win}_0{}' &\Leftrightarrow \exists i, j \in \mathbb{N} : \forall k \in \mathbb{N} : k > j \to i \notin M \\ &\Leftrightarrow \forall k > j \to \rho(k) \notin F_i \\ &\Leftrightarrow \rho \notin \mathit{Win}_0 \end{aligned}$$

Bemerkung 3.19. Auch bei dieser Reduktion können wieder überflüssige Zustände entfernt werden:

- alle Zustände, bei denen der zusätzliche Speicher nur 1 enthält und selber nicht in einer F_i-Menge sind,
- alle Zustände, die in der Menge F_i sind und bei denen das i-te Flag nicht gesetzt ist.

Diese Optimierung hat wieder vor allem beim enumerativen Zustandsraum Vorteile. Tabelle 3.5 belegt, dass eine erhebliche Anzahl von Zuständen durch diese Optimierung eingespart werden kann.

Opt.	e	Spielgraph		Erweiterter Spielgraph		Zeit	
		Zustände	Transit.	Zustände	Transitionen	symb.	enum.
nicht	3	25	85	200	680	10,57	10,22
optimiert	4	673	2.711	43.072	173.504	13,60	46,40
	5	12.913	71.371	6.611.456	36.541.952	65,83	—
optimiert	3	25	26	86	1.083	10,71	10,12
	4	673	2.711	1.512	5.134	12,80	35,54
	5	12.913	71.371	74.690	281.028	72,36	1562,30

Tabelle 3.5: Aufzugsteuerung: Auswirkung der Optimierung bei verallgemeinertem Büchi-Spiel mit exponentieller Reduktion auf Büchi-Spiele

3.4 Strategiesynthese für Live-Sequence-Charts-Spezifikationen

In diesem Abschnitt wird das Problem betrachtet, Streett-Spiele mit nur einem einzigen Streett-Paar zu lösen; einer Klasse von unendlichen Spielen, welche im Weiteren "einfache Streett-Spiele" genannt werden. Formal wird in Ergänzung zum Spielgraphen ein Paar von Zustandsmengen (F, E) gegeben. Spieler 0 gewinnt eine Partie ρ eines solchen Spiels genau dann, wenn für den Fall, dass ein Zustand aus der Menge E unendlich oft in ρ vorkommt, auch ein Zustand aus F unendlich oft auftritt.

Solche Spiele ermöglichen es, Gewinnbedingungen mit einfachen Fairness-Bedingungen zu spezifizieren: Dass die Menge F unendlich oft besucht wird, ist eine Annahme über das Verhalten der Umgebung. Wenn die Umgebung diese Annahme nicht erfüllt, braucht der Controller seine Liveness-Bedingung nicht zu erfüllen, d. h. er braucht die Menge E nicht unendlich oft zu besuchen.

Das Interessante an dieser Art Spiel ist, dass ihre Modellierungsstärke zwischen Büchi-Spielen, mit denen Fairness-Bedingungen *nicht* modelliert werden können, und Rabin/Streett-Spielen liegt, für die Algorithmen zum Lösen der Spiele komplex sind und bei denen der Speicherbedarf für den Controller in der Regel exponentiell größer ist als die Anzahl der Zustände im Spiel. Im Gegensatz dazu sind die Gewinnstrategien von einfachen Streett-Spielen positional, d. h. sie benötigen keinen Speicher, und es ist möglich, die Spiele in quadratischer Zeit, bezogen auf die Anzahl der Zustände im Spiel, zu lösen.

In der Praxis kommen diese Spiele aus dem Kontext von Szenarien-basierten Spezifikationen, siehe zum Beispiel [BS03]. Diese Beispiele werden hier als Basis für die praktische Auswertung der Analyse der einfachen Streett-Spiele benutzt.

Zuerst wird ein neuer Algorithmus vorgestellt, der einfache Streett-Spiele direkt ohne

Reduktion auf andere Spiele löst und auch symbolisch einfach zu implementieren ist. Anschließend wird die Laufzeit dieses Algorithmus mit der Reduktion auf Paritätsspiele erst theoretisch und anschließend an praktischen Beispielen verglichen. Dabei basieren die praktischen Vergleiche auf Spielen, die aus Live-Sequence-Charts-Spezifikationen [DH01, BS03] abgeleitet sind. Es wird gezeigt, dass sich der neue Algorithmus bei diesen Beispielen besser verhält als die Paritätsspielalgorithmen von Zielonka [Zie98], Jurdziński [Jur00] und Vöge/Jurdziński [VJ00].

3.4.1 Live-Sequence-Charts

Bevor die einfachen Streett-Spiele betrachtet werden, wird erst noch genauer auf die Motivation und Grundlage dieser Spiele eingegangen – die Live-Sequence-Charts (LSCs). Sie wurden von Damm und Harel in [DH01] eingeführt und erweitern die bekannte Sprache der Message-Sequence-Charts (MSC) mit syntaktischen Konstrukten und einer formalen Semantik, die es ermöglicht, formal das Verhalten eines reaktiven Systems zu spezifizieren. Wesentlich ist die Einbeziehung von Liveness-Bedingungen. Diese grafische Sprache basiert auf "Szenarien", ein erfolgreicher Ansatz bei der interaktiven Softwareentwicklung.

Man unterscheidet existentielle und universelle LSCs. Existentielle LSCs beschreiben mögliche Ausführungen des zukünftigen System. Universelle LSCs (ULSC) hingegen beschreiben die Eigenschaften, die das System in jeder Ausführung einhalten muss. Ein universelles Diagramm ist in zwei Teile unterteilt: den oberen Bereich, der auch Prechart genannt wird und in einer gestrichelten, hexagonalen Box dargestellt wird, und das Hauptdiagramm, auch Main Chart genannt, welches von einem durchgezogenen Rechteck umgeben wird. Die Semantik eines universellen LSCs besagt, dass immer, wenn das Prechart in einem Lauf ausgeführt wurde, das Hauptdiagramm folgen muss. Abbildungen 3.2 und 3.3 zeigen universelle LSCs für eine Lichtsteuerung. LSCs sind weiterhin mit einer Menge von sichtbaren oder eingeschränkten Ereignissen verknüpft. Diese Menge beinhaltet alle Ereignisse, die in dem Diagramm vorkommen. Hinzu kommen die Ereignisse, die in einer separaten Einschränkungsklausel (restrict) aufgeführt sind (siehe `turnOn` und `motion` in Abbildung 3.2). Auftreten von Ereignissen, die nicht sichtbar in einem Diagramm sind, werden für das Matching des Precharts bzw. Hauptdiagramms nicht berücksichtigt. Wenn zum Beispiel das Ereignis `motion` nach zwei Ereignissen `nomotion` auftritt, stimmt dies mit dem Prechart aus Abbildung 3.2 nicht überein. Wenn `turnOn` nach drei `nomotion` Ereignissen auftritt, ist das Chart verletzt.

In LSCs können auch andere Merkmale benutzt werden, wie zum Beispiel die Wahlmöglichkeit (siehe das "ALT" Rechteck in Abbildung 3.3) bzw. Bedingungen, die dafür sorgen, dass eine gegebene boolesche Formel wahr sein muss, wenn man dorthin kommt. In Abbildung 3.3 sind die Wahlmöglichkeit und Bedingung zu einem "if-then-else" Sze-

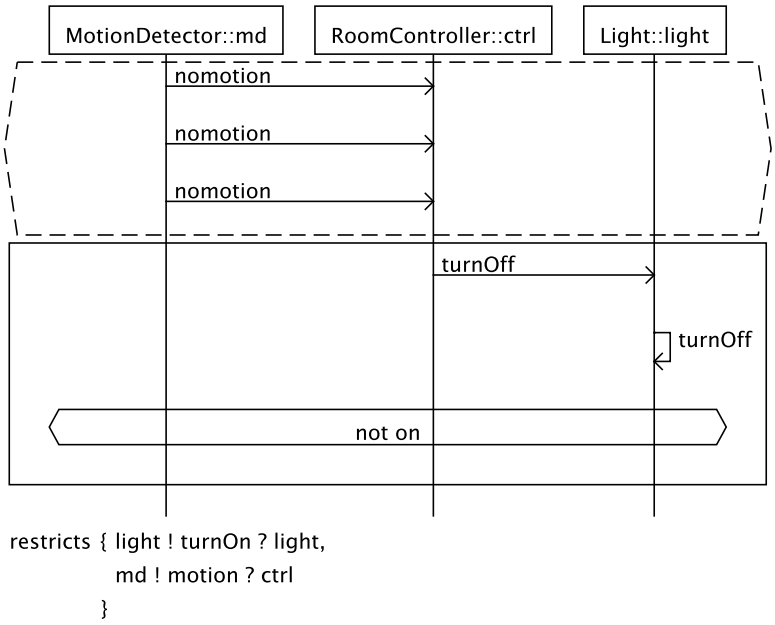


Abbildung 3.2: Lichtsteuerung (Personen verlassen den Raum)

narium kombiniert worden: Wenn das Licht angeschaltet ist, muss es sich selbst auf den richtigen Wert dimmen, ansonsten ist das Hauptdiagramm beendet.

Wenn ein reaktives System entwickelt wird, muss eine Unterscheidung zwischen dem System und seiner Umwelt gemacht werden. In diesem Fall werden Systemagenten, wie `ctrl` und `light`, und Umgebungsagenten, wie `md` und `env`, unterschieden. Der bedeutende Unterschied besteht darin, dass nur die Systemagenten implementiert werden. Deshalb können Aktionen, die von Agenten der Umgebung ausgelöst werden, nicht verhindert oder erzwungen werden. Entwickler müssen somit dafür Sorge tragen, dass die Implementierung der Systemagenten die Spezifikation erfüllt, unabhängig davon, wie sich die Agenten der Umgebung verhalten.

Allerdings ist es zu restriktiv, zu fordern, dass die Implementierung die Spezifikation gegen alle widrigen Verhalten der Umgebung sicherstellen muss. Meistens werden Annahmen über das Verhalten der Umgebungsagenten getroffen. In LSCs sind diese Annahmen erkennbar enthalten: Jeder Agent ist verantwortlich dafür, dass die Ereignisse ausgeführt werden, wenn sie gefordert werden, und ansonsten nicht auftreten, wenn sie das Chart verletzen würden. Zum Beispiel muss `env` irgendwann das Ereignis `getOutsidedim` senden, wenn das Prechart aus Abbildung 3.3 erfüllt wurde; hingegen darf `md` nicht die Ereignisse `motion` oder `nomotion` auslösen, bevor das System das Licht nicht ausgeschaltet hat.

Somit ist die Gewinnbedingung in einem entsprechenden Zwei-Personen-Spiel eine Streett-Bedingung mit genau einem Paar (F_1, E_1). Diese wird im Weiteren noch genau-

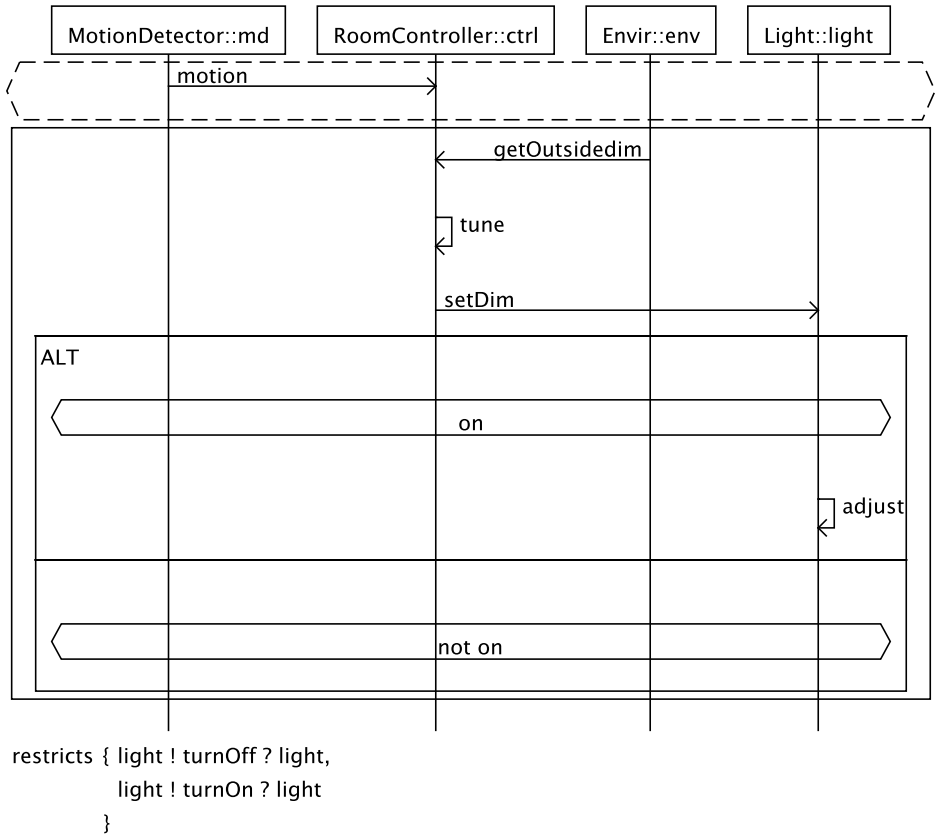


Abbildung 3.3: Lichtsteuerung (Personen betreten den Raum)

er betrachtet. Läufe, in denen die Umgebung für ihre Agenten sicher stellt, dass alle Ereignisse, die gefordert sind, auch irgendwann auftreten und nicht auftreten, wenn sie verboten sind, besuchen Zustände aus F_1 unendlich oft. Läufe, in denen sich das System gemäß der Spezifikation verhält, besuchen Zustände aus E_1 unendlich oft. Somit reduziert sich das Syntheseproblem für LSC-Spezifikationen auf das Lösen von einfachen Streett-Spielen.

3.4.2 Einfache Streett-Spiele

Bei der Streett-Gewinnbedingung [Str82] ist eine Menge von Paaren (F_i, E_i) gegeben, wobei beide Komponenten Teilmengen von Zuständen sind. Spieler 0 gewinnt eine Partie in einem Streett-Spiel, wenn er sicherstellen kann, dass, wenn ein Zustand aus F_i unendlich besucht worden ist, auch ein Zustand aus E_i unendlich oft aufgesucht wird. Einfache Streett-Spiele haben genau ein solches Paar (F_1, E_1) und entstehen auf natürliche Art aus Assume/Guarantee-Rahmenwerken. Die Zustände aus der Menge F_1 repräsentieren Liveness-Annahmen über das Verhalten der Umwelt, und die E_1 Zustände beschreiben Garantien, die der Controller einzuhalten hat. Wenn die Umgebung die

an sie gestellten Erwartungen erfüllt (d. h. die Menge F_1 unendlich oft besucht), dann muss auch der Controller sich korrekt verhalten, also E_1 unendlich oft aufsuchen. Es bleibt noch anzumerken, dass sich einfache Streett-Spiele nicht auf die deutlich einfacheren Erreichbarkeits-, Rekurrenz- (Büchi-) oder co-Büchi-Spiele reduzieren lassen.

3.4.2.1 Reduktion auf Paritätsspiele

Die Gewinnbereiche und -strategien von einfachen Streett-Spielen können in polynomieller Zeit berechnet werden. Die resultierenden Gewinnstrategien für beide Spieler sind positional. Dies gilt nur für den Fall der einfachen Streett-Spiele (also für Streett-Spiele mit genau einem Paar) und nicht allgemein für Streett-Spiele. Ein Ansatz, einfache Streett-Spiele zu lösen, ist die Reduktion auf Paritätsspiele mit genau drei Farben [Bon05]. Bei der Paritätsgewinnbedingung gibt es eine Funktion $c : Q \to \{0, \ldots, k-1\}$, die jedem Knoten eine Farbe (auch Priorität genannt) zuordnet. In diesem Fall ist k die Anzahl der verwendeten Farben. Eine Partie ρ in einem Paritätsspiel wird von Spieler 0 gewonnen, wenn die höchste unendlich oft auftretende Farbe gerade ist. Ein fundamentales Ergebnis der Theorie unendlicher Spiele ([EJ91, Mos84]) besagt, dass Paritätsspiele durch positionale Gewinnstrategien lösbar sind. Für eine Reduktion von einfachen Streett- auf Paritätsspiele werden den Knoten des einfachen Streett-Spiels die folgenden Farben zugewiesen:

$$c(q) = \begin{cases} 2, & q \in E_1 \\ 1, & q \in F_1 \setminus E_1 \\ 0, & q \notin (E_1 \cup F_1) \end{cases}$$

Polynomielle Zeit reicht zum Lösen dieser Paritätsspiele aus, da nur eine feste Anzahl von Farben benutzt wird [Jur00] – nämlich genau drei. Eine detaillierte Analyse von verschiedenen Algorithmen zum Lösen von Paritätsspielen unter Berücksichtigung, dass nur genau drei Farben benötigt werden, wird in Abschnitt 3.4.2.4 gegeben.

3.4.2.2 Direkte Strategiesynthese

In diesem Abschnitt wird ein neuer Algorithmus vorgestellt, der einfache Streett-Spiele ohne eine Reduktion auf andere unendliche Spiele löst. Ein weiterer Vorteil dieses Ansatzes ist, dass er einfach sowohl enumerativ als auch symbolisch implementiert werden kann.

Aus der Sicht von Spieler 1 handelt es sich beim einfachen Streett-Spiel um das komplementäre einfache Rabin-Spiel [Rab69, Rab72]: Spieler 1 gewinnt, wenn die Menge E_1 nur endlich oft besucht wird, aber F_1 unendlich oft. Somit ist die Rabin-Bedingung

eine Kombination aus *Rekurrenz-* und *Persistenzeigenschaft*, was im Weiteren eine *Percur*-Bedingung genannt wird. Klarlund hat in [Kla94] gezeigt, dass Spieler 0 in einem Rabin-Spiel eine positionale Gewinnstrategie auf seinem Gewinnbereich hat. An dieser Stelle soll jetzt ein direkter Lösungsalgorithmus für Rabin-Spiele mit einem Paar vorgestellt werden, der die Kombination aus Persistenz und Rekurrenz berechnet. Dazu wird die Gewinnbedingung noch einmal genauer betrachtet. Damit Spieler 1 gewinnt, muss er eine Schleife in dem Spielgraphen erreichen, in der er einen Zustand aus F_1 besucht und keinen aus der Menge E_1. Solch eine Schleife wird auch von dem Algorithmus $Recur(F_1)$ gefunden, dem wohlbekannten Algorithmus zum Lösen von Büchi-Spielen, der alle Zustände von F_1 bestimmt, von denen Spieler 0 erzwingen kann, dass sie unendlich oft besucht werden. Formale Definitionen von *Recur* und $Attr^+$ sind in Abschnitt 3.1 zu finden.

Allerdings löst *Recur* nicht das einfache Rabin-Spiel, da i.a. zuviele Schleifen berechnet werden. Unser Ansatz modifiziert den *Recur* Algorithmus in geeigneter Weise, so dass nur die Zustände von F_1 zurückgegeben werden, die E_1 nicht besuchen. Um anzudeuten, dass die Funktion auf einem Teilgraphen (auch Subarena genannt) berechnet werden soll, wird der entsprechende Teilgraph im Exponenten angedeutet.

1. Die Definition von $Attr^+$ wird so modifiziert, dass keine Zustände von E_1 benutzt werden.

$$\begin{aligned} Attr_1^+(F_1, E_1) &:= \{q \in Q \mid \text{ Spieler 1 kann in } \geq 1 \text{ Schritten } F_1 \\ &\qquad \text{erreichen, ohne Zustände aus } E_1 \text{ zu benutzen}\} \end{aligned}$$

$$\begin{aligned} A_1^0 &:= \emptyset \\ A_1^{i+1} &:= A_1^i \ \cup \\ &\quad \{q \in Q_1 \setminus E_1 \mid \exists (q,r) \in E : r \in (A_1^i \cup F_1)\} \cup \\ &\quad \{q \in Q_0 \setminus E_1 \mid \forall (q,r) \in E : r \in (A_1^i \cup F)\} \\ Attr_1^+(F_1, E_1) &:= \bigcup_{i \geq 0} A_1^i \end{aligned}$$

2. *Recur* wird so modifiziert, dass nur die Teilmenge von F_1 zurückgegeben wird, die unendlich oft besucht werden kann, ohne Zustände von E_1 zu benutzen.

$$\begin{aligned} Recur_1^0(F_1, E_1) &:= F_1 \setminus E_1 \\ Recur_1^{i+1}(F_1, E_1) &:= F_1 \cap Attr_1^+(Recur_1^i(F_1, E_1), E_1) \\ Recur_1(F_1, E_1) &:= \bigcap_{i \geq 0} Recur_1^i(F_1, E_1) \end{aligned}$$

3. Da diese Definition von *Recur* noch nicht alle Zustände liefert, die unendlich oft besucht werden können, ohne die Zustände aus E_1 zu benutzen, wird eine Fixpunktberechnung durchgeführt. *Percur* fordert "Rekurrenz" von F_1 und fordert "Persistenz" von E_1.

$$\begin{aligned} Percur_1^0(F_1, E_1) &:= \emptyset \\ Percur_1^{i+1}(F_1, E_1) &:= Percur_1^i(F_1, E_1) \cup Attr_1^{G'}(Recur_1^{G'}(F_1, E_1)) \\ &\quad \text{mit } G' = G \setminus Percur_1^i(F_1, E_1) \end{aligned}$$

Dann gilt $Percur_1^0(F_1, E_1) \subseteq Percur_1^1(F_1, E_1) \subseteq Percur_1^2(F_1, E_1) \subseteq \ldots$. Für ein $l \leq |Q|$ gilt $Percur_1^l(F_1, E_1) = Percur_1^{l+1}(F_1, E_1)$, die Inklusionsfolge wird also stationär, da höchstens $|Q|$-mal etwas hinzugefügt werden kann. Setze

$$Percur_1(F_1, E_1) := \bigcup_{i=0}^{|Q|} Percur_1^i(F_1, E_1) \qquad \text{oder statt } |Q| \text{ auch } l \text{ wie oben.}$$

Der Gewinnbereich von Spieler 1 in einem einfachen Streett-Spiel ist dann durch $W_1 = Percur_1(F_1, E_1)$ gegeben.

4. Die Gewinnstrategien für beide Spieler können wie folgt berechnet werden; dabei steht σ für die Strategie, die durch den im Exponenten stehenden Algorithmus berechnet worden ist:

   - Für Spieler 1 definieren wir: $strat_1 : W_1 \cap Q_1 \to Q$ als die Attraktorstrategien der *Percur*-Berechnungen wie folgt (wir fassen Funktionen hier als Mengen von Paaren (Argument, Wert) auf):

$$\begin{aligned} strat_1^0(q) &:= \emptyset \\ strat_1^{i+1}(q) &:= strat_1^i(q) \cup \sigma_{Attr_1^{G'}(Recur_1^{G'}(F_1, E_1))} \\ &\quad \text{mit } G' \text{ aus der } Percur \text{ Berechnung} \\ strat_1(q) &:= \bigcup_{i=0}^{|Q|} strat_1^i(q) \end{aligned}$$

   - Für Spieler 0 definieren wir: $strat_0 : W_0 \cap Q_0 \to Q$. Die Idee dabei ist, dass Spieler 0 versucht, E_1 zu besuchen, wenn dies möglich ist, und ansonsten schließlich keine Zustände aus E_1 und F_1 mehr aufsucht. Um die Strategie

zu berechnen, wird für den zweiten Fall eine Fixpunktberechnung benötigt.

$$
\begin{aligned}
strat_0^0(q) &:= \begin{cases} \sigma_{Attr_0(E_1 \cap W_0)}, & q \in Attr_0(E_1 \cap W_0) \setminus (E_1 \cap W_0) \\ r \in W_0 \text{ mit } (q,r) \in E, & q \in (E_1 \cap W_0) \end{cases} \\
G_0 &:= W_0 \setminus Attr_0(E_1 \cap W_0) \\
G_{i+1} &:= G_i \setminus T_i \\
S_i &:= G_i \setminus Attr_1^{G_i}(F_1) \\
T_i &:= Attr_0^{G_i}(S_i) \\
strat_0^{i+1}(q) &:= strat_0^i(q) \cup \begin{cases} \sigma_{Attr_1^{G_i}(F_1)}, & \text{falls } q \in S_i \\ \sigma_{Attr_0^{G_i}(S_i)}, & \text{sonst} \end{cases}
\end{aligned}
$$

Dann gilt $strat_0^0(q) \subseteq strat_0^1(q) \subseteq strat_0^2(q) \subseteq \ldots$ Für ein $l \leq |Q|$ gilt $strat_0^l(q) = strat_0^{l+1}(q)$, die Inklusionsfolge wird also stationär, da die Folge der Teilspielgraphen maximal |Q| Elemente umfassen kann. Setze

$$strat_0(q) := \bigcup_{i=0}^{|Q|} strat_0^i(q) \qquad \text{oder statt } |Q| \text{ auch } l \text{ wie oben.}$$

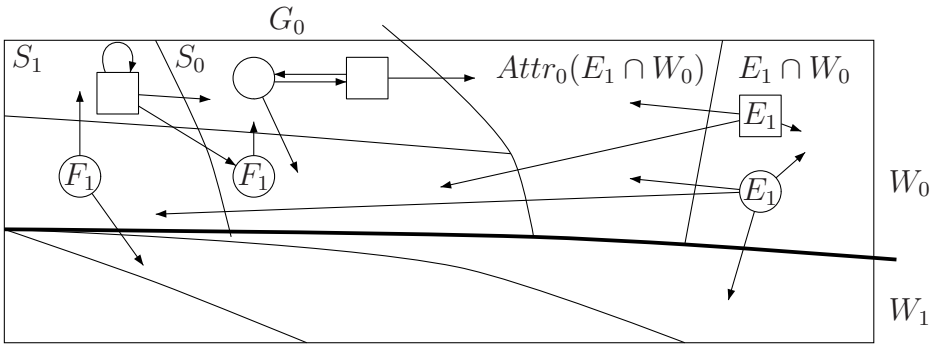


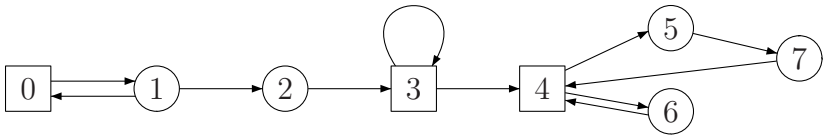
Abbildung 3.4: Gewinnstrategie von Spieler 0 in einem einfachen Streett-Spiel

Abbildung 3.4 soll die Definition der Gewinnstrategien für Spieler 0 in einem einfachen Streett-Spiel verdeutlichen. Spieler 0 mag die Möglichkeit haben, den Besuch von $E_1 \cap W_0$ zu erzwingen, also in $Attr_0(E_0 \cap W_0)$ zu sein. In diesem Fall wendet er die Attraktorstrategie an und erreicht schließlich $E_1 \cap W_0$. Danach stellt er sicher, dass er in W_0 bleibt. Dieser Zug kann ihn nach $Attr_0(E_0 \cap W_0)$ zurückführen oder zu einem Bereich, in dem er nicht den Besuch von E_1 erzwingen kann. In diesem Fall besteht seine Strategie darin, Spieler 1 an dem Besuch von F_1 zu hindern, was genau die Fixpunktberechnung bewirkt. Als erstes wird die Menge von Zuständen berechnet, von

denen Spieler 0 für immer verhindern kann, dass die Partie Zustände aus F_1 aufsucht. Dies ist nur solange möglich, wie Spieler 1 sich nicht entscheidet, den Attraktor von $E_1 \cap W_0$ aufzusuchen. Diese Menge wird S_0 genannt. Dann wird eine Strategie berechnet, um S_0 zu erreichen. Dabei ist es möglich, dass Zustände aus F_1 auf dem Weg zu S_0 besucht werden. Wieder wird der Teilspielgraph berechnet, um Spieler 1 die Möglichkeit zu geben, diesen Spielgraphen zu verlassen und in den Attraktor von $E_1 \cap W_0$ zu wechseln.

Bevor die Korrektheit nachgewiesen wird, soll der Algorithmus an einem Beispiel veranschaulicht werden.

Beispiel 3.20. Der folgende Spielgraph mit $E_1 = \{2, 7\}$ und $F_1 = \{1, 3, 5\}$ sei gegeben.



Der Algorithmus liefert die folgenden Zwischenergebnisse:

- $Percur_1^0(F_1, E_1) = \emptyset$
- $Percur_1^1(F_1, E_1)$:
  - $Recur_1^0(F_1, E_1) = F_1 = \{1, 3, 5\}$
  - $Attr_1^+(F_1, E_1) = \{0, 3, 4, 6\}$
    $Recur_1^1(F_1, E_1) = F_1 \cap \{0, 3, 4, 6\} = \{3\}$
  - $Attr_1^+(\{3\}, E_1) = \{3\}$
    $Recur_1^1(F_1, E_1) = F_1 \cap \{3\} = \{3\}$
    $\Rightarrow Recur_1(F_1, E_1) = \{3\}$
  - $Attr_1(\{3\}) = \{2, 3\}$

  $\Rightarrow Percur_1^1(F_1, E_1) = \{2, 3\}$
- $Percur_1^2(F_1, E_1)$: Auf dem Teilspielgraphen mit den Zuständen 2 und 3 ergibt sich:
  - $Recur_1^0(F_1, E_1) = F_1 = \{1, 5\}$
  - $Attr_1^+(F_1, E_1) = \{0, 1, 4, 6\}$
    $Recur_1^1(F_1, E_1) = \{1\} = Recur_1(F_1, E_1)$

– $Attr_1(\{1\}) = \{0, 1\}$

Also $Percur_1^2(F_1, E_1) = \{2, 3\} \cup \{0, 1\} = \{0, 1, 2, 3\} = Percur_1(F_1, E_1)$

Satz 3.21. *Der Funktionswert $Percur_1(F_1, E_1)$ liefert die Gewinnbereiche für beide Spieler in einem einfachen Streett-Spiel mit dem Streett-Paar (E_1, F_1); es sind $strat_i$ die korrespondierenden Gewinnstrategien für beide Spieler.*

Beweis. $Percur_1(F_1, E_1) \subseteq W_1$: Betrachte die beiden Fälle:

1. Sei q ein Element von einer $Recur_1^{G'}(F_1, E_1)$-Berechnung auf dem Teilgraph $G' = G \setminus Percur_1^i(F_1, E_1)$. Dann gibt es die Möglichkeit, mit der Strategie $strat_1$ einen Zustand von F_1 immer wieder zu besuchen, ohne dabei Zustände von E_1 zu benutzen. Deshalb gilt: $q \in W_1$.

2. Bei einer $Attr_1$-Berechnung sei q zuerst aufgenommen. Dann kann ein Zustand, der zu dem ersten Fall gehört, mittels der Attraktor-Strategie $strat_1$ aufgesucht werden. Dabei werden nur endlich oft Zustände von E_1 besucht (die Attraktorstrategie verkürzt den Abstand zu der gesuchten Menge und somit wird jeder Zustand höchstens einmal besucht). Darum gilt: $q \in W_1$.

$W_1 \subseteq Percur_1(F_1, E_1)$:
Annahme: Es existiert ein Zustand $q \in W_1 \setminus Percur_1(F_1, E_1)$. Daraus folgt, dass Spieler 1 von q aus garantieren kann, dass schließlich immer mal wieder ein Zustand aus F_1 aufgesucht wird, aber keiner von E_1. Oder anders ausgedrückt: Es gibt für jede Partie, die in q beginnt, einen Zustand $q_i \in F_1$, so dass kein Zustand aus E_1 in der Zukunft besucht wird, aber q_i immer wieder. Daraus folgt, dass jedes q_i in einer Iteration der *Percur* Berechnung in $Recur_1(F_1, E_1)$ auftaucht, da Spieler 1 von q_i aus garantieren kann, dass q_i wieder aufgesucht werden kann ohne Zustände aus E_1 zu besuchen. Wenn aber jedes q_i in $Percur_1(F_1, E_1)$ enthalten ist, dann muss auch $q \in Percur_1(F_1, E_1)$ gelten. Widerspruch.

Da Streett-Spiele nach Martin [Mar75] determiniert sind, liefert *Percur* auch den Gewinnbereich für Spieler 0. Es bleibt noch zu zeigen, dass $strat_0$ eine (positionale) Gewinnstrategie für Spieler 0 ist.

Es wird damit begonnen, zu zeigen, dass $\emptyset$ der größte Fixpunkt der G_i-Folge ist. Dies beruht auf den folgenden Beobachtungen: Kein $E_1 \cap W_0$-Zustand gehört zu einer G_i-Menge. Zweitens sind alle Transitionen von Spieler 0 in allen G_i-Teilgraphen dieselben wie im globalen Spielgraphen. Dies liegt daran, dass die G_i-Zustände all die Zustände sind, die nicht im Attraktor von S_{i-1} liegen. Jeder Zustand von Spieler 0 in G_i mit einer Transition nach T_i gehört zwangsläufig zum Attraktor von S_i, und somit gehört er nicht

zu G_{i+1}. Nehmen wir nun an, dass die Folge mit einer nicht leeren Menge konvergiert, also $G_n = G_{n+1}$. Dies bedeutet (i) $S_n = \emptyset$, da ansonsten $Attr_0(S_n) = T_n \neq \emptyset$ und $G_{n+1} \neq G_n$. Somit hat Spieler 1 von jedem Zustand in G_n eine Strategie, um einen Besuch von F_1 zu erreichen, während er in G_n verbleibt. Da kein Zustand aus E_1 besucht wird *und* der F_1-Zustand auch zu G_n gehört, kann dieser unendlich oft besucht werden. Da diese Strategie von Spieler 1 auf dem G_n-Teilgraphen auch eine Gewinnstrategie auf dem globalen Spiel ist (da keine Transitionen von Spieler 0 zuvor entfernt worden sind), gehört G_n nicht zum Gewinnbereich von Spieler 0. Dies ist ein Widerspruch dazu, dass $W_0 \supseteq G_0 \supseteq G_1 \supseteq \ldots \supseteq G_n$. Aus diesem Grund ist $strat_0$ für alle Knoten in W_0 definiert.

Es bleibt zu zeigen, dass $strat_0$ korrekt ist. Wir gehen mit einer Fallunterscheidung vor.

1. $q \in Attr_0(E_1 \cap W_0) \setminus (E_1 \cap W_0)$: Da gemäß der Attraktorstrategie gespielt wird, wird nach einer endlichen Anzahl Züge die Menge E_1 erreicht – auf dem Weg dorthin kann jeder Zustand aus F_1 höchstens einmal besucht werden.

2. $q \in E_1 \cap W_0$: Dies gleicht alle vorherigen Besuche von F_1-Zuständen aus. Jede beliebige Wahl einer Transition zu einem Zustand aus W_0 liefert eine Gewinnstrategie.

3. $q \in T_i \cup S_i$: Wir führen eine Induktion über $i \geq 1$. Der Induktionsbeginn ist $i = 1$. Wenn $q \in S_0$ gilt, so ist die Strategie eine Gewinnstrategie, da Spieler 0 Spieler 1 von dem Besuch von F_1 dauerhaft abhalten kann, solange Spieler 1 nicht nach G_0 wechselt. Dies würde zum ersten Fall führen. Nun nehmen wir für jedes $1 \leq j < i$ an, dass $strat_0$ von den Zuständen in T_j und S_j aus eine Gewinnstrategie ist. Es bleibt zu zeigen, dass dann $strat_0$ auch eine Gewinnstrategie von S_i und T_i aus ist.

   Sei $q \in S_i$. Durch die Anwendung von $strat_0$ bleibt die Partie entweder in dem Teilgraphen G_{i-1}, in dem $strat_0$ garantiert, dass F_1 nie besucht wird, oder sie bewegt sich zu einem Zustand außerhalb von G_{i-1}, welcher somit zu einem T_j mit $j < i$, gehört. Nach der Induktionsannahme folgt die Behauptung. Wenn $q \in T_i$ gilt, kann dasselbe Argument angewandt werden, um zu zeigen, dass entweder S_i aufgesucht wird, da das Spiel in G_{i-1} verbleibt, oder G_{i-1} verlassen wird und die Induktionsannahme angewandt werden kann. □

3.4.2.3 Komplexität

In diesem Abschnitt wird die worst-case-Laufzeit des gerade vorgestellten Algorithmus mit der Lösung durch Reduktion auf Paritätsspiele verglichen. Dabei sind die Paritätsspielalgorithmen von Zielonka, Jurdziński und Vöge/Jurdziński berücksichtigt worden. Begonnen wird mit einer Analyse des neuen Algorithmus.

Satz 3.22. *Die Laufzeit des Percur-Algorithmus ist* $\mathcal{O}((\frac{|Q|}{2})^2 \cdot |E|)$*. Somit können die Gewinnbereiche und -strategien für beide Spieler in einfachen Streett-Spielen in dieser Zeit berechnet werden.*

Beweis. Wenn die einzelnen Komponenten genauer betrachtet werden, stellt man fest, dass:

- für $Recur_1(F_1, E_1)$ höchstens $|F_1 + 1|$ $Attr_1^+$-Berechnungen gebraucht werden,
- jede $Attr_1$-Berechnung in Zeit $\mathcal{O}(|E|)$ durchgeführt werden kann (da $Attr_1^+(F_1, E_1)$ eine Modifikation der Attraktorkonstruktion ist, welche $\mathcal{O}(|E|)$ Zeit benötigt),
- der $Percur_1(F_1, E_1)$-Algorithmus selbst maximal $|E_1 + 1|$ Iterationen benötigt, da eine weitere Iteration nur dann weitere Zustände hinzufügt, wenn in einer vorhergehenden Iteration zumindest ein Zustand aus E_1 hinzugefügt worden ist.

Somit ergibt sich insgesamt die Schranke $\mathcal{O}(|E_1| \cdot |F_1| \cdot |E|) = \mathcal{O}((\frac{|Q|}{2})^2 \cdot |E|)$, da $|E_1| + |F_1| \leq |Q|$. Die Strategien für beide Spieler können in der Zeit $\mathcal{O}(|Q| \cdot |E|)$ bestimmt werden. □

3.4.2.4 Vergleich mit Algorithmen für Paritätsspiele

Wir vergleichen Satz 3.22 nun mit der Reduktion auf Paritätsspiele. Dabei wird der Algorithmus von McNaughton ([McN93]) nicht weiter berücksichtigt, da er in der Anzahl der Zustände eine exponentielle Komplexität hat. Die Variante von Zielonka ([Zie98]) ist hingegen interessanter. Sie ist dadurch gekennzeichnet, dass die Rekursionstiefe nur von der Anzahl der verwendeten Prioritäten abhängt. Das Prinzip dieses Algorithmus basiert darauf, dass anstatt eines einzelnen Pivotelementes die komplette Menge der Zustände mit der höchsten Farbe berücksichtigt wird. Es gibt Gemeinsamkeiten zwischen Zielonkas Algorithmus und dem neu vorgestellten – die Laufzeit ist in beiden Fällen $\mathcal{O}(|Q|^2 \cdot |E|)$ (für Zielonkas Algorithmus ist in [Jur00] die Laufzeit grob als $\mathcal{O}(|Q|^3 \cdot |E|)$ abgeschätzt worden, eine genauere Abschätzung liefert $\mathcal{O}(|Q|^2 \cdot |E|)$). Allerdings unterscheiden sich die Zwischenergebnisse der beiden Algorithmen.

In [Jur00] beschreibt Jurdziński einen Algorithmus, der auf der Idee der Fortschrittsmaße [KK91] beruht. Diese sind Zeugen für Gewinnstrategien in Paritätsspielen. Dabei

wird eine äquivalente Formulierung der Paritätsbedingung benutzt – Spieler 0 gewinnt eine Partie genau dann, wenn die kleinste, unendlich oft besuchte Farbe gerade ist. Der Algorithmus berechnet für solch ein Paritätsspiel die Fortschrittsmaße durch "Lifting"-Operationen eines initialen Fortschrittmaßes. Von dem resultierenden Fortschrittsmaß können dann die Gewinnbereiche und die Gewinnstrategie für Spieler 0 extrahiert werden. Der Algorithmus liefert keine Gewinnstrategie für Spieler 1.

Bemerkung 3.23 ([Jur00])**.** Der Algorithmus von Jurdziński berechnet für ein gegebenes Paritätsspiel mit c Farben die Gewinnbereiche für beide Spieler und die Gewinnstrategie für Spieler 0 in Zeit $\mathcal{O}((\frac{|Q|}{\lfloor |c|/2 \rfloor})^{\lfloor |c|/2 \rfloor} \cdot |E| \cdot |c|)$ mit Platz $\mathcal{O}(|Q| \cdot |c|)$.

Somit kann ein reduziertes, einfaches Streett-Spiel (mit drei Farben) in Zeit $\mathcal{O}((|Q| \cdot |E|)$ gelöst werden. Es liegt in der Natur des Algorithmus, dass er mehr Zeit benötigt, wenn der Gewinnbereich von Spieler 1 größer ist, besonders wenn er alle Knoten des Spiels umfasst. Der Grund dafür ist, dass dann mehr "Lifting"-Operationen nötig sind. In [Jur00] ist der Algorithmus als nicht-deterministisches Programm präsentiert worden. Die Reihenfolge, in der die Fortschrittsmaße bearbeitet werden sollen, und wie ein Knoten gefunden wird, an dem ein Fortschrittsmaß einem Lifting-Schritt unterzogen werden kann, ist nicht definiert. Beide Effekte haben in der Praxis einen Einfluss auf die Laufzeit. Eine effiziente Implementierung der "Lifting"-Operation ist in [EWS01] zu finden.

Der Algorithmus von Vöge und Jurdziński in [VJ00] gehört zu der Klasse der Strategieverbesserungsalgorithmen. Er basiert auf einem Algorithmus von Puri [Pur95], der optimale Strategien in Discount-Payoff-Spielen berechnet. Der gefärbte Spielgraph bei Vöge/Jurdziński muss zwei Bedingungen erfüllen: der Graph muss bipartit und die Färbungsfunktion injektiv sein. Jeder gefärbte Spielgraph kann in einen äquivalenten transformiert werden, der diese beiden Bedingungen erfüllt.

Die wichtigsten Notationen für diesen Algorithmus sind die Partieprofile und Bewertungen. Ein Partieprofil beschreibt die wichtigen Eigenschaften eines gegebenen Spiels. Auf dieser Basis sind die Bewertungen definiert. Diese weisen jedem Knoten des Spielgraphens ein Partieprofil für die Partie, die von diesem Knoten ausgeht, zu. Der Algorithmus wählt eine Bewertung (basierend auf einer zufällig gewählten Strategie von Spieler 0), welche optimal für Spieler 1 ist. Dann wird die Bewertung solange schrittweise verbessert, bis eine optimale Bewertung für beide Spieler gefunden ist, aus der dann die Gewinnbereiche und -strategien extrahiert werden können.

Bemerkung 3.24 ([VJ00])**.** Der Algorithmus von Vöge und Jurdziński berechnet für ein gegebenes Paritätsspiel die Gewinnbereiche und -strategien für beide Spieler in Zeit $\mathcal{O}(|E| \cdot |Q|^{|Q_0|+1})$ mit Platz $\mathcal{O}(|Q|^2 + |E|)$.

Obwohl die Laufzeit nur exponentiell abgeschätzt werden kann, ist kein Beispiel bekannt, was wirklich diese worst-case-Schranke erreicht. Es ist zudem auch kein Beispiel bekannt, das mehr als n^2 Iterationen benötigt.

Das Ergebnis dieses Vergleichs ist in Tabelle 3.6 zusammengefasst. Konstante Faktoren sind in der Zeitkomplexität weggelassen, da sie von der Implementierung abhängen.

Algorithmus	**Zeitkomplexität**
Percur-Algorithmus	$\mathcal{O}(\|Q\|^2 \cdot \|E\|)$
Zielonka	$\mathcal{O}(\|Q\|^2 \cdot \|E\|)$
Jurdziński	$\mathcal{O}(\|Q\| \cdot \|E\|)$
Vöge/Jurdziński	$\mathcal{O}(\|Q\|^{\|Q_0\|+1} \cdot \|E\|)$

Tabelle 3.6: Obere Laufzeitschranken für verschiedene Algorithmen zum Lösen von einfachen Streett-Spielen

3.4.2.5 Praktischer Vergleich

Nachdem im letzten Abschnitt die verschiedenen Algorithmen zur Lösung von einfachen Streett-Spielen auf theoretischer Ebene miteinander verglichen worden sind, soll in diesem Abschnitt ein praktischer Vergleich zwischen den verschiedenen Algorithmen zur Lösung von einfachen Streett-Spielen vorgestellt werden. Als Fallstudien werden die LSC-Spezifikationen betrachtet, die auch in der Vergangenheit genutzt worden sind, um die Funktionalität von REMoRDS [Bon05] zu demonstrieren. Genauere Spezifikationen finden sich in [Bon05, BHK03, Bon03].

coffee: die Spezifikation einer Kaffeemaschine, die einen Konflikt enthält und somit nicht realisierbar ist.

coffee': eine korrigierte Version der Spezifikation "Kaffeemaschine" (3 Szenarien).

ctas: ein Protokoll inspiriert von dem Initialisierungsteil des Central TRACON Automation System (CTAS). Dabei handelt es sich um ein Client-Server-System, in dem Clients eine Verbindung anfordern können und sich dann gemäß dem Initialisierungsprotokoll verhalten. Diese Spezifikation fordert von dem zukünftigen System, dass es eine mutual-exclusion-Eigenschaft bereitstellt, damit nicht gleichzeitig zwei Clients initialisiert werden. Allerdings liegt ein Fehler in der Spezifikation vor, die es der Umgebung ermöglicht, das System zu zwingen, die mutual-exclusion-Eigenschaft zu verletzen. Siehe auch [CTA, BHK03].

ctas': die korrigierte Version des CTAS Protokolls.

lcs: eine Lichtsteuerung.

pss: ein Kinderspiel (Papier, Stein, Schere), dass den Einfluss von Wahlmöglichkeiten und perfekter Information verdeutlicht.

pss': eine korrigierte Version des Kinderspiels.

Für zwei der obigen Spezifikationen wurden parametrisierte Versionen benutzt. Für das Kinderspiel "Papier, Stein, Schere" wurden weitere Wahlmöglichkeiten zu den ursprünglichen drei hinzugefügt. Hier kann jeder Spieler zwischen n Möglichkeiten $(k_0, \ldots, k_{n-1})$ wählen und Spieler 0 gewinnt genau dann, wenn Spieler 1 k_i auswählt und er selber $k_{i+1} \bmod n$. Im anderen Fall wird bei ctas eine Reihe von Spezifikationen betrachtet, die über die Anzahl der Clients im System parametrisiert sind.

Es wurde ein Rahmenwerk geschaffen, welches ermöglicht, die Laufzeiten zu vergleichen. Dazu wird die Experimentierplattform GASt (siehe auch Kapitel 5) verwendet, bei der die Algorithmen zur Lösung der Spiele in Java programmiert sind (mit Ausnahme des Algorithmus von Vöge und Jurdziński, bei dem eine existierende Implementierung in C benutzt wurde). Die Spielgraphen sind von "realen" anstelle von zufällig generierten Spezifikationen (siehe oben) erzeugt worden.

Alle Ergebnisse sind in Tabelle 3.7 festgehalten. In den ersten Spalten stehen Informationen zu der verwendeten Spezifikation, nämlich der Name, die Anzahl der Szenarien, aus der die Spezifikation besteht, die Anzahl der Locations, die als Größe der Spezifikation in [BS03, Bon05] definiert wurden, und ob die Spezifikation realisierbar ist oder nicht. Anzumerken ist, dass, wenn die Spezifikation nicht realisierbar ist, oftmals der Gewinnbereich von Spieler 0 leer ist. Dies beeinflusst deutlich die Performance des Algorithmus von Jurdziński (wie bereits im Abschnitt 3.4.2.3 erläutert). Dem Namen der Spezifikation ist der Wert des Parameters hinzugefügt, wenn diese parametrisiert ist. Zum Beispiel bedeutet "ctas-3" das Initialisierungsprotokoll basierend auf CTAS mit 3 Clients und "pps-4" steht für "Papier, Stein, Schere" mit 4 Wahlmöglichkeiten.

Es ist sehr schwer, eine formale Schlussfolgerung über die relative Performance dieser Algorithmen zu treffen. Allerdings ist der neu vorgestellte Algorithmus in vielen Fällen besser als die anderen diskutierten Algorithmen und ist nur in bestimmten Beispielen geringfügig schlechter.

Jurdzińskis Algorithmus hat bei diesen Testfällen am schlechtesten abgeschnitten. Obwohl er alle anderen bei einigen kleinen Beispielen deutlich geschlagen hat, hat er auf der anderen Seite bei nicht realisierbaren Spezifikationen sehr lange Laufzeiten, die

Spezifikation				Automat		Neuer	Ziel.	Vöge/	Jurd.
Name	LSCs	Loc.	Impl.	Zust.	Tran.	Alg.		Jurd.	
coffee	3	13	Nein	112	527	0.02	0.03	0.04	0.08
coffee'	3	13	Ja	64	272	0.01	0.01	0.03	0.01
lcs	8	28	Ja	1867	10163	0.68	0.60	3.27	0.54
ctas-2'	2	27	Ja	163	1016	0.07	0.07	0.07	0.07
ctas-2	2	27	Nein	1171	5239	0.46	0.75	1.69	7.09
ctas-3'	3	40	Ja	242	2230	0.16	0.17	0.11	0.15
ctas-3	3	40	Nein	14848	84983	9.57	13.12	199.35	1669.16
ctas-4'	4	53	Ja	321	3918	0.33	0.32	0.19	0.31
ctas-5'	5	67	Ja	400	6080	0.58	0.56	0.28	0.57
ctas-6'	6	81	Ja	479	8716	0.91	0.93	0.40	0.91
ctas-6	6	81	Nein	3362	55276	7.03	6.99	10.86	60.57
ctas-7'	7	95	Ja	558	11268	1.32	1.31	0.52	1.32
pss-3	13	68	Nein	314	987	0.14	0.15	0.18	0.21
pss-4'	16	86	Ja	933	3601	0.33	0.45	0.86	1.71
pss-4	16	86	Nein	476	1516	0.14	0.22	0.26	0.33
pss-5'	19	104	Ja	1321	5157	0.47	0.68	1.70	3.96
pss-5	19	104	Nein	672	2159	0.20	0.30	0.47	0.62
pss-6'	22	122	Ja	1777	6993	0.66	0.90	3.05	8.59
pss-6	22	122	Nein	902	2916	0.28	0.38	0.86	1.04
pss-7'	25	140	Ja	2301	9109	0.90	1.20	4.75	12.30
pss-7	25	140	Nein	1166	3787	0.37	0.55	1.28	1.57

Tabelle 3.7: Einfache Streett-Spiele: Laufzeiten der verschiedenen Algorithmen

mehrere Größenordnungen schlechter verglichen mit den anderen Algorithmen sind. Somit zeigt sich kein einheitliches Verhalten über alle Testfälle.

Der Algorithmus von Vöge und Jurdziński ist in dieser Beziehung besser. Bei einigen Testfällen schneidet er am besten ab, z.B. benötigt er die Hälfte der Laufzeit aller anderen in der Serie der CTAS Spezifikationen. Auf der anderen Seite gibt es auch Beispiele, bei denen er sich schlecht verhält, besonders bei der Lichtsteuerung und der nicht realisierbaren ctas-3 Spezifikation.

Zielonka's Algorithmus und der neu vorgestellte verhalten sich in Bezug auf die Einheitlichkeit über alle Testfälle am besten. Erstens beeinflusst die Größe der Gewinnbereiche die Laufzeit nicht stark und zweitens liefern sie für große Beispiele die mit Abstand besten Laufzeiten.

Abschließend halten wir als Ergebnis der theoretischen Analyse im Vergleich zur praktischen Evaluierung fest, dass die im vorangehenden Abschnitt bestimmten algorith-

mischen Laufzeiten keine klare Entsprechung mit den praktischen Resultaten haben. Zudem schneidet der von uns vorgestellte Algorithmus mit Zielonkas Algorithmus am besten ab (desweiteren sind auch leichte Vorteile gegenüber Zielonkas Algorithmus feststellbar).

Kapitel 4

Optimierung von Gewinnstrategien

Mit der Request-Response-Gewinnbedingung

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

wird nur gefordert, dass nach einem Besuch der P_i-Menge zu einem späteren Zeitpunkt der Besuch der Menge R_i erfolgt. Es wird nicht gefordert, dass dies möglichst schnell oder zu einem bestimmten Zeitpunkt geschieht. Dies ist somit nur eine qualitative Anforderung. Betrachten wir das Beispiel der Aufzugsteuerung (vgl. [WHT03]). Hier ist der Benutzer des Aufzugs daran interessiert, dass der Aufzug möglichst schnell kommt und ihn dann auch schnell zu seiner gewünschten Etage fährt.

Hier kommt der Wunsch nach der Optimierung von Strategien für den Controller ins Spiel. In diesem Zusammenhang steht "optimal" nicht für die benötigte Speichergröße des Controllers, sondern für die Güte des Gewinnens bzw. Verlierens. Man möchte also eine quantitative Aussage über eine Partie treffen können, d. h. einer Partie einen numerischen Wert zuordnen, der eine Aussage über die Güte der Partie macht. Hierbei wird unterschieden, *wie* gewonnen oder verloren wird. Wir unterscheiden nicht nur, ob Spieler 0 das Spiel gewinnt oder verliert (qualitative Aussage), sondern es wird ein Maß eingeführt, welches im Weiteren *Partiewert* genannt wird und bestimmt, *wie gut* eine Partie gewonnen bzw. verloren wird. Wenn man auf das Beispiel der Aufzugsteuerung zurückkommt, wird der Partiewert die Wartezeiten für den Benutzer berücksichtigen.

Die Vorgehensweise besteht zunächst darin, Maße für Partien einzuführen und daran anknüpfend zu definieren, was unter einer optimalen Strategie verstanden werden soll. Für einfache Erreichbarkeitsspiele ist die Definition eindeutig – die Zielmenge soll mit der minimal nötigen Anzahl von Zügen erreicht werden. Dies wird bereits durch die Attraktorkonstruktion erreicht. Für eine Menge von RR-Bedingungen ist die Situation

komplizierter, sowohl aufgrund der unendlich häufigen Erfüllung von Bedingungen als auch wegen des Zusammenspiels mehrerer Bedingungen.

Im Bereich der Payoff Spiele [EM79, GZ05] ist die Frage nach optimalen Strategien in den letzten Jahren intensiv studiert worden. Gimbert und Zielonka haben in [GZ06] interessante Verbindungen zwischen Paritätsspielen und verschiedenen Discounted-Spielen vorgestellt, immer auch im Hinblick auf die Frage nach optimalen Strategien. Die Ergebnisse sind eine Verallgemeinerung der Arbeit von de Alfaro, Henzinger und Majumdar [dAHM03] über den Discounted μ-Kalkül. Wiederum Gimbert und Zielonka haben in der Arbeit [GZ05] ein notwendiges und hinreichendes Kriterium vorgestellt, wann Spiele mit einem Payoff Mapping optimale positionale Gewinnstrategien für beide Spieler haben.

Hauptziel dieses Kapitels ist ein Ergebnis, welches einen Zusammenhang zwischen RR-Spielen und Mean-Payoff-Spielen herstellt und damit (wenigstens prinzipiell) die Bestimmung optimaler Gewinnstrategien für RR-Spiele erlaubt. Schlüsselresultat ist die Angabe einer uniformen Schranke für die Wartezeiten in RR-Spielen, die für optimale Gewinnstrategien gilt. Damit lässt sich die Bestimmung optimaler Strategien auf das Lösen von Mean-Payoff-Spielen über endlichen Graphen zurückführen. Somit gehen wir noch einen Schritt weiter als bei den finitary Paritäts- bzw. finitary Streett-Spielen von Chatterjee und Henzinger [CH06, Hor07]. Dort wird nur gefordert, dass es eine endliche Schranke gibt, die jedoch nicht aus der Gewinnbedingung und dem Spielgraphen abgeleitet werden kann.

4.1 Partiebewertungen

Um eine Definition optimaler Strategien für RR-Spiele zu ermöglichen, werden zunächst einige Beobachtungen vorgestellt. Die Güte einer Partie in einem Request-Response-Spiel hängt direkt mit den *Wartezeiten* zusammen. Dazu werden zuerst formal einige Begriffe eingeführt.

Definition 4.1. In einer Partie ρ eines RR-Spiels heißt eine Bedingung (P_i, R_i) zum Zeitpunkt t

(a) *aktiv*, wenn gilt:

$$\exists t' \leq t : \rho(t') \in P_i \wedge \forall s \in [t', t] : \rho(s) \notin R_i \ .$$

Dies wird auch durch die Funktion $active(\rho, t, i)$ ausgedrückt, die wie folgt definiert ist:

$$active(\rho, t, i) := \begin{cases} 1, & \text{in Partie } \rho \text{ ist Bedingung } i \text{ im Zug } t \text{ aktiv} \\ 0, & \text{sonst} \end{cases}$$

Der Parameter i der Funktion bezeichnet dabei die i-te RR-Bedingung (P_i, R_i).

(b) gerade *aktiviert*, wenn gilt:

$$\neg active(\rho, t-1, i) \wedge active(\rho, t, i)$$

Die Funktion $newActive(\rho, t, i)$ ist die zugehörige boolesche Funktion:

$$newActive(\rho, t, i) := \begin{cases} 1, & \neg active(\rho, t-1, i) \wedge active(\rho, t, i) \\ 0, & \text{sonst} \end{cases}$$

Die Funktion *newActive* liefert in der Partie ρ zum Zeitpunkt t genau dann eine 1 zurück, wenn die i-te RR-Bedingung nach einer Erfüllung das erste Mal wieder oder erstmalig überhaupt aktiviert wird.

(c) gerade *erfüllt*, wenn gilt:

$$active(\rho, t-1, i) \wedge \rho(t) \in R_i$$

Definition 4.2 (Wartezeit)**.** In einer Partie ρ eines RR-Spiel ist die Wartezeit einer RR-Bedingung (P_i, R_i) zum Zeitpunkt t die Zeit, die seit der letzten Aktivierung der Bedingung vergangen ist. Formal definieren wir:

$$WZ(\rho, t, i) := \begin{cases} 0 & \neg active(\rho, t, i) \\ t - t' + 1 & \exists t' \leq t : newActive(\rho, t', i) \wedge \forall s \in [t', t] : active(\rho, s, i) \end{cases}$$

In einer Partie ist die Wartezeit bezüglich einer RR-Bedingung lokal definiert. Global kann die Wartezeit in einer Partie bezüglich einer RR-Bedingung entweder

- beschränkt sein, wenn es eine Schranke gibt, so dass alle Wartezeiten in der Partie bezüglich dieser RR-Bedingung immer kleiner oder gleich dieser Schranke sind, oder
- unbeschränkt sein, wenn es keine solche Schranke gibt, aber in der Partie immer auf eine Aktivierung der RR-Bedingung auch eine Erfüllung folgt, oder
- undefiniert sein, wenn es eine Aktivierung der RR-Bedingung gibt, auf die keine Erfüllung folgt.

Definition 4.3 (Warteschranke)**.** Die Warteschranke $wb(\rho, i)$ für die i-te RR-Bedingung in der Partie ρ ist das Maximum aller Wartezeiten dieser Bedingung in der Partie ρ, wenn dieses existiert, sonst der Wert "unendlich" (geschrieben als ∞).

Bemerkung 4.4. Die Warteschranke $wb(\rho, i)$ für die i-te RR-Bedingung in der Partie ρ ist unendlich, wenn in der Partie ρ entweder diese Bedingung irgendwann nicht mehr erfüllt wird oder die Folge der Wartezeiten unbeschränkt ist. Sonst ist $wb(\rho, i)$ endlich.

Definition 4.5. Für eine Partie ρ in einem RR-Spiel heißt ein RR-Paar (P_i, R_i) *unbeschränkt*, wenn die Wartezeiten für dieses Paar in der Partie ρ unbeschränkt sind, kurz $wb(\rho, i) = \infty$ ist, aber zusätzlich für die Partie ρ gilt, dass auf jede Aktivierung der Bedingung auch eine Erfüllung erfolgt. Ein RR-Paar (P_i, R_i) heißt bezüglich einer Strategie σ von Spieler 0 unbeschränkt, wenn es eine gemäß der Strategie σ gespielte Partie ρ gibt, in welcher das Paar unbeschränkt ist. Ist das RR-Paar in allen gemäß σ gespielten Partien beschränkt, heißt es auch bezüglich der Strategie σ beschränkt.

Definition 4.6. Eine Partie ρ eines RR-Spiels mit r Paaren kann um folgende Vektoren erweitert werden:

- den *Wartezeitvektor* $\begin{pmatrix} WZ(\rho,t,1) \\ \vdots \\ WZ(\rho,t,r) \end{pmatrix} \in \mathbb{N}^r$ zur Zeit t, der für jede RR-Bedingung die Wartezeit gemäß Definition 4.2 angibt;
- den *Aktivvektor* $\begin{pmatrix} \mathrm{sg}(WZ(\rho,t,1)) \\ \vdots \\ \mathrm{sg}(WZ(\rho,t,r)) \end{pmatrix} = \begin{pmatrix} active(\rho,t,1) \\ \vdots \\ active(\rho,t,r) \end{pmatrix} \in \mathbb{B}^r$, der für jede RR-Bedingung angibt, ob sie zur Zeit t gerade aktiv ist.

Definition 4.7. Für ein RR-Spiel mit n Zuständen und r Bedingungen lässt sich der zugrunde liegende Spielgraph (Q, E) mit $Q = Q_0 \dot\cup Q_1$ um den Wartezeitvektor wie folgt erweitern:

- $Q' := Q'_0 \cup Q'_1$ mit $Q'_i := Q_i \times \mathbb{N}^r$
- $((q, i_1, \ldots, i_r), (q', i'_1, \ldots, i'_r)) \in E' :\Leftrightarrow$
  - $(q, q') \in E$
  - $i'_k = \begin{cases} i_k + 1, & \text{falls } i_k > 0 \wedge q' \notin R_k \\ 0, & \text{falls } (q' \in R_k) \vee (i_k = 0 \wedge q' \in Q \setminus P_k) \\ 1, & \text{falls } i_k = 0 \wedge q' \in P_k \setminus R_k \end{cases}$
- Die Initialisierungsfunktion $f : Q \to Q'$ transformiert die Zustände des gegebenen Spielgraphen in jene des erweiterten Spielgraphen wie folgt:
  $f(q) = (q, i_1, \ldots, i_r)$ mit $i_k = \begin{cases} 1, & \text{falls } q \in P_k \\ 0, & \text{sonst} \end{cases}$

Dieser um die Wartezeitvektoren erweiterte Spielgraph wird im Weiteren als G_r^+ bezeichnet. Können die Wartezeiten für alle RR-Bedingungen in allen möglichen Partien durch eine Schranke w_{max} beschränkt werden, bekommt man den Spielgraphen $G_{r|w_{max}}^+$ als Variante des G_r^+, bei dem alle Werte der Wartezeiten durch w_{max} beschränkt sind. Wir erhalten:

- $Q_i' := Q_i \times \{0, \ldots, w_{max} + 1\}^r$
- $((q, i_1, \ldots, i_r), (q', i_1', \ldots, i_r')) \in E' :\Leftrightarrow$
  - $(q, q') \in E$
  - $i_k' = \begin{cases} i_k + 1, & \text{falls } 1 \leq i_k \leq w_{max} \wedge q' \notin R_k \\ 0, & \text{falls } q' \in R_k \\ 1, & \text{falls } i_k = 0 \wedge q' \in P_k \setminus R_k \\ i_k, & \text{sonst} \end{cases}$

    Im Sonstfall liegt der Wert $i_k = w_{max} + 1$ oder $i_k = 0$ vor.

Bemerkung 4.8.

- Eine Partie ρ in G induziert eindeutig eine Partie ρ' in G_r^+ bzw. $G_{r|w_{max}}^+$ (wenn eine solche Schranke w_{max} existiert).
- Der Spielgraph G_r^+ ist unendlich, wenn man die Wartezeiten nicht für alle RR-Paare und alle möglichen Partien beschränken kann und somit einen Spielgraphen $G_{r|w_{max}}^+$ erhält.
- Ein RR-Spielgraph lässt sich analog auch um den Aktivvektor an Stelle des Wartezeitvektors erweitern. Dann erhält man aus G_r^+ selbstverständlich einen endlichen Spielgraphen.

Lemma 4.9 (Bounded Waiting Time)**.** *Sei σ eine Automatengewinnstrategie im RR-Spiel über dem endlichen Spielgraphen G von q aus. Jede Partie ρ gemäß σ von q aus hat eine beschränkte Wartezeit bezüglich jeder RR-Bedingung.*

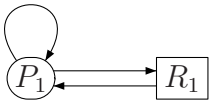
Beweis. Es sei die Gewinnstrategie σ durch einen Strategieautomaten mit m Zuständen gegeben. Ist ρ eine Partie gemäß σ und $t \geq 0$, so wird jede bei t aktive Bedingung irgendwann erfüllt. Wir zeigen, dass die maximale Wartezeit $n \cdot m$ (mit $n = |Q|$) ist. Angenommen, die RR-Bedingung (P_i, R_i) sei nach Aktivierung bei t nach $n \cdot m + 1$ Zügen noch nicht erfüllt. Dann wird mindestens ein Zustand des Strategieautomaten zwischen t und $t + m \cdot n + 1$ zweimal besucht. Hieraus gewinnen wir sofort eine Partie ρ' gemäß σ, mit $\rho'(x) = \rho(x)$ für $x \leq t$, in der für $y > t$ kein R_i-Zustand besucht wird. σ ist demnach keine Gewinnstrategie. Widerspruch. □

Korollar 4.10. *Lemma 4.9 gilt nicht für folgende Abschwächungen der Voraussetzungen:*

a) Gewinnstrategien von Spieler 0, die unendlichen Speicher haben;

b) unendliche Spielgraphen.

Beweis. a) Betrachten wir folgenden Spielgraphen:



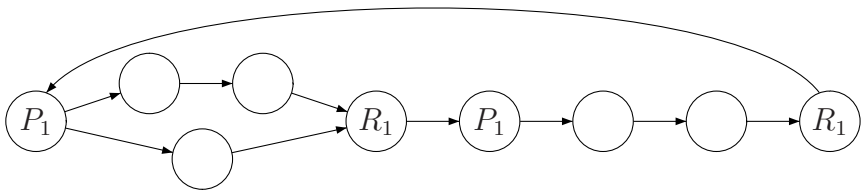
Wir konstruieren eine Gewinnstrategie, bei der sich die Wartezeit immer weiter jeweils um eins erhöht. Dann ist offensichtlich, dass keine globale Schranke für die Wartezeit existiert.

Ein möglicher Strategieautomat, der diese Strategie realisiert, benötigt unendlichen Speicher. Die Zustände des Automaten bestehen dabei aus zwei Komponenten. Die erste Komponente speichert die Wartezeit bei der letzten Erfüllung, die zweite Komponente ist ein Zähler für die aktuelle Wartezeit. Ist der Wert in der zweiten Komponente größer als in der ersten Komponente, wird die RR-Bedingung erfüllt und die Wartezeit in der ersten Komponente abgespeichert. Formal ist der Zustandsraum des Strategieautomaten gegeben durch $\mathbb{N} \times \mathbb{N}$.

b) Der in Abbildung 4.1 dargestellte unendliche Spielgraph zeigt, dass sich Lemma 4.9 nicht auf unendliche Spielgraphen übertragen lässt, da Spieler 1 eine immer längere Wartezeit bis zur Erfüllung der Anforderung erzwingen kann. Somit gibt es keine endliche Schranke der Wartezeit über diesem unendlichen Spielgraphen. □

Damit ist gezeigt worden, dass die Annahmen von Lemma 4.9 auf endliche Spielgraphen und Automatengewinnstrategien wirklich nötig sind.

Das folgende RR-Spiel zeigt, dass sich eine "optimale" Strategie nicht nur anhand der globalen Schranken für die Wartezeiten der einzelnen Bedingungen definieren lässt. Wir betrachten ein Ein-Personen-Spiel mit folgendem Transitionsgraphen und einer RR-Bedingung (P_1, R_1).



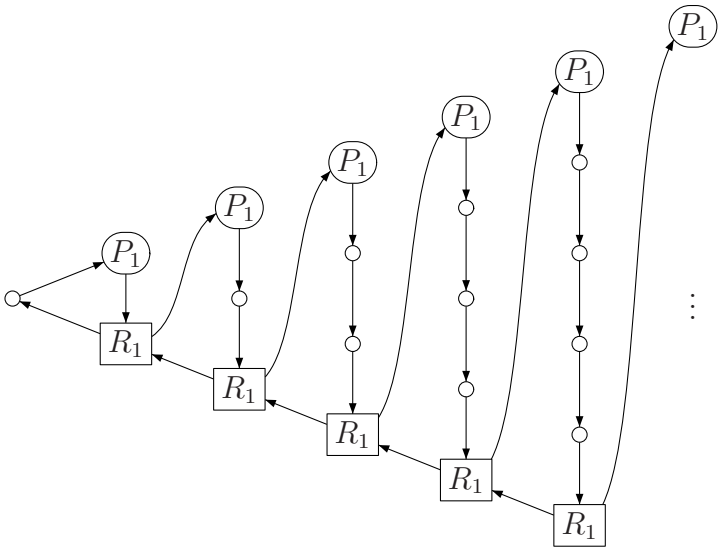


Abbildung 4.1: Unendlicher Spielgraph, in dem Lemma 4.9 nicht gilt.

Die globale Schranke der Wartezeit für die einzige RR-Bedingung (P_1, R_1) ist 3. Würde nur die globale Schranke für die Definition von Optimalität berücksichtigt, dann wären die beiden möglichen Strategien in diesem Spielgraphen gleichwertig. Allerdings ist die Strategie, die bei der Wahlmöglichkeit den unteren Weg nimmt, besser, da dort die lokale Wartezeit nur 2 beträgt. Um der Definition einer optimalen Strategie für RR-Spiele näher zu kommen, betrachten wir zunächst einzelne Partien und definieren dafür geeignete Bewertungen, die etwas über die Länge der Wartezeiten aussagen sollen.

In den folgenden Unterabschnitten werden drei verschiedene Definitionen für Partiewerte von Request-Response-Spielen vorgestellt und ihre Vor- und Nachteile diskutiert: die durchschnittliche Anzahl aktiver Anforderungen, die durchschnittliche Wartezeit und die quadratische Gewichtung der Wartezeiten. Ziel dabei ist es, eine geeignete Definition zu finden, die es erlaubt, eine optimale Strategie bei Request-Response-Spielen zu definieren. In allen drei Fällen sind die Partiewerte ≥ 0 und wachsen mit den Wartezeiten. Optimale Strategien sollen also möglichst kleine Partiewerte garantieren.

4.1.1 Durchschnittliche Anzahl aktiver Anforderungen

Eine erste, naheliegende Überlegung für einen Partiewert eines Request-Response-Spiels besteht darin, zu berücksichtigen, wie viele Anforderungen durchschnittlich aktiv sind. Dabei ist eine Anforderung genau dann zum Zeitpunkt t aktiv, wenn nach dem letzten Besuch eines Zustands aus der Anforderungsmenge noch kein Zustand aus der passenden Antwortmenge bis zum Zeitpunkt t aufgesucht worden ist.

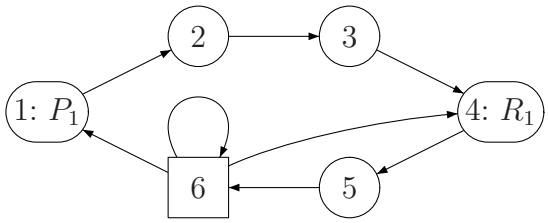
Definition 4.11 (Partiewert $a(\rho)$)**.** Gegeben sei ein RR-Spiel mit r Bedingungen. Dann ist der *Partiewert* $a(\rho)$ für eine Partie ρ definiert als

$$a(\rho) := \limsup_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n} \sum_{i=1}^{r} \mathit{active}(\rho, t, i)$$

mit $\mathit{active}(\rho, t, i)$ aus Definition 4.1 (a).

Der Partiewert $a(\rho)$ entspricht somit der durchschnittlichen Anzahl aktiver Bedingungen der Partie ρ. Es ist klar, dass dieser Limes in der Berechnung von $a(\rho)$ existiert, da immer Werte zwischen 0 und r aufsummiert werden.

Beispiel 4.12. Gegeben sei folgendes RR-Spiel – dabei ist in den Zuständen vermerkt, in welchen P_i bzw. R_i-Mengen sie enthalten sind:



Für die Partie $\rho = \overline{1, 2, 3, 4, 5, 6}$ (dabei symbolisiert der durch den oberen Strich markierte Teil den sich immer wiederholenden Teil der Partie) ist der Partiewert $a(\rho) = 0.5$, da die Hälfte der Zeitpunkte das RR-Paar (P_1, R_1) aktiv ist und die andere Hälfte über nicht. Für die Partie $\rho' = 1, 2, 3, \overline{4, 5, 6}$ ist der Partiewert $a(\rho') = 0$, da in der sich wiederholenden Schleife nie eine RR-Bedingung aktiviert wird.

Bevor auf die Definition näher eingegangen wird, folgen zunächst einige Bemerkungen, die für diese und die beiden noch folgenden Partiewert-Definitionen gelten. Es wird immer der erste Zeitpunkt einer Anforderung betrachtet, auch wenn eine Bedingung mehrfach ohne zwischenzeitliche Erfüllung angefordert wird. Eine andere Variante wäre es, dass pro Anforderung eine Erfüllung erfolgen muss (also dass nicht mehr ein Besuch der Antwortmenge ausreicht, alle vorherigen Anforderungen zu erfüllen). Dies geht aber über die ursprüngliche Request-Response-Gewinnbedingung hinaus und wird daher nicht weiter verfolgt.

Bemerkung 4.13 (Diskussion des Partiewertes $a(\rho)$)**.** Der Ansatz, $a(\rho)$ als Grundlage für Optimalität von Strategien zu nehmen, also zu fordern, dass es jeweils möglichst wenige aktive RR-Bedingungen gibt, eignet sich nicht. Dies ist dadurch begründet, dass

die Möglichkeit besteht, dass eine Bedingung ohne negative Auswirkung im Partiewert immer unerfüllt bleibt, obwohl Spieler 0 diese durchaus erfüllen könnte. Damit ist gemeint, dass Spieler 0 u. U. bei Nichterfüllung einer Bedingung im Durchschnitt weniger aktive Bedingungen hat, als wenn er alle erfüllt. Dies steht im Gegensatz zur Idee, dass Wartezeiten zu minimieren sind. Als Beispiel dafür betrachten wir den Spielgraphen aus Abbildung 4.2.

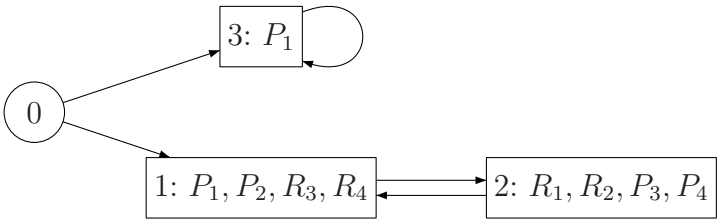
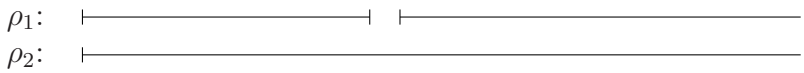


Abbildung 4.2: Minimaler Partiewert $a(\rho)$ muss nicht gewinnbringend sein, obwohl Spieler 0 eine Gewinnstrategie hat

Entscheidet sich Spieler 0 im Zustand 0 für die Partie $0, \overline{3}$ so ist der Partiewert 1. Ein höherer Wert ergibt sich, wenn sich Spieler 0 für die Partie $0, \overline{1, 2}$ entscheidet. Der kleinere Wert signalisiert hier also den Verlust einer Partie.

Weiterhin kann die Möglichkeit bestehen, dass Bedingungen sehr lange aktiv bleiben, womit ein geringerer Durchschnittswert erreicht wird. Dies widerspricht aber auch dem Gedanken einer optimalen Strategie – zumindest wenn auch der Aspekt der Zeit zwischen Anforderung und Erfüllung eine Rolle spielen soll. Des weiteren differenziert dieser Ansatz (bedingt durch die Limes-Bildung) gewisse Fälle nicht, die aus der praktischen Sicht einen deutlichen Unterschied aufweisen. Wir betrachten die Fälle, dass eine Bedingung schließlich nicht mehr erfüllt wird (Partie ρ_2), und dass in der Partie ρ_1 die Bedingung zwischendurch einmal erfüllt wird und kurz darauf wieder aktiviert wird. Die folgende Figur illustriert dies:



Dabei soll in der Abbildung der durchgezogene Strich andeuten, dass eine Anforderung aktiv ist. Der Limes liefert für diese Konstellation in beiden Fällen den gleichen Wert, obwohl sich die Partien intuitiv unterscheiden. Für den in Abbildung 4.3 dargestellten Request-Response-Spielgraphen wäre die Partie ρ_1 durch $0, 1, 2, \overline{1}$ und ρ_2 durch $0, \overline{1}$ gegeben. Es gilt, dass $a(\rho_1) = a(\rho_2) = 1$.

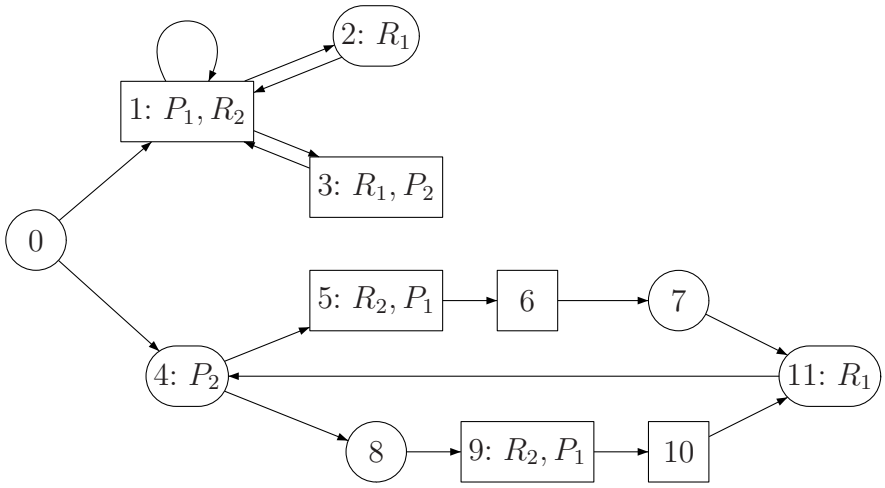
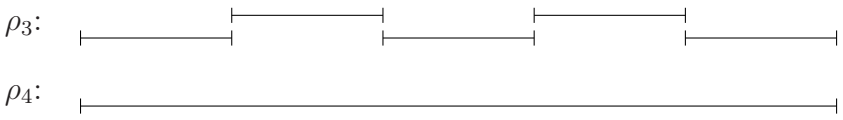


Abbildung 4.3: Spielgraph mit RR-Paaren

Einen noch deutlicheren Unterschied gibt es zwischen den folgenden beiden Partien ρ_3 und ρ_4 mit jeweils zwei RR-Bedingungen, wieder in der bildlichen Darstellung:



In der Partie ρ_3 wechselt sich die Aktivierung der beiden RR-Bedingungen immer wieder ab, während in der zweiten Partie eine Anforderung von Beginn an aktiv ist und nicht erfüllt wird. Der Unterschied zwischen den beiden Partien besteht darin, dass die erste Partie von Spieler 0 gewonnen wird und die zweite von Spieler 1. Dies drückt sich aber nicht in den Partiewerten $a(\rho_3)$ und $a(\rho_4)$ aus. Beide haben den gleichen Wert 1, da im Mittel genau eine Bedingung aktiv ist. Somit liefert der Partiewert $a(\rho)$ keine Information zur Bestimmung des Gewinners einer Partie ρ in einem Request-Response-Spiel. In dem Spielgraphen aus Abbildung 4.3 entspricht ρ_3 der Partie $0, \overline{1, 3}$ und ρ_4 der Partie $0, \overline{1}$.

Eine weitere, noch gewichtigere Eigenschaft von $a(\rho)$ ist die, dass bei Grundlage von $a(\rho)$ für die Optimalität nicht sichergestellt ist, dass optimale Gewinnstrategien endlichen Speicher haben. Betrachten wir dazu den Spielgraphen in Abbildung 4.4.

In der Schleife mit den Zuständen 1 bis 4 sind nach dem ersten Durchlauf immer genau 2 Anforderungen aktiv (RR-Paar 1 und RR-Paar 2 oder 3). Entscheidet sich aber Spieler 0 im Zustand 1 zum Knoten 5 zu gehen, sind alle drei Anforderungen aktiv. Somit ist es für Spieler 0 im Hinblick auf einen niedrigen Partiewert $a(\rho)$ sinnvoll, möglichst oft die Schleife 1 bis 4 zu durchlaufen und die obere Schleife nur sehr selten

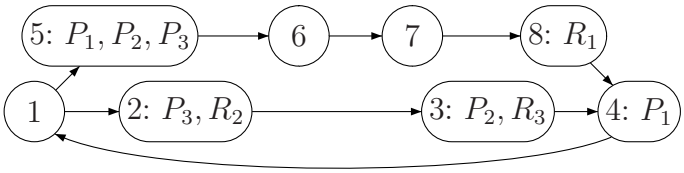


Abbildung 4.4: Optimale Strategie gemäß $a(\rho)$ muss keinen endlichen Speicher haben

zu benutzen. Trotz allem muss die obere Schleife von Zeit zu Zeit durchlaufen werden, da sonst Spieler 0 die Partie verliert. Also folgt: Je mehr Speicher für die Strategie verwendet wird, um so besser wird der dazugehörige Strategiewert.

Alle diese Beispiele führen uns dazu, den Partiewert $a(\rho)$ nicht als Grundlage zur Definition einer optimalen Strategie in unseren weiteren Betrachtungen zu verwenden.

4.1.2 Durchschnittliche Wartezeit

Da der Partiewert $a(\rho)$ einige Schwachpunkte hat, wie im letzten Abschnitt gezeigt, wollen wir jetzt eine andere Definition des Partiewertes vorschlagen, die diese Nachteile nicht mehr hat. Anstatt die durchschnittliche Anzahl offener Anforderungen zu betrachten, nehmen wir jetzt die durchschnittliche Wartezeit bis zur Erfüllung einer Anforderung als Kriterium.

Definition 4.14 (Partiewert $t(\rho)$)**.** Gegeben sei ein RR-Spiel mit r Bedingungen. Dann ist der *Partiewert* $t(\rho)$ für eine Partie ρ wie folgt definiert:

$$t(\rho) := \limsup_{n\to\infty} \frac{\sum_{t=1}^{n}\sum_{i=1}^{r} \mathit{active}(\rho,t,i)}{\sum_{t=1}^{n}\sum_{i=1}^{r} \mathit{newActive}(\rho,t,i)}$$

mit $\mathit{active}(\rho,t,i)$ und $\mathit{newActive}(\rho,t,i)$ aus Definition 4.1.

Der Partiewert $t(\rho)$ gibt also die durchschnittliche Wartezeit bis zur Erfüllung der Bedingungen an.

Beispiel 4.15. Betrachtet wird derselbe RR-Spielgraph wie im Beispiel 4.12 und die Partie $\rho = \overline{1,2,3,4,5,6}$. In dem sich wiederholenden Zykel liefert $\mathit{active}(\rho,t,1)$ den Wert 1 für $1 \leq t \leq 3$ und $\mathit{newActive}(\rho,t,1)$ den Wert 1 nur für $t = 1$. Somit ergibt sich der Partiewert $t(\rho) = 3$.

Bemerkung 4.16 (Existenz des Limes)**.** Falls Spieler 0 die Partie ρ gewinnt und die Partie gemäß einer Automatengewinnstrategie gespielt hat, existiert der Wert $t(\rho)$. Gewinnt Spieler 1 die Partie, existiert der Limes nicht ($t(\rho) = \infty$).

Bemerkung 4.17 (Diskussion des Partiewertes $t(\rho)$)**.** Die Größe $t(\rho)$ misst die durchschnittliche Zeit in einer Partie zwischen Aktivierung und Erfüllung einer RR-Bedingung. Bei diesem Partiewert ergeben sich für das Beispiel mit den Partien ρ_3 und ρ_4 von der Diskussion von $a(\rho)$ auch unterschiedliche Werte für $t(\rho_3)$ und $t(\rho_4)$. Deshalb ist $t(\rho)$ besser zur Definition einer optimalen Strategie geeignet als $a(\rho)$.

Betrachten wir die zwei Partien ρ_5 und ρ_6, die dem Schema der nachfolgenden Figur entsprechen. Beide haben einen Zyklus über fünf Züge. In der Partie ρ_5 bleibt die erste RR-Bedingung einen Zug lang und die zweite Bedingung drei Züge lang aktiv. Hingegen sind in der Partie ρ_6 beide Bedingung jeweils zwei Züge aktiv.

Im Spielgraph aus Abbildung 4.3 entspricht ρ_5 der Partie $0, \overline{4, 5, 6, 7, 11}$ und ρ_6 der Partie $0, \overline{4, 8, 9, 10, 11}$. Bestimmt man die beiden Partiewerte $t(\rho_5)$ und $t(\rho_6)$, stellt man fest, dass beide denselben Wert 2 haben, da die durchschnittliche Dauer, bis eine Anforderung erfüllt wird, gleich ist. Gewichtet man alle Bedingungen gleich und zielt (wie zumindest in der Praxis gefordert) auf eine möglichst niedrige obere Schranke für alle Wartezeiten, sollten längere Wartezeiten vermieden werden und somit stärker ins Gewicht fallen. Die beiden Partien $t(\rho_5)$ und $t(\rho_6)$ unterscheiden sich in diesem Punkt und haben trotzdem denselben Partiewert. Wir fangen dies im nachfolgenden Abschnitt durch eine verfeinerte Definition ab.

4.1.3 Quadratische Gewichtung der Wartezeit

Um eine längere Wartezeit zu bestrafen, wird bei der nächsten Definition die Wartezeit bis zur Erfüllung einer Bedingung stärker berücksichtigt.

Definition 4.18 (Partiewert $w(\rho)$)**.** Gegeben sei ein RR-Spiel mit r Bedingungen. Dann ist der *Partiewert* $w(\rho)$ für eine Partie ρ wie folgt definiert:

$$w(\rho) := \limsup_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n} \sum_{i=1}^{r} WZ(\rho, t, i)$$

mit $WZ(\rho, t, i)$ aus Definition 4.2.

Beispiel 4.19. Betrachtet wird derselbe RR-Spielgraph wie im Beispiel 4.12 und die Partie $\rho = \overline{1, 2, 3, 4, 5, 6}$. In dem sich wiederholenden Zykel liefert $WZ(\rho, t, 1)$ den Wert 1 für $t = 1$, den Wert 2 für $t = 2$, den Wert 3 für $t = 3$ und sonst den Wert 0. Somit ergibt sich ein Partiewert $w(\rho) = 1$.

Für die Existenz dieses Limes gilt das gleiche wie beim Partiewert $t(\rho)$ (siehe Bemerkung 4.16).

Bemerkung 4.20 (Diskussion des Partiewertes $w(\rho)$)**.** Beim Ansatz $w(\rho)$ fließen die Zeiten bis zur Erfüllung quadratisch ein, indem zu jedem Zeitpunkt, während dessen die Bedingung aktiv ist, die Zeit seit der Aktivierung aufsummiert wird ($\sum_{i=0}^{n} i = \frac{n\cdot(n+1)}{2}$). Somit fallen längere Zeiten bis zur Erfüllung stärker ins Gewicht als kürzere. Dies hat zur Folge, dass aus den auf Seite 70 genannten Partien ρ_3 und ρ_4 unterschiedliche Partiewerte resultieren. Aber auch bei den Partien ρ_5 und ρ_6 von Seite 72 ist dies der Fall. Die Partie ρ_5 mit den beiden unterschiedlichen Wartezeiten für die Bedingungen bekommt hier einen höheren Wert als ρ_6 mit gleichen Wartezeiten für beide Bedingungen, entsprechend der Forderung, längere Wartezeiten stärker zu berücksichtigen.

Eine Variante dieses Ansatz besteht darin, wie bei $t(\rho)$ durch die Anzahl der Zeitpunkte anstatt durch die Zuganzahl zu dividieren, während derer eine Bedingung aktiviert wird (siehe Definition von *newActive*). Somit würde eine durchschnittliche quadratische Zeit bis zur Erfüllung einer Anforderung ermittelt. Diese Variante bringt keine für unsere Diskussion relevanten Vorteile gegenüber $w(\rho)$ und hat den Nachteil, dass die algorithmische Bestimmung des Werts einer Strategie schwieriger ist.

Für alle drei Definitionen gilt, dass für Spieler 0 ein niedriger Wert besser ist. Eine Zusammenfassung der Diskussion der Partiebewertungen $a(\rho)$, $t(\rho)$ und $w(\rho)$ zeigt die folgende Tabelle – hierbei enthält die letzte Zeile ein Ergebnis der nachfolgenden Abschnitte.

	$a(\rho)$	$t(\rho)$	$w(\rho)$
Bestimmung des Gewinners der Partie ρ möglich	–	√	√
Berücksichtigung der Zeit bis Erfüllung	nur indirekt	√	√
Größere Gewichtung längerer Wartezeiten	–	–	√
Existenz optimaler Strategien mit endlichem Speicher	–	√	√

Tabelle 4.1: Vergleich der verschiedenen Partiewerte

Aufgrund dieses Vergleiches der unterschiedlichen Ansätze eines Partiewertes wird im weiteren Verlauf die Definition $w(\rho)$ benutzt. Die Entscheidung ist so ausgefallen, da nur bei $w(\rho)$ sowohl die Zeiten berücksichtigt werden, bis eine Anforderung erfüllt wird, als auch der Umstand, dass längere Wartezeiten stärker ins Gewicht fallen als kürzere.

4.1.4 Strategiewerte und optimale Gewinnstrategien

Wir definieren in diesem Abschnitt den Wert einer Strategie ausgehend von Werten der durch die Strategie möglichen Partien. Ausgangspunkt ist folgender offensichtlicher Sachverhalt: Wenn beide Spieler gemäß ihren Strategien σ bzw. τ spielen und die Partie im Zustand q_0 startet, gibt es eine eindeutige Partie ρ. Wir schreiben kurz $w(\sigma, \tau, q_0) = w(\rho)$.

Definition 4.21 (Wert einer Strategie)**.** Gegeben sei ein Request-Response-Spiel. Dann lässt sich für eine Strategie σ von Spieler 0 folgender *Strategiewert* definieren:

$$w(\sigma, q_0) := \sup_{\tau \in S_1} w(\sigma, \tau, q_0)$$

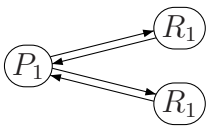
wobei S_1 die Menge aller Strategien von Spieler 1 ist. Wir schreiben $w(\sigma, q_0) = \infty$ für den Fall, dass der Wert nicht existiert.

Man beachte, dass die Existenz von $w(\sigma, q_0)$ impliziert, dass σ eine Gewinnstrategie ist. Umgekehrt zeigt Korollar 4.10, dass es Gewinnstrategien σ (mit unendlichem Speicher) geben kann, deren Wert $w(\sigma, q_0)$ nicht existiert. Ein zentrales Ergebnis dieses Abschnitts lautet, dass für eine "optimale" Strategie σ von Spieler 0 die Äquivalenz "$w(\sigma, q_0)$ existiert gdw. σ ist Gewinnstrategie von q_0 aus" gilt.

Definition 4.22 (Optimalität von RR-Strategien)**.** Für ein gegebenes RR-Spiel mit Startzustand q_0 ist eine Strategie σ_0 für Spieler 0 genau dann optimal, wenn gilt

$$w(\sigma_0, q_0) \leq w(\sigma, q_0) \text{ für alle Strategien } \sigma \text{ von Spieler 0.}$$

Bemerkung 4.23. Das Optimalitätskriterium liefert keine eindeutige optimale Strategie in einem RR-Spiel mit Startzustand q_0. Dies wird durch folgendes triviales Beispiel klar:



Jede mögliche (Gewinn-)Strategie für Spieler 0 hat für dieses RR-Spiel den Wert $\frac{1}{2}$. Somit gibt es keine eindeutige optimale Strategie für dieses RR-Spiel.

Aus der Büchi-Reduktion von Request-Response-Spielen aus Satz 3.8 folgt, dass die Strategiewerte optimaler Strategien beschränkt sind:

Korollar 4.24. *Für ein Request-Response-Spiel mit n Zuständen und r Bedingungen gilt für den Strategiewert einer optimalen Gewinnstrategie σ immer (für $q_0 \in W_0$):*

$$w(\sigma, q_0) \leq \frac{n \cdot r^2 + r}{2} \ .$$

Beweis. Sei ein beliebiges RR-Spiel mit n Zuständen und r Bedingungen gegeben. Der Strategiewert einer optimalen Gewinnstrategie ist immer kleiner oder gleich dem Strategiewert aller anderen möglichen Strategien. Dies gilt insbesondere auch für die Strategie σ, die sich mittels der Büchi-Reduktion ergibt. Die Reduktion auf ein Büchi-Spiel liefert, dass eine Anforderung in einer Partie, die gemäß der Strategie σ gespielt worden ist, höchstens $n \cdot r$ Schritte aktiv bleibt (siehe Korollar 3.10). Der höchste mögliche Strategiewert wird angenommen, wenn die Strategie σ aus der Büchi-Reduktion alle Bedingungen direkt, nachdem sie erfüllt worden sind, wieder aktiviert und die maximal mögliche Dauer von $n \cdot r$ Schritten aktiv lässt.

Eine einzelne Bedingung erzeugt, wenn sie $n \cdot r$ Schritte aktiv bleibt, den Beitrag $\frac{1}{n \cdot r} \cdot \sum_{i=0}^{n \cdot r} i = \frac{n \cdot r + 1}{2}$ zum Partiewert (nach Definition 4.18). Da sich die Schleife immer wiederholen soll, reicht es für die Bestimmung des Partiewertes aus, den Wert eines Schleifendurchlaufs zu bestimmen. Für r Bedingungen ergibt sich somit der Partiewert von $\frac{r \cdot (n \cdot r + 1)}{2} = \frac{n \cdot r^2 + r}{2}$. Da dies der maximal mögliche Partiewert einer gemäß σ gespielten Partie ist, ist somit der Strategiewert für σ kleiner oder gleich $\frac{n \cdot r^2 + r}{2}$. □

Somit ergibt Korollar 4.24 eine obere Schranke für den Wert optimaler Strategien in einem Request-Response-Spiel.

Die Definition 4.18 des Partiewertes $w(\rho)$ lässt sich auch auf ein Segment einer Partie ρ übertragen.

Definition 4.25. Gegeben sei ein RR-Spiel mit n Zuständen und r Bedingungen. Dann ist der *Segmentwert* $w(\rho, t_1, t_2)$ für eine Partie ρ auf dem Segment $\rho(t_1) \dots \rho(t_2)$ wie folgt definiert:

$$w(\rho, t_1, t_2) := \frac{1}{t_2 - t_1 + 1} \sum_{t=t_1}^{t_2} \sum_{i=1}^{r} WZ(\rho, t, i) \ .$$

Mit dieser Definition sind auch die Kosten für einen einzelnen Schritt in einer Partie definiert.

4.2 Optimale Strategien

In den folgenden beiden Abschnitten stellen wir eine Methode vor, in Request-Response-Spielen optimale Gewinnstrategien für Spieler 0 zu bestimmen. Im vorliegenden Abschnitt weisen wir zunächst nach, dass der optimale Strategiewert innerhalb einer globalen Beschränkung der Wartezeiten realisiert werden kann. Kern dieses Nachweises ist die Einsicht, dass es unmöglich ist, den Wert einer Partie dadurch zu verbessern, dass man für ausgewählte RR-Paare die Wartezeiten unbeschränkt wachsen lässt, um über die schnellere Erfüllung anderer RR-Paare einen Vorteil zu gewinnen. Als Folgerung dieses Beschränktheitssatzes erhalten wir, dass der optimale Wert von Gewinnstrategien durch eine Automatenstrategie realisierbar ist.

Im nachfolgenden Abschnitt nutzen wir den Beschränktheitssatz zur algorithmischen Bestimmung einer optimalen Strategie und ihres Wertes. Dazu nehmen wir eine Reduktion auf Mean-Payoff-Spiele [EM79] über endlichen Graphen vor, für die entsprechende Verfahren zur Verfügung stehen. Die Endlichkeit des Graphen wird durch den Beschränktheitssatz garantiert.

Wir beginnen nun mit dem Nachweis, dass es für jedes RR-Spiel, das von q_0 aus durch Spieler 0 gewonnen wird, optimale Strategien die Eigenschaft haben, dass alle RR-Paare beschränkt sind. Zunächst betrachten wir der Übersichtlichkeit wegen den Fall, dass genau ein RR-Paar unbeschränkt ist. Es wird gezeigt werden, dass es in einem solchen Fall eine Gewinnstrategie gleichen oder kleineren Werts konstruiert werden kann, bei der alle Paare beschränkt sind. Dieses wird im zweiten Schritt auf den Fall mehrerer unbeschränkter Paare übertragen.

Satz 4.26 (Beschränktheitssatz)**.** *Für jedes Request-Response-Spiel gilt, dass eine optimale Gewinnstrategie bezüglich aller RR-Paare beschränkt ist.*

Beweis. Sei σ eine optimale Gewinnstrategie für ein RR-Spiel mit n Zuständen und r Paaren. Nach Korollar 4.24 können wir annehmen, dass der Strategiewert $w(\sigma, q_0) \leq \frac{n \cdot r^2 + r}{2}$ ist. Wir müssen zeigen, dass man zu einer beschränkten Gewinnstrategie σ' übergehen kann mit $w(\sigma', q_0) \leq w(\sigma, q_0)$.

I.) Wir nehmen an, dass das erste RR-Paar (P_1, R_1) unbeschränkt ist. Für die Strategie σ gilt dann für $2 \leq i \leq r$, dass das i-te Paar durch s_i beschränkt ist (bezogen auf alle möglichen Partien, die von Spieler 0 gemäß der Strategie σ gespielt werden).

Ausgehend von der Strategie σ wollen wir eine neue Gewinnstrategie σ' definieren, bei der alle RR-Bedingungen beschränkt sein werden. Wir werden dann zeigen,

dass diese Strategie σ' auch optimal für das gegebene RR-Spiel ist. Zuvor muss erst noch der Begriff des *Spielbaums* zu einem *Partiepräfix* $\pi = \pi(0)\dots\pi(n)$ eingeführt werden. Der Spielbaum t_π^σ für ein um die Wartezeitvektoren erweitertes Partiepräfix π und eine Strategie σ von Spieler 0 in einem RR-Spiel hat als Wurzel den Knoten $\pi(n)$ und hat als Pfade davon ausgehend alle möglichen Partien, die gemäß der Strategie σ möglich sind. Für Knoten von Spieler 0 enthält der Spielbaum somit keine Wahlmöglichkeiten mehr.

Ist die erste RR-Bedingung nach dem Präfix π nicht aktiv, verhält sich die Strategie σ' für das Präfix π so wie die Strategie σ. Ansonsten wird der Spielbaum t_π^σ auf dem Spielgraphen G_r^+ für das Partiepräfix $\pi = \pi(0)\dots\pi(n)$ betrachtet, um die Gewinnstrategie σ' für das Präfix π definieren zu können; es werden also die Wartezeitvektoren adjungiert. Ein Pfad im Spielbaum t_π^σ wird nach dem ersten Besuch eines R_1-Knotens gekappt. Der resultierende Baum t_π^σ ist somit endlich, wenn $\pi(0) \in W_0$ ist, da σ eine Gewinnstrategie für Spieler 0 ist.

In einem Partiesegment, in dem die Wartezeit des ersten, unbeschränkten RR-Paares größer ist als $w = n \cdot \Pi_{i=2}^r s_i$, gibt es eine Wiederholung eines Zustands $q \in Q_0$, bei der sich nur die erste Komponente des Wartezeitvektors unterscheidet. Genau diese Knoten betrachten wir nun Baum t_π^σ, bei denen die Werte identisch zum Wert des Wurzelknotens mit Ausnahme des ersten Eintrags des Wartezeitvektors sind. Da diese Menge von Knoten endlich ist, gibt es Knoten mit maximalem ersten Eintrag des Wartezeitvektors – das können auch mehrere mit gleichem maximalen ersten Eintrag des Wartezeitvektors sein. Diese Situation ist in Abbildung 4.5 dargestellt. Dabei sind alle Knoten in dem Baum t_π^σ, die abgesehen von dem ersten Eintrag des Wartezeitvektors identisch zur Wurzel sind, durch das Symbol $\bullet$ dargestellt. Für einen solchen Knoten v ist der Spielbaum eingezeichnet, der diesen Knoten v als Wurzel hat.

Gibt es mehrere Knoten mit maximalem ersten Eintrag des Wartezeitvektors, wird ein beliebiger Knoten v aus dieser Menge ausgewählt, ansonsten sei v der eindeutige derartige Knoten. Es sei π' das bis zum Knoten v erweiterte Partiesegment π. Wenn $w(\pi', n, v) > w(\sigma, \pi(0))$ gilt, wählt die Strategie σ' den Nachfolger vom Knoten v im Baum t_π^σ aus und setzt den Wert des ersten Eintrags des Wartezeitvektors auf den Wert von v für die weiteren Partiepräfixberechnungen, ansonsten verhält sich die Strategie σ' für das Partiepräfix π genauso wie die Strategie σ.

Es reicht an dieser Stelle aus, die Wertewiederholung nur bei der Wurzel zu betrachten. Für den Fall, dass die Wertewiederholung für ein Präfix π erst nach der Wurzel auftritt, gibt es ein Präfix π' (mit π als Präfix von π'), so dass die

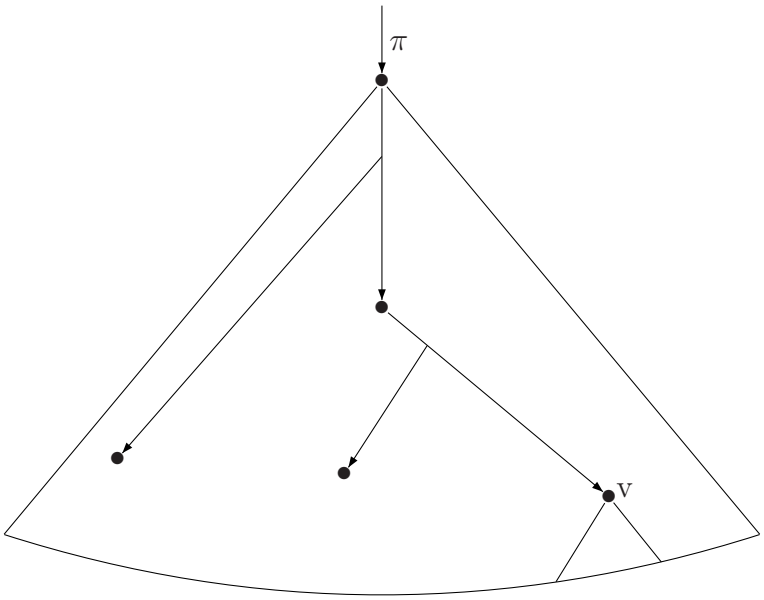


Abbildung 4.5: Spielbaum t^{σ}_{π}

Wertewiederholung für das Präfix π' bei der Wurzel auftritt.

Da dieser Vorgang für jedes Partiepräfix π gemacht werden muss, entspricht die Bestimmung der neuen Strategie σ' einer schrittweisen Kontraktion des Spielbaums $t^{\sigma}_{q_0}$. Ausgehend von der Wurzel werden Teilbäume, wie sie in Abbildung 4.5 dargestellt sind, betrachtet und gegebenenfalls, wie gerade beschrieben, kontrahiert.

Wird die Strategie für ein Partiepräfix geändert, garantiert die Änderung des ersten Eintrags des Wartezeitvektors, dass der Baum $t^{\sigma}_{\pi'}$ nicht verlassen wird, da bereits alle möglichen Transitionen von Spieler 1 berücksichtigt worden sind (und keine konkrete Strategie von Spieler 1) und Spieler 1 also keine Möglichkeit hat, den Besuch von R_1 beliebig lange hinauszuzögern. Somit ist klar, dass für alle Partien, die von Spieler 0 gemäß σ' gespielt werden, gilt, dass alle RR-Paare beschränkt sind. Für das erste Paar gilt dies durch die Konstruktion von σ', für die anderen RR-Paare bleibt die Eigenschaft erhalten. Da nur eine Strategieänderung erfolgt, wenn ein Partiesegment möglich ist, dessen Wert größer dem Strategiewert ist, ist sichergestellt, dass der Partiewert aller gemäß dieser Strategie möglichen Partien ausgehend von q_0 kleiner oder gleich $w(\sigma, q_0)$ ist. Somit gilt dann $w(\sigma', q_0) \leq w(\sigma, q_0)$ für $q_0 \in Q$.

II.) Nachdem für den Fall, dass genau ein RR-Paar unbeschränkt ist, gezeigt worden ist, dass eine optimale Strategie die Beschränktheit aller Paare garantiert, wird

dies jetzt für den allgemeinen Fall gezeigt. Dazu werden die gerade gewonnenen Erkenntnisse weiter abstrahiert. Der allgemeine Fall hat zur Folge, dass die unbeschränkten Wartezeiten als Tupel vorliegen, für die wir eine Halbordnung an Stelle einer totalen Ordnung verwenden. Wir definieren also eine Halbordnung auf den Wartezeitvektoren. Für zwei Wartezeitvektoren $w = (w_1, \ldots, w_r)$ und $w' = (w'_1, \ldots, w'_r)$ gilt $w \leq w'$ genau dann, wenn $w_i \leq w'_i$ für alle $1 \leq i \leq r$ gilt.

Weiterhin wird Dicksons Lemma [Dic13] als Eigenschaft einer unendlichen Folge von Wartezeitvektoren benötigt:

Lemma 4.27 (Dicksons Lemma [Dic13])**.** *Für eine unendliche Folge von Vektoren $M_1, M_2, M_3, \ldots \in \mathbb{N}^k$ existieren Indizes i und j mit $i < j$, so dass $M_i \leq M_j$ gilt.* □

Wir zeigen nun Satz 4.26 für den allgemeinen Fall, in Analogie zum betrachteten Fall eines einzelnen unbeschränkten RR-Paares. Sei σ eine optimale Gewinnstrategie, die in mehreren RR-Paaren unbeschränkt ist.

Analog zu dem Fall genau eines unbeschränkten RR-Paares kann auch jetzt eine Strategie σ' definiert werden, die in allen Paaren beschränkt ist und für die $w(\sigma', q_0) \leq w(\sigma, q_0)$ gilt. Es wird nacheinander für jede RR-Bedingung eine neue Strategie erzeugt, in der die bisher betrachteten und das aktuell betrachtete RR-Paar beschränkt sein werden. Somit wird immer ein RR-Paar mehr beschränkt, bis schließlich bei der Strategie $\sigma^{\{1,\ldots,r\}}$ alle Paare gemäß dieser Strategie beschränkt sind. Dabei wird im Exponenten der Strategie angegeben, welche RR-Bedingungen bereits betrachtet worden sind und somit bezogen auf die Strategie beschränkt sind. Wir setzen dazu $\sigma^{\emptyset} = \sigma$.

Ausgehend von der Strategie σ^B wird für ein $i \notin B$ die Strategie $\sigma^{B \cup \{i\}}$ für ein um die Wartezeitvektoren aus Definition 4.6 erweitertes Partiepräfix π definiert. Ist die RR-Bedingung (P_i, R_i) nach dem Präfix π nicht aktiv, ist die Strategie $\sigma^{B \cup \{i\}}$ für das Präfix π gleich dem Wert der Strategie σ^B. Sonst wird analog zum Fall eines einzigen unbeschränkten RR-Paares der Spielbaum $t_\pi^{\sigma^B}$ auf dem Spielgraphen G_r^+ für das Partiepräfix $\pi = \pi(0) \ldots \pi(m)$ betrachtet, dessen Pfade in diesem Fall nach dem Erreichen von R_i gekappt werden (anstelle von R_1). Dieser Baum ist genau dann endlich, wenn $\pi(0) \in W_0$ gilt (denn σ ist eine Gewinnstrategie und garantiert die Erfüllung von R_i). Existiert ein in der Baumhalbordnung maximaler Zustand (der Zustand ist entweder eindeutig oder es wird ein beliebiger in Frage kommender ausgewählt) in dem Baum $t_\pi^{\sigma^B}$, der die folgenen fünf Bedingungen erfüllt, wählt die Strategie $\sigma^{B \cup \{i\}}$ den Nachfolger dieses Knotens als Nachfolger für das Partiepräfix π aus:

1. Das RR-Paar (P_i, R_i) ist zwischen $t = m$ und t' konstant aktiv;
2. $\rho^B(t) = \rho^B(t') \in Q_0$;
3. $\rho^B(t+1) \neq \rho^B(t'+1)$ (unterschiedliche Folgezustände);
4. Für die beiden Wartezeitvektoren $w = (w_1, \ldots, w_r)$ und $w' = (w'_1, \ldots, w'_r)$ zu den Zeitpunkten t bzw. t' gilt $w \leq w'$. Dicksons Lemma [Dic13] garantiert, dass die Folge der Wartezeitvektoren, für die $w_i \not\leq w_j$ für alle $i < j$ gilt, endlich ist;
5. $w(\rho^B, t, t') > w(\rho)$.

Existiert ein solcher in dem Baum $t_\pi^{\sigma^B}$ nicht, verhält sich die Strategie $\sigma^{B\cup\{i\}}$ genau wie die Strategie σ^B für dieses Partiepräfix π.

Somit ist analog zum ersten Fall klar, dass für alle Partien, die von Spieler 0 gemäß σ' gespielt worden sind, gilt, dass alle RR-Paare in $B \cup \{i\}$ beschränkt sind. Die RR-Bedingungen aus B bleiben weiterhin beschränkt. Die Argumentation ist in diesem Fall analog zu dem Fall genau eines unbeschränkten RR-Paares. Dicksons Lemma garantiert, dass die Folge der Wartezeitvektoren, für die $w_i \not\leq w_j$ für alle $i < j$ gilt, endlich ist. Somit gibt es zusammen mit der fünften Bedingung eine Schranke, wie lange eine Bedingung (P_i, R_i) höchstens aktiv sein kann, bis die Wartezeit bedingt durch eine Strategieänderung gekürzt werden kann. Auch durch ein unendlich häufiges Entfernen von Partiesegmenten kann trotzdem nicht der Fall eintreten, dass von einem Zeitpunkt ab jede Erfüllung einer Bedingung aus B entfernt wird und somit schließlich doch eine Bedingung nicht erfüllt wird. Dies liegt darin begründet, dass der Strategiewert $w(\sigma^{B\cup\{i\}})$ durch $\frac{n\cdot r^2+r}{2}$ beschränkt ist, da σ optimal ist, und es somit auch immer wieder Partiesegmente geben muss, für die der Segmentwert echt kleiner dem Strategiewert ist.

Eine Strategieänderung erfolgt nur, wenn ein Partiesegment möglich ist, dessen Wert größer als der Strategiewert $w(\sigma, q_0)$ ist, und zusätzlich die Bedingung an die Wartezeitvektoren erfüllt ist, die garantiert, dass eine Bedingung nicht länger aktiv sein kann, als es ohne Strategieänderung möglich wäre. Somit ist sichergestellt, dass der Partiewert aller gemäß dieser Strategie möglichen Partien ausgehend von q_0 kleiner oder gleich $w(\sigma, q_0)$ ist. Somit gilt dann $w(\sigma^{B\cup\{i\}}, q_0) \leq w(\sigma, q_0)$ für $q_0 \in Q$.

Die Strategie $\sigma^{\{1,\ldots,r\}}$ ist dann schließlich in allen Paaren beschränkt.

Korollar 4.28. *Eine optimale Gewinnstrategie in einem RR-Spiel ist eine Automatenstrategie.*

Beweis. Da die Optimalität einer Gewinnstrategie nur von den Wartezeiten der einzelnen Bedingungen abhängt und für diese im Satz 4.26 gezeigt worden ist, dass bei einer optimalen Strategie die Wartezeit für alle RR-Bedingungen beschränkt ist, folgt die Behauptung aus der Tatsache, dass der resultierende Spielgraph endlich ist. □

Bemerkung 4.29. Für die Partiewertdefinition $t(\rho)$, bei der ein linearer Einfluss der Wartezeiten vorliegt, lässt sich der Satz 4.26 übertragen, da im Beweis von Satz 4.26 an keiner Stelle der quadratische Einfluss einer aktiven Bedingung ausgenutzt worden ist. Dies bedeutet, dass auch für den Partiewert $t(\rho)$ Automatengewinnstrategien ausreichen. Für die Partiewertdefinition $a(\rho)$ lässt sich der Satz 4.26 nicht übertragen, wie das Beispiel in Abbildung 4.4 gezeigt hat.

Wir definieren die Schar $f_n : \mathbb{N} \to \mathbb{N}$ von Funktionen f_n induktiv wie folgt:

$$f_n(1) = n$$
$$f_n(i+1) = f_n(i) + n \cdot f_n(i)^i$$

Lemma 4.30 (Max open request)**.** *In einem gegebenen Request-Response-Spiel mit n Zuständen und r Paaren kann unter Anwendung einer optimalen Strategie für Spieler 0 jede Anforderung maximal $w_{max}(n,r) = \frac{n \cdot r^2 + r}{2} + f_n(r)$ Züge offen bleiben.*

Beweis. Aus der fünften Bedingung des Zusammenfassens aus dem Beweis von Satz 4.26 folgt, dass eine RR-Bedingung mindestens so lange aktiv sein muss, bis $w(\rho, t, t') > w(\rho)$ gilt. Dieses lässt sich nach oben durch $w(\rho) \leq \frac{n \cdot r^2 + r}{2}$ abschätzen, da die Partie ρ gemäß der optimalen Strategie σ gespielt wird. Erst danach kann die Wartezeit zusammengekürzt werden.
Dazu wird im weiteren Verlauf die vierte Bedingung dafür genauer untersucht. Eine Komponente des Wartezeitvektors kann in einem Spielzug entweder

- um genau eins erhöht werden, wenn die Bedingung aktiv ist,
- auf Null zurückgesetzt werden, wenn die Bedingung gerade erfüllt wurde, oder
- den Wert Null behalten, wenn die Bedingung nicht aktiviert wurde.

Betrachten wir verschiedene Werte für r (hier der Anzahl von RR-Paaren), um herzuleiten, wie viele Schritte maximal möglich sind, bis die fünfte Bedingung des Zusammenkürzens erfüllt ist.

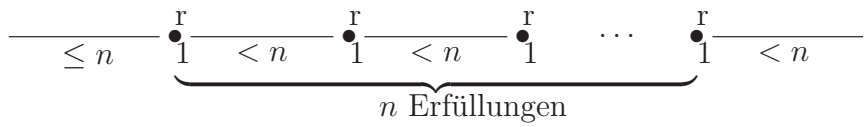
$r = 1$ Das einzige betrachtete Paar muss ein ganzes Partiesegment lang aktiv sein, damit es gekürzt werden kann. Nach $n+1$ Schritten gibt es eine Zustandswiederholung und die Bedingung ist erfüllt. Folglich sind $f_n(1) = n$ Schritte möglich, bis die Bedingung erfüllt sein muss.

$r = 2$

- Mindestens ein Paar muss die gesamte Zeit über aktiv sein.
- Das andere Paar muss immer wieder erfüllt werden und somit die Komponente des Wartezeitvektors auf Null zurückgesetzt werden, ansonsten wäre die Bedingung nach $n + 1$ Schritten erfüllt.
- Dieses Paar kann maximal $n - 1$ Schritte aktiv bleiben (mit dem Wert 0 ergeben sich insgesamt n verschiedene mögliche Werte).

Es gibt also nur n verschiedene Möglichkeiten für das Produkt aus Zustand und Wartezeitvektor, bei dem immer dieselbe Komponente den Wert Null hat.

Somit ist die Situation wie folgt:



Die Abbildung zeigt das (zweite) RR-Paar, welches immer wieder erfüllt wird. In der Abbildung sind die Punkte der Erfüllung durch den Buchstaben r markiert.

Es folgt $f_n(2) = n \cdot n + n = n \cdot (n + 1)$ als Abschätzung für die Schrittanzahl.

$r = 3$

- Es gilt: Mindestens ein Paar muss die gesamte Zeit lang aktiv sein.
- Mindestens ein Paar muss innerhalb von $n+1$ Schritten erfüllt werden, sonst gibt es eine Zustandswiederholung.
- RR-Paare können höchstens $n \cdot (n + 1) - 1$ Schritte aktiv bleiben (mit dem Wert 0 ergeben sich wieder $n \cdot (n + 1)$ verschiedene mögliche Werte).

Also gilt: Nach $n \cdot (n+1)$ Schritten sind alle RR-Paare mindestens einmal erfüllt worden. Folglich kann danach jedes Paar höchstens $n \cdot (n + 1) - 1$ Schritte aktiv sein. Es ergibt sich $f_n(3) = f_n(2)+n\cdot f_n(2)^2$ als Abschätzung für die Schrittanzahl.

$r = i + 1$

- Es gilt: Mindestens ein Paar muss die gesamte Zeit über aktiv sein.
- Mindestens ein Paar muss innerhalb von $n+1$ Schritten erfüllt werden, sonst gibt es eine Zustandswiederholung.
- RR-Paare können höchstens $f_n(i)-1$ Schritte aktiv bleiben (mit dem Wert 0 ergeben sich wieder $f_n(i)$ verschiedene mögliche Werte).

Also werden nach $f_n(i)$ Schritten alle RR-Paare mindestens einmal erfüllt. Danach kann jedes Paar nur höchstens $f_n(i) - 1$ Schritte aktiv sein. Es ergibt sich $f_n(i + 1) = f_n(i) + n \cdot f_n(i)^i$ als Abschätzung für die Schrittanzahl.

Fasst man die gerade gewonnenen Ergebnisse zusammen, bekommt man als eine obere Schranke für die maximale Wartezeit $w_{max}(n, r) = \frac{n \cdot r^2 + r}{2} + f_n(r)$. □

Bemerkung 4.31. $w_{max}(n,r)$ lässt sich durch $\mathcal{O}\left(n^{\mathcal{O}((r-1)!)}\right)$ nach oben abschätzen.

Beweis. Es gilt zunächst die Funktion $f_n(i+1) = f_n(i)+n\cdot f_n(i)^i$ geeignet abzuschätzen.

$$f_n(i) = f_n(i-1) + n \cdot f_n(i-1)^{i-1}$$

Wir führen eine Hilfsfunktion F_n ein und schätzen diese ab.

$$\begin{aligned}
F_n(i) &= \log f_n(i) \\
&= \log(f_n(i-1)\cdot(1+n\cdot f_n(i-1)^{i-2})) && \text{(Def. } f_n(i)\text{)} \\
&= \log f_n(i-1) + \log(1+n\cdot f_n(i-1)^{i-2}) \\
&= F_n(i-1) + \log\left(n\cdot f_n(i-1)^{i-2}\cdot\left(1+\frac{1}{n\cdot f_n(i-1)^{i-2}}\right)\right) && \text{(Def. } F_n(i)\text{)} \\
&\leq F_n(i-1) + \log n + (i-2)\cdot F_n(i-1) + \mathcal{O}\left(\frac{1}{n\cdot f_n(i-1)^{i-2}}\right) \\
&\leq F_n(i-1) + (i-2)\cdot(\log n + F_n(i-1)) + \mathcal{O}\left(\frac{1}{n\cdot f_n(i-1)^{i-2}}\right) \\
&= (i-1)\cdot F_n(i-1) + (i-2)\cdot\log n + \mathcal{O}\left(\frac{1}{n\cdot f_n(i-1)^{i-2}}\right) \\
&\leq (i-1)\cdot F_n(i-1) + (i-1)\cdot\log n \\
\frac{F_n(i)}{(i-1)!} &\leq \frac{F_n(i-1)}{(i-2)!} + \frac{\log n}{(i-2)!}
\end{aligned}$$

Die Substitution $G_n(i) = \frac{F_n(i)}{(i-1)!}$ ergibt dann:

$$\begin{aligned}
G_n(i) &\leq G_n(i-1) + \frac{\log n}{(n-2)!} \\
&= \frac{\log n}{(i-2)!} + \frac{\log n}{(i-3)!} + \frac{\log n}{(i-4)!} + \ldots + \frac{\log n}{0!} + G_n(1) \\
&= \log n\cdot\left(1+1+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{(i-2)!}\right) \\
&\leq \log n\cdot e
\end{aligned}$$

Es folgt

$$\begin{aligned}
F_n(i) &\leq (i-1)!\cdot\log n\cdot e \\
f_n(i) &\leq 2^{(i-1)!\cdot\log n\cdot e} \\
&= n^{(i-1)!\cdot e}
\end{aligned}$$

Somit ergibt sich insgesamt: $w_{max}(n,r) = \frac{n\cdot r^2+r}{2} + f_n(r) \in \mathcal{O}\left(n^{(r-1)!\cdot e}\right) = \mathcal{O}\left(n^{\mathcal{O}((r-1)!)}\right)$, da $\frac{n\cdot r^2+r}{2} \leq n^{(r-1)!\cdot e}$. □

4.3 Berechnung optimaler Gewinnstrategien

In diesem Abschnitt wird ein Verfahren zur Berechnung einer optimalen Strategie (gemäß Definition 4.22) vorgestellt. Zugleich wird damit die Existenz einer optimalen Gewinnstrategie gezeigt. Wir führen eine Reduktion auf Mean-Payoff-Spiele durch und nutzen aus, dass Mean-Payoff-Spiele nach Zwick und Paterson [ZP96] optimale, positionale Gewinnstrategien haben. Deshalb werden im folgenden Abschnitt kurz die Mean-Payoff-Spiele eingeführt.

4.3.1 Mean-Payoff-Spiele

Definition 4.32 (Mean-Payoff-Spiel, [EM79])**.** Gegeben sei ein Spielgraph $G = (Q, E)$ und eine Gewichtsfunktion $wf : E \to \{-W, \ldots, 0, \ldots, W\}$, die jeder Kante ein ganzzahliges Gewicht zwischen $-W$ und W zuordnet. Spieler 0 versucht $\liminf_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} wf(e_i)$ zu maximieren, während Spieler 1 versucht, $\limsup_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} wf(e_i)$ zu minimieren.

Ehrenfeucht und Mycielski [EM79] haben die lokale Determiniertheit von Mean-Payoff-Spielen gezeigt, d. h. jedes Mean-Payoff-Spiel hat einen Wert v, so dass Spieler 0 eine Strategie hat, die sicherstellt, dass $\liminf_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} wf(e_i) \geq v$ gilt, während Spieler 1 eine Strategie hat, die sicherstellt, dass $\limsup_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} wf(e_i) \leq v$ gilt.

Satz 4.33 ([ZP96])**.** *In einem Mean-Payoff-Spiel $G = (Q, E)$ mit einer Gewichtsfunktion $wf : E \to \mathbb{R}$, bei dem für einen Spieler gilt, dass er bei allen seinen Knoten nur einen Nachfolger hat, lassen sich die Werte für alle Knoten und eine optimale positionale Strategie des Gegners in Zeit $\mathcal{O}(|Q| \cdot |E|)$ bestimmen.*

Für Mean-Payoff-Spiele ist in der Arbeit von Gurvich, Karzanov und Khachiyan [GKK88] ein Algorithmus mit exponentiellem Zeitbedarf für die Berechnung optimaler Strategien angegeben. In der Arbeit von Zwick und Paterson [ZP96] ist zusätzlich ein Algorithmus angegeben, der polynomiell in der Größe des Graphen und pseudo-polynomiell in den Gewichten ist.

Satz 4.34 ([ZP96])**.** *Optimale positionale Strategien für beide Spieler in einem Mean-Payoff-Spiel $G = (Q, E)$ mit einer Gewichtsfunktion $wf : E \to \{-W, \ldots, 0, \ldots, W\}$ können mit Zeitaufwand $\mathcal{O}(|Q|^4 \cdot |E| \cdot \log(|E|/|Q|) \cdot W)$ berechnet werden.*

4.3.2 Reduktion auf Mean-Payoff-Spiele

Nachdem mit Lemma 4.30 und der Einführung der Mean-Payoff-Spiele die Grundlagen gelegt worden sind, soll nun eine Reduktion von Request-Response-Spielen auf Mean-Payoff-Spiele vorgestellt werden, um optimale Strategien in RR-Spielen zu berechnen.

Satz 4.35. *Für ein Request-Response-Spiel $G = (Q, E)$ mit $Q = Q_0 \dot\cup Q_1$, $|Q| = n$ und r RR-Paaren ist eine optimale Strategie für Spieler 0 nach Definition 4.22 berechenbar.*

Idee. Lemma 4.30 liefert eine Schranke für die Wartezeitvektoren, die wir im Weiteren als w_{max} bezeichnen. Diese ermöglicht es, als Spielgraphen den endlichen Graphen $G^+_{r|w_{max}}$ aus Definition 4.7, bei dem die Knoten von Spieler 0 und 1 vertauscht sind, für das Mean-Payoff-Spiel zu benutzen.

Als Gewicht einer Kante im konstruierten Mean-Payoff-Spiel wird dann die Summe der aktuellen Wartezeiten der aktiven Anforderungen gesetzt. Die Strategie von Spieler 1 im Mean-Payoff-Spiel, der diese Summe minimiert, ist optimal im Request-Response-Spiel für Spieler 0.

Formal wird die Kantengewichtsfunktion $wf : E' \to \{0, \ldots, r^2 \cdot w_{max} + r\}$ wie folgt definiert:

$$wf((q, \begin{pmatrix} w_1 \\ \vdots \\ w_r \end{pmatrix}, (q', \begin{pmatrix} w'_1 \\ \vdots \\ w'_r \end{pmatrix})) = \sum_{k=1}^{r} (\mathit{open}(w'_k) \cdot w'_k + \mathit{failed}(w'_k) \cdot (r \cdot w_{max} + 1)) \text{ mit}$$

$$\mathit{open}(i) = \begin{cases} 1, & \text{falls } 0 \leq i \leq w_{max} \\ 0, & \text{sonst} \end{cases} \text{ und } \mathit{failed}(i) = \begin{cases} 1, & \text{falls } i = w_{max} + 1 \\ 0, & \text{sonst} \end{cases}$$

Eine Vergröberung für den Fall, dass Spieler 1 gewinnt, kann so definiert werden: $wf' : E' \to \{0, \ldots, r \cdot w_{max} + r\}$ mit

$$wf'((q, \begin{pmatrix} w_1 \\ \vdots \\ w_r \end{pmatrix}), (q', \begin{pmatrix} w'_1 \\ \vdots \\ w'_r \end{pmatrix})) = \begin{cases} \sum_{k=1}^{r} \mathit{open}(w'_k) \cdot w'_k, \text{ falls } \sum_{k=1}^{r} \mathit{failed}(w'_k) = 0 \\ r \cdot w_{max} + \sum_{k=1}^{r} \mathit{failed}(w'_k), \text{ sonst} \end{cases}$$

Die Vergröberung besteht darin, dass, wenn Spieler 1 gewinnt, nur noch in der Anzahl der unerfüllten Bedingungen differenziert wird. Wie lange Bedingungen, die erfüllt werden, offen bleiben, spielt dabei keine Rolle. Der Vorteil der Vergröberung besteht darin, dass die Gewichtsfunktion einen kleineren Wertebereich hat und somit die Algorithmen für die Mean-Payoff-Spiele effizienter sind, da der Wertebereich der Gewichtsfunktion die Laufzeit zur Berechnung optimaler Strategien beeinflusst (siehe Satz 4.34). Deshalb wird im weiteren Verlauf auch nur noch die Vergröberung benutzt.

Nachdem die Definition von $w(\sigma, q_0)$ nur eine qualitative Aussage über das Gewinnen von Spieler 0 macht, erlauben die beiden Definitionen der Kantengewichtsfunktion *wf*

bzw. wf' auch eine quantitative Aussage über das Gewinnen von Spieler 1 (also das Verlieren von Spieler 0).

Bevor wir zu dem Korrektheitsbeweis kommen, soll zunächst demonstriert werden, dass bei dieser Konstruktion immer noch die Gewinnbereiche für beide Spieler bestimmt werden können. Unsere Konstruktion liefert ein spezielles Payoff-Spiel nach Zielonka [Zie05]. Unter einem Payoff-Spiel versteht Zielonka ein Spiel, bei dem jede Aktion einen Preis kostet. Formal ist die Menge der möglichen Aktionen als $Q \times \mathcal{R} \times Q$ definiert, wobei die Menge $\mathcal{R}$ die sofortigen Kosten darstellt. Die Payoff-Funktion u über $\mathcal{R}$ ist eine Funktion, die die Kosten in reelle Zahlen überführt, also $u : \mathcal{R} \to \mathbb{R}$. Ändert man dieses Payoff-Mapping in der Reduktion des Request-Response-Spiels auf ein Mean-Payoff-Spiel, können direkt die Gewinnbereiche des RR-Spiels angegeben werden. Dies erfolgt analog dazu, wie Zielonka es für Paritätsspiele in [Zie05] vorgemacht hat.

Korollar 4.36. *Für ein RR-Spiel auf dem erweiterten Spielgraphen $G^+_{r|w_{max}}$ lassen sich die Gewinnbereiche für beide Spieler mittels der Lösung eines Mean-Payoff-Spiels berechnen.*

Beweis. Wir betrachten die gerade vorgestellte Reduktion eines RR-Spiels auf ein Mean-Payoff-Spiel. Der erweiterte Spielgraph ist dann durch $G^+_{r|w_{max}}$ gegeben. Der Algorithmus von Zwick und Paterson [ZP96] bestimmt nicht nur die optimalen Strategien für beide Spieler in einem Mean-Payoff-Spiel, sondern auch für alle Zustände $q \in Q$ des Spiels die dazugehörigen Werte $v(q)$. Betrachten wir folgende Payoff-Funktion u:

$$u(q) := \begin{cases} 1, & \text{wenn } v(q) > r \cdot w_{max} \\ 0, & \text{sonst} \end{cases}$$

Ist der Wert für einen Zustand größer als $r \cdot w_{max}$, so gehört der Zustand zu dem Gewinnbereich von Spieler 1, ansonsten zu dem von Spieler 0. Dies wird durch das Payoff-Mapping u ausgedrückt. Ist der Wert größer als $r \cdot w_{max}$, bedeutet dies, dass Spieler 0 es nicht geschafft hat, eine Bedingung innerhalb der maximalen Wartezeit bei einer optimalen Strategie zu erfüllen. Daraus folgt weiterhin, dass Spieler 0 es nicht erzwingen kann, die Bedingung zu erfüllen. Somit gehört der Zustand zum Gewinnbereich von Spieler 1. Wenn ein Zustand zum Gewinnbereich von Spieler 1 gehört, folgt daraus, dass der Wert größer als $r \cdot w_{max}$ ist. Da das Spiel nach Martin [Mar75] determiniert ist, sind somit die Gewinnbereiche für beide Spieler bestimmt. □

Behauptung: Die gerade vorgestellte Reduktion eines RR-Spiels auf ein Mean-Payoff-Spiel erzeugt eine optimale Strategie für Spieler 0 im RR-Spiel.

Beweis. Die Reduktion auf ein Mean-Payoff-Spiel liefert die optimalen Gewinnstrategien σ_0' bzw. σ_1' für Spieler 0 bzw. 1 im Mean-Payoff-Spiel und die daraus resultierenden Strategien σ_0 und σ_1 im RR-Spiel. Wir wollen von der Optimalität von σ_0' im Mean-Payoff-Spiel auf die Optimalität von σ_0 im ursprünglichen RR-Spiel schließen.

Annahme: Die Gewinnstrategie σ_0 ist nicht optimal für Spieler 0 im ursprünglichen RR-Spiel.

Daraus folgt, dass es eine Gewinnstrategie τ_0 mit $w(\tau_0, q_0) < w(\sigma_0, q_0)$ für einen Zustand q_0 aus dem Gewinnbereich von Spieler 0 im RR-Spiel gibt. Der Wert einer gemäß den Strategien σ_0' und σ_1' gespielten Strategie vom Zustand q_0 im Mean-Payoff-Spiel ist definiert als $\min \sup_{n\to\infty} \sum_{i=1}^{n} w(e_i)$. Dabei lässt sich die eben vorgestellte Gewichtsfunktion *wf* wie folgt vereinfachen, da in einem Zustand q_0 des Gewinnbereichs von Spieler 0 gestartet worden ist und beide Spieler gemäß ihrer Gewinnstrategien spielen: $wf((p,q)) = \sum_{k=1}^{r} open(i_k) \cdot i_k$ mit der Definition von *open* wie zuvor. Dies entspricht genau der Definition von *WZ*, da im Zähler c_i genau gerade gespeichert wird, wie lange die Bedingung i schon aktiv ist. Dies wiederum bedeutet, dass die Strategie σ_0 bereits optimal für Spieler 0 ist. Somit gilt $w(\sigma_0, q_0) = w(\tau_0, q_0)$. Widerspruch. □

Für den Fall, dass q nicht im Gewinnbereich von Spieler 0 liegt, ist die Frage nach der Optimalität nicht eindeutig. Die Definitionen *wf* und *wf′* für die Kantenbewertungsfunktion zeigen zwei unterschiedliche Definitionsmöglichkeiten auf.

Es fehlt noch eine Laufzeitabschätzung für die vorgestellte Reduktion.

Korollar 4.37 (Laufzeit)**.** *Eine optimale Strategie für Spieler 0 in einem RR-Spiel* $G = (Q, E)$ *mit* r *RR-Paaren lässt sich in Zeit* $\mathcal{O}\left(r \cdot |Q|^{\mathcal{O}(r!)} \cdot |E| \cdot \log\left(\frac{|E|}{|Q|}\right)\right)$ *berechnen.*

Beweis. Aus Satz 4.34 folgt, dass sich das Mean-Payoff-Spiel des reduzierten RR-Spiels in Zeit $\mathcal{O}\left(|Q'|^4 \cdot |E'| \cdot \log\left(\frac{|E'|}{|Q'|}\right) \cdot W\right)$ lösen lässt. Setzt man nun in diese Gleichung die Werte ein, die sich bei der Reduktion des RR-Spiels auf das Mean-Payoff-Spiel ergeben, ergibt sich mit $W = r \cdot w_{max} + r = r \cdot (w_{max} + 1)$, $|Q'| = |Q| \cdot (w_{max})^r$ und $|E'| = |E| \cdot (w_{max})^r$:

$$\begin{aligned} &\mathcal{O}\left(|Q'|^4 \cdot |E'| \cdot \log\left(\frac{|E'|}{|Q'|}\right) \cdot W\right) \\ &= \mathcal{O}\left(|Q|^4 \cdot |E| \cdot (w_{max})^{5 \cdot r} \cdot \log\left(\frac{|E|}{|Q|}\right) \cdot r \cdot (w_{max} + 1)\right) \end{aligned}$$

und mit $w_{max} \in \mathcal{O}\left(|Q|^{\mathcal{O}((r-1)!)}\right)$ aus Bemerkung 4.31

$$
\begin{aligned}
&= \mathcal{O}\left(|Q|^{5\cdot r\cdot\mathcal{O}((r-1)!)+4} \cdot |E| \cdot \log\left(\frac{|E|}{|Q|}\right) \cdot r \cdot \left(|Q|^{\mathcal{O}((r-1)!)} + 1\right)\right) \\
&= \mathcal{O}\left(|Q|^{\mathcal{O}(r!)} \cdot |E| \cdot \log\left(\frac{|E|}{|Q|}\right) \cdot r \cdot |Q|^{\mathcal{O}((r-1)!)}\right) \\
&= \mathcal{O}\left(r \cdot |Q|^{\mathcal{O}(r!)} \cdot |E| \cdot \log\left(\frac{|E|}{|Q|}\right)\right) \quad . \qquad \square
\end{aligned}
$$

4.3.2.1 Berechnung des Strategiewerts

Nachdem gerade gezeigt wurde, wie eine optimale Strategie für Spieler 0 in einem RR-Spiel berechnet werden kann, stellt sich die Frage, wie man den Strategiewert einer beliebigen Automatengewinnstrategie σ von Spieler 0 berechnen kann. Dazu wird der Spielgraph um den Strategieautomaten für die Strategie σ erweitert und anschließend der Strategiewert durch eine Reduktion auf ein Mean-Payoff-Spiel berechnet.

Zu einem gegebenen RR-Spiel $G = (Q, E)$ mit r RR-Bedingungen, $|Q| = n$ und einer Automatengewinnstrategie σ von Spieler 0 kann der Strategiewert $w(\sigma, q_0)$ wie folgt bestimmt werden: Der Spielgraph G' entsteht aus G durch Erweiterung um den Strategieautomaten von σ, wie bereits angedeutet. Der Strategieautomat von σ habe m Zustände. Bis auf die Strategiekanten werden sämtliche Transitionen ausgehend von Knoten von Spieler 0 in G' gelöscht. Somit liegt der Fall vor, dass Spieler 0 immer genau eine ausgehende Kante hat, d. h. die Voraussetzungen von Satz 4.33 sind erfüllt. Lemma 4.9 besagt, dass Spieler 0 die Erfüllung einer Bedingung nur innerhalb von $n \cdot m$ Zügen erzwingen kann. Da σ eine Gewinnstrategie von Spieler 0 ist, reicht es aus, den Spielgraphen $G'^{+}_{r|n\cdot m}$ aus Definition 4.7 zu betrachten, bei dem alle Bedingungen durch $n \cdot m$ beschränkt sind. Danach wird wieder, wie bei der Berechnung einer optimalen Strategie, ein Mean-Payoff-Spiel konstruiert. Im Mean-Payoff-Spiel auf dem Spielgraphen $G'^{+}_{r|n\cdot m}$, bei dem die Knoten von Spieler 0 und 1 vertauscht sind, liefert der Wert des transformierten Knotens q_0 dann genau den Strategiewert $w(\sigma, q_0)$ des RR-Spiel – dies ist genau der Mean-Payoff-Wert des Knotens $(q_0, m_0, 0/1, \ldots, 0/1)$, d. h. die Komponenten sind: der Startzustand im ursprünglichen Graphen, Startzustand des Strategieautomaten und die Wartezeiten t_i, welche den Wert 0 annehmen, wenn der Zustand $q_0 \notin P_i$ ist, ansonsten den Wert 1. Die Vertauschung der Knoten von Spieler 0 und 1 ist notwendig, da Spieler 0 im RR-Spiel einen möglichst niedrigen Wert erreichen möchte, aber Spieler 0 im Mean-Payoff-Spiel den Wert maximiert.

Es ist leicht zu sehen, dass die Konstruktion korrekt ist, da gerade genau die Berechnung von $w(\sigma, q_0)$ nachgeahmt wird. Der Zeitaufwand für die Berechnung von $w(\sigma, q_0)$ kann mit Hilfe von Satz 4.33 durch $\mathcal{O}(|Q'| \cdot |E'|) \leq \mathcal{O}(|Q'|^3) = \mathcal{O}((n \cdot m \cdot (n \cdot m)^r)^3) = \mathcal{O}((n \cdot m)^{3\cdot(r+1)})$ abgeschätzt werden.

4.4 Anwendung auf andere Spiele

In diesem Abschnitt wollen wir andere verbreitete Gewinnbedingungen betrachten und diskutieren, in welchem Umfang sich für diese Gewinnbedingungen die Frage nach der Optimalität von Gewinnstrategien beantworten lässt. Dazu wird jeweils kurz erläutert, was wir unter einer optimalen Strategie verstehen wollen und ob die in der Literatur bekannten Algorithmen bereits optimale Gewinnstrategien berechnen.

Beginnen wir dazu bei den Erreichbarkeitsspielen, bei denen Spieler 0 eine gegebene Menge $F \subseteq Q$ von Knoten des Spielgraphen erreichen muss, um eine Partie zu gewinnen. Das Qualitätsmaß ist in diesem Fall natürlicherweise die Anzahl der Züge, die zum Erreichen der gegebenen Menge F benötigt werden. Die durch die Attraktor-Konstruktion gewonnene Gewinnstrategie ist (über endlichen Graphen) in dieser Hinsicht optimal. Für das komplementäre Sicherheitsspiel, bei dem eine gegebene Menge $F \subseteq Q$ nicht verlassen werden soll, ist es schwierig, natürliche Maße für "Optimalität" einzuführen.

Bei den schwachen Paritätsspielen, die Spieler 0 gewinnt, wenn die größte besuchte Priorität gerade ist, wird die Güte der Gewinnstrategie durch zwei Komponenten beeinflusst: einerseits die größte besuchte Priorität, und andererseits die Anzahl der Züge, die nötig ist, um genau diese Priorität zu besuchen. Auch hier gilt wieder, dass das Verfahren aus der Literatur in diesem Sinn optimal ist (geschachtelte Attraktorkombination). Normale Paritätsspiele, die Spieler 0 gewinnt, wenn die größte unendlich oft gesehene Priorität gerade ist, verhalten sich ähnlich. Bei diesen wird die Güte der Gewinnstrategie einerseits durch die größte unendlich oft besuchte Priorität beeinflusst und andererseits von der durchschnittlichen Dauer, bis diese höchste Priorität besucht wird.

Betrachtet man für verallgemeinerte Büchi-Spiele (mit Gewinnkomponente $\{F_1, \ldots, F_n\}$) die Frage nach einer optimalen Strategie, spielt die Zeit zwischen zwei Besuchen einer F_i-Menge die entscheidende Rolle. Analog entspricht dies bei den RR-Spielen der Zeit zwischen Aktivierung und Erfüllung einer Bedingung. Verallgemeinerte Büchi-Spiele sind ein Spezialfall von Request-Response-Spielen, denn eine F_i-Menge des verallgemeinerten Büchi-Spiels kann durch $P_i = Q \setminus F_i$ und $R_i = F_i$ in eine RR-Bedingung überführt werden. Deshalb liefert die Reduktion auf Mean-Payoff-Spiele aus dem Beweis von Satz 4.35 auch für verallgemeinerte Büchi-Spiele optimale Strategien für beide Spieler. Offen bleibt an dieser Stelle, ob optimale Strategien für verallgemeinerte Büchi-Spiele einfacher berechnet werden können. Dafür würde sprechen, dass die Lösung derartiger Spiele auch einfacher ist als die von RR-Spielen.

Florian Horn hat in einer bisher unveröffentlichten Arbeit optimale Strategien für "finitary" Streett-Spiele [CH06] definiert. In der Berechnung von optimalen Strategien für

diese Spiele wird die in dieser Arbeit vorgestellte Berechnung von optimalen Gewinnstrategien in Request-Response-Spielen benutzt.

Bei anderen Spieltypen, wie zum Beispiel Spielen mit der Staiger-Wagner-Bedingung oder der Muller-Bedingung, ist bereits die Definition eines Maßes für Optimalität nicht offensichtlich.

Kapitel 5

Implementierung

In den vorangehenden Kapiteln haben wir Algorithmen zur Lösung von Spielen auf abstrakter Ebene diskutiert. Im vorliegenden Kapitel beschreiben wir die Implementierung der vorgestellten Algorithmen. Entwickelt wurde hierzu die Experimentierplattform GASt (Games, Automata & Strategies), die sich aus drei Komponenten zusammensetzt:

1. Determinisierung von Büchi-Automaten (Determinisierung ist eine Grundvoraussetzung zur Spezifikation unendlicher Spiele)
2. Algorithmen zum Lösen von unendlichen Zwei-Personen-Spielen
3. Strategiesynthese für LSC-Spezifikationen (als Anwendung der unendlichen Zwei-Personen-Spiele)

Möchte man sich selber einen Eindruck von der entstandenen Plattform verschaffen, ist diese unter `http://www-i7.informatik.rwth-aachen.de/research/GASt/` zu finden. Ziel war es, eine einheitliche Oberfläche für alle drei Komponenten zu schaffen. Für die erste Komponente existierte eine C++-Implementierung ("OmegaDet"), die im Rahmen der Diplomarbeit [Alt05] von Christoph Schulte Althoff entstanden ist. Für die letzte Komponente ist das Programm REMoRDS von Yves Bontemps (Namur, Belgien) integriert worden, welches im Rahmen seiner Dissertation [Bon05] entstanden ist. Dieses nutzt zum Lösen der einfachen Streett-Spiele die hier vorgestellten und in der zweiten Komponente implementierten Algorithmen.

Im folgenden Abschnitt werden die obigen Komponenten genauer vorgestellt. Des weiteren findet sich dort auch eine Übersicht, wie die einzelnen Algorithmen zusammenhängen. Der darauf folgende Abschnitt betrachtet dann softwaretechnische Aspekte wie den Entwicklungsprozess und die Architektur der Implementierung.

5.1 Bestandteile der Implementierung

Die entstandene Experimentierplattform GASt ist eine Java-Anwendung, die ein installiertes Java Runtime Environment[1] der Version 5 benötigt, welches frei von den Seiten von Sun Microsystems zu beziehen ist. Als C-Anwendungen benutzen wir das Programm `OmegaDet` zur Determinisierung von Büchi-Automaten und die Implementierung `c-parity-obda` des Vöge/Jurdziński-Algorithmus zum Lösen von Paritätsspielen. Ferner wird zur Implementierung symbolischer Algorithmen eine BDD-Bibliothek benötigt.

Im gleichen Verzeichnis wie das eigentliche Programm `GASt.jar` müssen auch die beiden C-Anwendungen `OmegaDet` bzw. `c-parity-obda` liegen, die fertig kompiliert sowohl für Linux als auch für Windows mitgeliefert werden. Die BDD-Bibliothek muss im Falle der Windows-Version ebenfalls im selben Verzeichnis wie das Programm selber liegen (Dateiname `bdd.dll`). Unter Linux muss die BDD-Bibliothek (`libbdd.so`) in einem Verzeichnis liegen, welches in der Shell-Variablen `LD_LIBRARY_PATH` enthalten ist. Dies gilt in der Regel für das Verzeichnis `/usr/lib`. Das Programm kann unter Windows mit einem Doppelklick gestartet werden, wenn die beschriebenen Voraussetzungen geschaffen worden sind. Unter Linux (oder auch anderen Betriebssystemen) kann das Programm durch folgenden Aufruf gestartet werden:

```
java -jar GASt.jar
```

Damit Java mehr Heap-Speicher nutzen kann, um zum Beispiel größere Beispiele enumerativ berechnen zu können, kann man obigen Aufruf noch um einen weiteren Parameter erweitern. Um z.B. 512 MB Speicher nutzen zu können, ergibt sich folgender Aufruf:

```
java -Xmx512m -jar GASt.jar
```

In den folgenden Unterabschnitten wird nun die Bedienung der drei Komponenten genauer vorgestellt. Im nächsten Abschnitt wird auf die softwaretechnischen Aspekte der Implementierung eingegangen.

[1] zu finden unter `http://www.java.sun.com`

5.1.1 Determinisierung von Büchi-Automaten

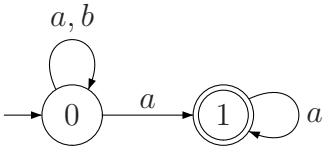
Die erste Komponente der Plattform GASt dient zur Determinisierung von Büchi-Automaten und basiert auf der C++-Implementierung OmegaDet, welche im Rahmen der Diplomarbeit von Christoph Schulte Althoff entstanden ist. Die Komponente bietet vier verschiedene Algorithmen zur Determinisierung von Büchi-Automaten:

- die Safra-Konstruktion
- die Muller-Schupp-Konstruktion
- eine verbesserte Muller-Schupp-Konstruktion (siehe Abschnitt 2.2)
- die Hayashi-Miyano-Konstruktion (anwendbar auf co-Büchi-Automaten)

Die vierte Option ist implementiert worden, da sich herausgestellt hat, dass viele Spezifikationen als co-Büchi-Automaten vorliegen. Dies ist zum Beispiel der Fall, wenn eine akzeptierende Schleife nur aus einem einzelnen Zustand besteht. In diesem Fall trifft die Büchi-Bedingung genau dann zu, wenn von einem beliebigen Zeitpunkt an nur noch Endzustände auftreten (co-Büchi-Akzeptierbedingung). Auch Automaten, die bei gleicher Endzustandsmenge als Büchi- bzw. co-Büchi-Automat erstmal nicht sprachäquivalent sind, können zuweilen so verändert werden, dass durch das Hinzufügen von zusätzlichen Endzuständen ein sprachäquivalenter co-Büchi-Automat erzeugt wird. Zustände, von denen nur Endzustände erreichbar sind, können ebenfalls als Endzustände deklariert werden, da sich die akzeptierte Sprache des Büchi-Automaten dadurch nicht ändert. Somit besteht ein sinnvoller Vorverarbeitungsschritt darin, auch solche Zustände als Endzustände zu deklarieren, von denen nur Endzustände erreichbar sind, um den Büchi-Automaten als sprachäquivalenten co-Büchi-Automaten auffassen zu können.

Die Hayashi-Miyano-Konstruktion ist eine Variante der Potenzmengenkonstruktion. Sie erzeugt nur $2^{\mathcal{O}(n)}$ Zustände (im Gegensatz zu $2^{O(n \log n)}$ bei den anderen Konstruktionen) und ist somit für große Automaten, die sich als co-Büchi-Automat repräsentieren lassen, gegenüber den anderen Konstruktionen zu bevorzugen. Eine Implementierung dieser Konstruktion existiert auch im LASH-Paket in Lüttich (siehe [MH84, BJW05, LAS]).

Die Eingabe eines nicht-deterministischen Büchi-Automaten erfolgt über eine Textdatei oder direkt über die Programmoberfläche. Das Format der Textdatei soll jetzt an einem Beispiel verdeutlicht werden. Betrachten wir dazu den folgenden Büchi-Automaten $\mathcal{A}_1$, der alle ω-Wörter akzeptiert, die nur endlich oft ein b enthalten.



Wir repräsentieren diesen Automaten als Textdatei wie folgt:

```
2
ab
1
0 a 0
0 b 0
0 a 1
1 a 1
```

Die erste Zeile enthält die Zustandsanzahl n des Büchi-Automatens, wobei die Zustandsmenge $\{0, ..., n-1\}$ ist. Der Anfangszustand ist immer der Zustand 0. Die zweite Zeile beschreibt das verwendete Alphabet Σ als eine Folge von ASCII-Zeichen. Jedes Zeichen repräsentiert genau einen Buchstaben. Danach werden in der dritten Zeile die Endzustände aufgelistet, getrennt durch Leerzeichen. Zum Schluss werden die Transitionen aufgelistet.

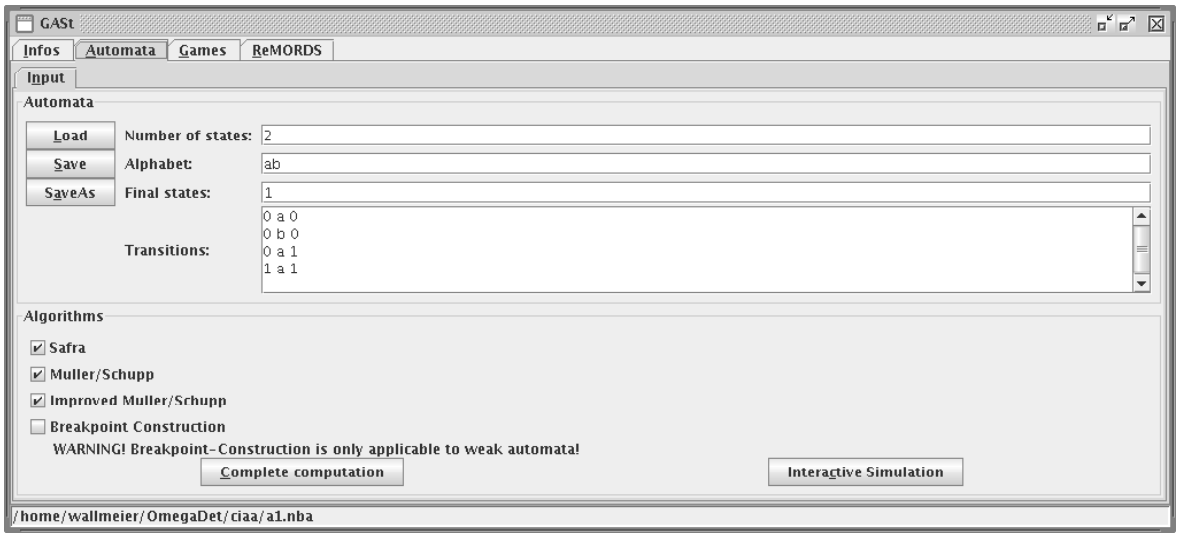


Abbildung 5.1: GUI Automaten: Eingabe

Wie Abbildung 5.1 zeigt, kann der Benutzer natürlich den Automaten auch direkt in GASt eingeben. Dabei gelten die selben Regeln wie bei der Textdatei. Ebenfalls steht hier die Möglichkeit zum Laden und Speichern von Automaten zur Verfügung. Nachdem ein Automat gespeichert oder geladen worden ist, wird der dazugehörige Dateiname in der Statusleiste angezeigt.

Der Benutzer kann dann die gewünschten Algorithmen zur Determinisierung des Automaten im Kasten "Algorithms" auswählen. Anzumerken bleibt noch, dass für die Hayashi-Miyano-Konstruktion keine Überprüfung eingebaut ist, ob der gegebene Automat wirklich die co-Büchi-Bedingung erfüllt. Diese Überprüfung bleibt dem Benutzer im Vorfeld überlassen.

Danach kann der Benutzer entscheiden, ob er die Automaten vollständig gemäß der ausgewählten Algorithmen berechnet haben möchte, oder ob er interaktiv einen Lauf simulieren möchte. Letztere Variante bietet sich gerade für sehr große Automaten an, wenn die komplette Berechnung zu lange dauern würde bzw. wenn man das Verhalten der ausgewählten Algorithmen miteinander vergleichen möchte.

5.1.1.1 Vollständige Berechnung

Nachdem der Benutzer den Button “Complete computation” angeklickt hat, beginnt das Programm, den Automaten gemäß der ausgewählten Algorithmen zu determinisieren. Dazu wechselt es auf die Anzeige in Abbildung 5.2. Die Anzeige wird fortlaufend immer dann aktualisiert, sobald weitere 200 Zustände des Zielautomatens berechnet sind bzw. ein Algorithmus das Ende einer Berechnung erreicht hat. Ist die Konstruktion eines Automatens abgeschlossen, kann man sich die textuelle Darstellung des Automaten über den Button “Show” anzeigen lassen.

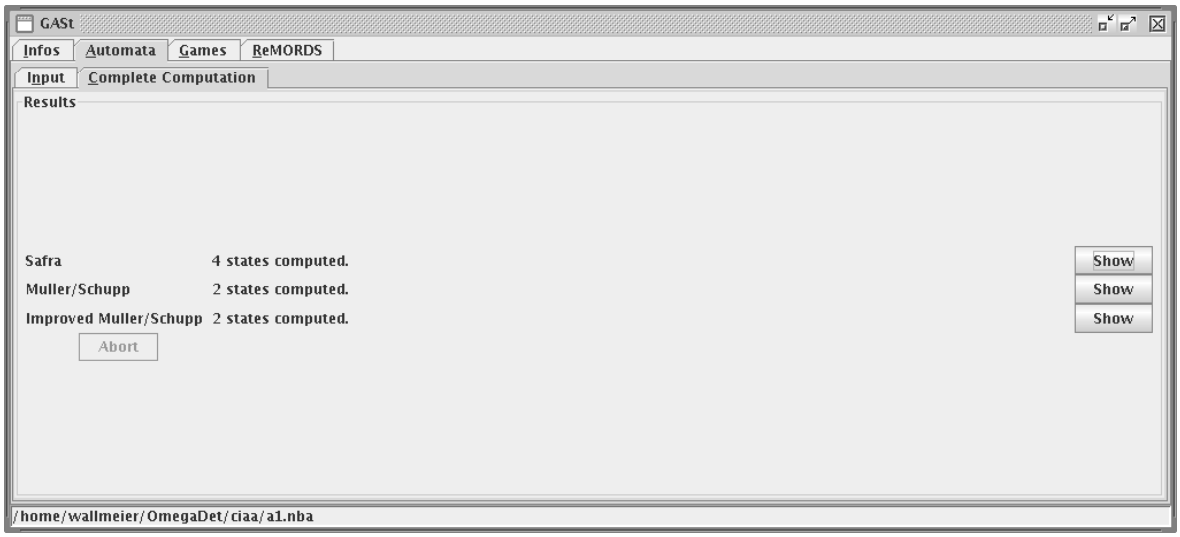


Abbildung 5.2: GUI Automaten: Komplette Berechnung

Die textuelle Darstellung des resultierenden Automaten ist in vier Teile unterteilt:

- die Anzahl der Zustände,
- die Liste der Safra- bzw. Muller-Schupp-Bäume, die jeweils als si bzw. ki (für $i = 0, 1, \ldots$) bezeichnet werden, zusammen mit einem Wort über das der Zustand erreicht werden kann (und zwar das erste in der kanonischen Reihenfolge der Wörter) und einer textuellen Darstellung des Baums (siehe Erklärung weiter unten),
- die Transitionstabelle, die auf die Namen si bzw. ki zurückgreift,
- die Liste und Anzahl der akzeptierenden Rabin-Paare.

Für die Darstellung der Bäume ist eine textuelle Repräsentation gewählt worden, bei der die Söhne eines Knotens unter dem Vaterknoten eingerückt werden und mit einem

Markierungssymbol +-> beginnen. Somit werden Brüderknoten mit gleicher Einrücktiefe dargestellt. Die Farben “rot”, “gelb” und “grün” werden durch die Symbole -, 0, + repräsentiert, die einem Knoten im Muller-Schupp-Baum nachgestellt werden. In einem Safra-Baum wird das Ausrufungszeichen “!” benutzt, wenn es sich um einen grün gefärbten Knoten handelt.

Als Beispiel für die Ausgabe soll der gemäß Safra-Konstruktion aus obigem Beispiel erzeugte Automat dienen.

```
Deterministic Rabin automaton
according to Safra:

4 States:
s0:
    [1|0]

s1: a
    [1|0,1]

s2: aa
    [1|0,1]
     +-> [2|1]

s3: aaa
    [1|0,1]
     +-> [2|1]!
```

```
Transition table:
     a   b
s0  s1  s0
s1  s2  s0
s2  s3  s0
s3  s3  s0

Acceptance pairs:
for vertex 2 (sizes 2,1):
({s0,s1},{s3})

Overall: 1 pair with non-empty
acceptance set
```

5.1.1.2 Interaktive Simulation

Wählt der Benutzer statt der kompletten Berechnung der Automaten die interaktive Simulation aus (Button “Interactive Simulation”), gelangt er zu der in Abbildung 5.3 dargestellten Oberfläche. In dieser Ansicht kann der Benutzer interaktiv den Lauf in dem von ihm ausgewählten deterministischen Automaten verfolgen. Für jeden der gewählten Algorithmen wird der jeweils aktuelle Zustand des entsprechenden Automaten angezeigt. Dazu drückt er jeweils den Buchstaben auf der Tastatur, für den der Lauf fortgesetzt werden soll. Natürlich sollte der gewählte Buchstabe auch im Eingabealphabet des Büchi-Automaten vorkommen. Damit man immer weiß, wie das Eingabealphabet lautet, wird es oben im Fenster nochmals angezeigt. Dort wird ebenfalls angegeben, wie das bisherige Eingabewort aussieht. Der Benutzer hat nicht nur die Möglichkeit, weitere Buchstaben an das Eingabewort anzuhängen, sondern kann mit-

tels der Backspace-Taste auch wieder zurückgehen – z.B., wenn man feststellt, dass man in einen uninteressanten Lauf abgebogen ist.

Abbildung 5.3: GUI Automaten: Interaktive Simulation

Mittels des "Reset"-Buttons kann auch wieder zum leeren Eingabewort ε zurückgekehrt werden. Möchte man die berechneten, partiellen[2] Automaten speichern, um später darauf zurückgreifen zu können, kann man dies mit dem "Save"-Button veranlassen. Bestandteile des neuen Dateinamen sind der Dateiname des Büchi-Automaten, der ausgewählte Algorithmus und das Eingabewort. Mit dem "Stop"-Button beendet man die interaktive Simulation und gelangt wieder zu der Eingabe des Büchi-Automaten zurück.

5.1.2 Strategiesynthese für unendliche Spiele

Mittels der Strategiesynthese für unendliche Zwei-Personen-Spiele hat der Benutzer die Möglichkeit, reaktive Systeme mit den unterschiedlichsten Anforderungen zu spezifizieren und anschließend automatisch erzeugen bzw. überprüfen zu lassen. Damit ist es z.B. möglich, eine Aufzugsteuerung (siehe Beispiel 3.16) einfach und schnell zu spezifizieren und sich anschließend den Controller aus der Spezifikation erzeugen zu lassen (soweit die Größe der Zustandsraums dies zulässt). Dazu sind in der Komponente der Strategiesynthese für unendliche Spiele alle wichtigen Algorithmen aus der Literatur und selbstentwickelte sowohl enumerativ als auch symbolisch umgesetzt, sofern dies möglich ist (siehe dazu auch die Tabellen 5.1 und 5.2). Darunter befinden sich auch die

[2] im Sinne von "noch nicht vollständig berechnet"

bekannten Reduktionen von Streett-Spielen auf Paritätsspiele mittels IAR[3], sowie von Muller-Spielen auf Paritätsspiele mittels LAR[4].

Über eine parametrisierbare, symbolische Eingabesprache werden sowohl der Spielgraph als auch die Gewinnbedingung spezifiziert (für weitere Details siehe Unterabschnitt 5.1.2.2). Das Programm berechnet dann die Gewinnbereiche und -strategien für beide Spieler und gibt das Ergebnis auf der "Result"-Seite der Benutzeroberfläche aus. Über die angezeigten Zwischenergebnisse besteht die Möglichkeit, die Algorithmen zum Lösen der Spiele besser nachzuvollziehen.

5.1.2.1 Unterstützte Spieltypen

Tabelle 5.1 gibt einen Überblick, welche Art Spiele bzw. reaktive Systeme der Benutzer über die Plattform GASt spezifizieren kann. Man sieht, dass alle gängigen Spieltypen zumindest enumerativ umgesetzt sind, auch die beiden aus der Literatur bekannten Reduktionen auf Paritätsspiele mittels IAR und LAR.

Spieltyp	**Enum.**	**Sym.**	**Lösungsmethode**
Erreichbarkeit	√	√	Attraktor-Berechnung
Sicherheit	√	√	Attraktor-Berechnung
schwache Parität	√	√	mehrere Attraktor-Berechnungen
Staiger-Wagner	√	√	Reduktion auf schwache Parität
Request-Response	√	√	Reduktion auf Büchi
Büchi	√	√	Berechnung von *Attr+* und *Recur*
verallgemeinert Büchi	√	√	Reduktion auf Büchi
Parität	√	√	verschiedene Verfahren
Muller	√	—	Reduktion auf Parität (LAR)
einfach Streett	√	√	vorgestellter Algorithmus
Streett	√	—	Reduktion auf Parität (IAR)
Mean-Payoff	√	—	Algorithmus nach Zwick/Paterson

Tabelle 5.1: Unterstützte Spieltypen in der Plattform GASt

Für alle diese Spiele bestimmt GASt sowohl die Gewinnbereiche für beide Spieler als auch deren mögliche Gewinnstrategien. Einzige Ausnahme ist hier der Algorithmus von Jurdziński zum Lösen von Paritätsspielen, der nur eine Gewinnstrategie für Spieler 0 liefert.

In Tabelle 5.2 sind die Spieltypen aufgeführt, für die die Plattform GASt optimale Gewinnstrategien berechnen kann (siehe dazu auch Kapitel 4).

[3] index appearance record
[4] latest appearance record

Spieltyp	Enum.	Sym.	Lösungsmethode
Mean-Payoff	√	—	Algorithmus nach Zwick/Paterson
Request-Response	√	—	Reduktion auf Mean-Payoff

Tabelle 5.2: Spieltypen, für die eine Berechnung optimaler Gewinnstrategien implementiert ist

Anzumerken bleibt, dass die Reduktion von RR-Spielen auf Mean-Payoff-Spiele zur Bestimmung optimaler Strategien nur der Vollständigkeit halber implementiert worden ist. Die Komplexität dieser Reduktion (siehe auch Korollar 4.37) lässt einen praktisch sinnvollen Einsatz nicht zu.

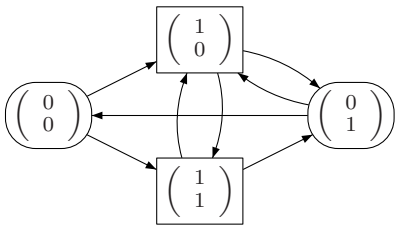
5.1.2.2 Eingabesprache

Die Eingabesprache, mit der Spielgraphen und Gewinnbedingungen spezifiziert werden können, ist von zentraler Bedeutung, da sie die Interaktion zwischen dem Programm und dem Benutzer maßgeblich beeinflusst. Auf der einen Seite soll sie möglichst einfach zu erlernen und die resultierenden Ausdrücke gut verständlich sein, andererseits soll sie möglichst ausdrucksstark sein, um kurze, prägnante Spezifikationen zu ermöglichen.

Auch wenn die Plattform GASt unendliche Zwei-Personen-Spiele sowohl enumerativ als auch symbolisch lösen kann, erfolgt die Spezifikation der Spiele immer symbolisch über boolesche Formeln. Für größere Beispiele wie der Aufzugsteuerung aus Beispiel 3.16 führt kein Weg an einer symbolischen Eingabe vorbei, da man den Transitionsgraph im Fall von fünf Stockwerken mit 10.000 Zuständen und 70.000 Transitionen nicht explizit angeben kann. Soll ein Spiel enumerativ gelöst werden, wird die symbolische Eingabe in BDDs übersetzt (wie beim symbolischen Verfahren auch) und dann über die erfüllenden Belegungen die enumerative Darstellung erzeugt. Die Eingabesprache, welche für die Implementierung benutzt wurde, ist durch die Grammatik in Tabelle 5.3 spezifiziert.

Beispiel 5.1. Betrachten wir einige Beispiele, um die Eingabesprache zu illustrieren.

(a) Gegeben sei folgender einfacher Spielgraph (die Knoten sind mit den Variablenbelegungen $x[0]$ und $x[1]$ beschriftet, wobei $x[0]$ oben steht):



		Produktionen	**Bedeutung**
Term	→	Quantor Var OpVarCond Term |	Quantor Variablenindices
		TermElem BinaryOp Term |	Binäre Operationen
		TermElem	
Var	→	$[a-z]^+$	
Quantor	→	“E“ |	Existenzieller Quantor
		“A“	Universeller Quantor
OpVarCond	→	“{“ VarCond “}“ | ε	VarCond oder leer
BinaryOp	→	“&“ | “|“ | “<>“ |	And, Or, XOr
		“!&“ | “!|“ |	NAnd, NOr
		“->“ | “=“ |	Implikation, Biimplikation
		“<-“ |	Umgekehrte Implikation
		“-“ | “<“	Differenz, Kleiner
TermElem	→	“(“Term“)“ |	geklammerter Term
		“!(“ Term “)“ |	Negation
		“x[“ VarIndex “]“ |	Variable
		“!x[“ VarIndex “]“ |	negierte Variable
		“x’[“ VarIndex “]“ |	Variable für Trans.ziel
		“!x’[“ VarIndex “]“ |	neg. Variable für Trans.ziel
		”bin(”VarIndex ”,”VarIndex ”,”VarIndex ”)	[a]
		”bin’(”VarIndex ”,”VarIndex ”,”VarIndex ”)	[b]
		“true“ | “false“	Boolesche Werte
VarCond	→	VarCond “&“ VarCond |	And
		VarCond “|“ VarCond |	Or
		VarCondElem	
VarCondElem	→	“(“ VarCond “)“ |	geklammerte Bedingung
		Var “=“ VarIndex |	gleich
		Var “!=“ VarIndex |	ungleich
		Var “<“ VarIndex |	kleiner
		Var “>“ VarIndex	größer
VarIndex	→	VarIndex “*“ VarIndex |	Multiplikation
		VarIndex “+“ VarIndex |	Addition
		VarIndex “-“ VarIndex |	Subtraktion
		”ln”VarIndexElem |	Logarithmus (Basis 2)
		VarIndexElem	
VarIndexElem	→	“(“ VarIndex “)“ |	geklammerter Ausdruck
		Var | $[0-9]^+$	Variable oder Wert

[a] bin(3,0,2) erzeugt den BDD mit der binären Kodierung vom Wert 3 beginnend mit der Variablen 0, wobei insgesamt 2 Variablen benutzt werden

[b] Analog zu bin(...) für das Ziel einer Transition

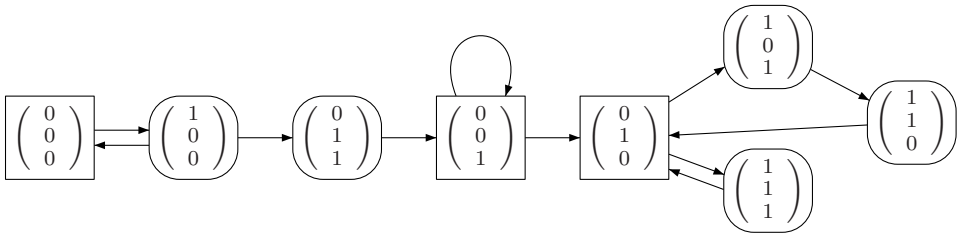
Tabelle 5.3: Grammatik der Eingabesprache

Der abgebildete Spielgraph lässt sich in der Eingabesprache notieren als:

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

Dabei beschreiben die Formeln φ_i die Knoten von Spieler i und τ die Transitionen zwischen diesen Knoten.

(b) Der Spielgraph aus Beispiel 3.20



lässt sich schreiben als:

$$\begin{aligned}
\varphi_0 &:= x[0] \mid (x[1] \;\&\; x[2]) \\
\varphi_1 &:= !x[0] \;\&\; !(x[1] \;\&\; x[2])
\end{aligned}$$

$$\begin{aligned}
\tau \;:=\; & (x[0] \;\&\; !x[1] \;\&\; !x[2] \;\&\; !x'[0] \;\&\; (x'[1] = x'[2])) \mid \\
& (!x[0] \;\&\; !x[1] \;\&\; x[2] \;\&\; !x'[0] \;\&\; (x'[1] = !x'[2])) \mid \\
& (!x[0] \;\&\; x[1] \;\&\; !x[2] \;\&\; x'[0] \;\&\; x'[2]) \mid \\
& (!x[0] \;\&\; !x[1] \;\&\; !x[2] \;\&\; x'[0] \;\&\; !x'[1] \;\&\; !x'[2]) \mid \\
& (!x[0] \;\&\; x[1] \;\&\; x[2] \;\&\; !x'[0] \;\&\; !x'[1] \;\&\; x'[2]) \mid \\
& (x[0] \;\&\; !x[1] \;\&\; x[2] \;\&\; x'[0] \;\&\; x'[1] \;\&\; !x'[2]) \mid \\
& (x[0] \;\&\; x[1] \;\&\; !x[2] \;\&\; !x'[0] \;\&\; x'[1] \;\&\; !x'[2]) \mid \\
& (x[0] \;\&\; x[1] \;\&\; x[2] \;\&\; !x'[0] \;\&\; x'[1] \;\&\; !x'[2]) \mid
\end{aligned}$$

5.1.2.3 Die Programmoberfläche

Dieser Abschnitt beschreibt kurz die Programmoberfläche für die Strategiesynthese von unendlichen Zwei-Personen-Spielen. Sie ist so ausgerichtet, dass sie möglichst intuitiv bedienbar ist.

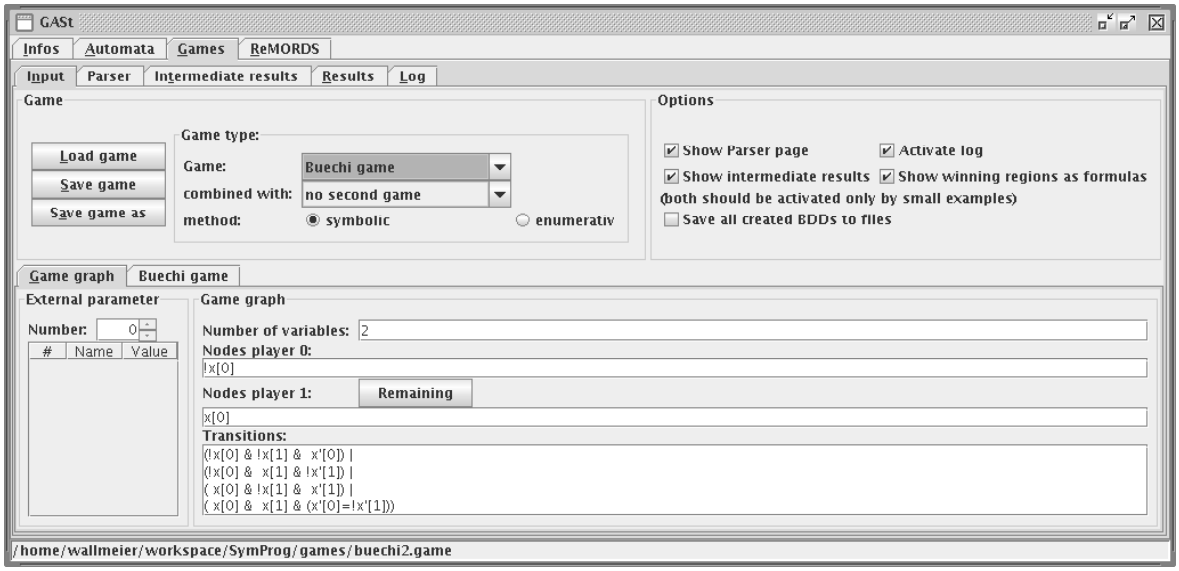


Abbildung 5.4: GUI unendliche Spiele: Eingabe-Seite

Die Eingabeseite in Abbildung 5.4 ist die zentrale Stelle zur Definition eines Spiels. Es wird mittels der im letzten Unterabschnitt vorgestellten Eingabesprache definiert – der Screenshot zeigt das Beispiel 5.1 (a). Der Benutzer kann auswählen, welchen Typ von Spiel er spezifizieren möchte. Besitzt der ausgesuchte Spieltyp mehrere Lösungsverfahren bzw. Optimierungen für diese Verfahren, kann der Benutzer diese neben der Box für den Spieltyp auswählen – diese Auswahloptionen werden nur angezeigt, wenn es auch wirklich eine Wahlmöglichkeit gibt (z.B. bei enumerativ zu lösenden Paritätsspielen). Wenn man auf dem Gewinnbereich von Spieler 0 im ersten Spiel ein weiteres Spiel definieren möchte, kann man dieses über die Auswahlbox, die mit “combined with” beschriftet ist, ausdrücken.

Der Spielgraph wird im Karteireiter mit der Aufschrift “Game graph” definiert. Der Kasten “External parameter” dient dazu, Variablen zu definieren, die in den Formeln für die Spieler-Knoten, der Transitionsformel, der Gewinnbedingung oder der Variablenanzahl benutzt werden können. Somit lässt sich beispielsweise eine Aufzugsteuerung unabhängig von der Etagenanzahl spezifizieren. Die Etagenanzahl würde als externe Variable definiert. Die Gewinnbedingung wird auf dem Karteireiter des Spiels (z.B. beschriftet mit “Request response game”) eingeben.

Im Kasten “Options” kann man sich einige nützliche Debug-Informationen aktivieren, wie das Ergebnis des Parsers, der die eingegebenen Formeln in BDDs übersetzt, berechnete Zwischenergebnisse oder auch die Ausgabe einer einfachen Log-Datei. Ein anderer interessanter Punkt ist die Möglichkeit, alle BDDs (Knoten der Spieler, Transitionen, Gewinnbereiche und Strategien) in Dateien abspeichern zu lassen, um sie mit anderen Programmen weiter benutzen zu können. Dazu wird das BDD-Paket BuDDy [BuD] benutzt. Die Dokumentation dieses Paketes enthält weitere Informationen zum Dateiformat dieser BDDs.

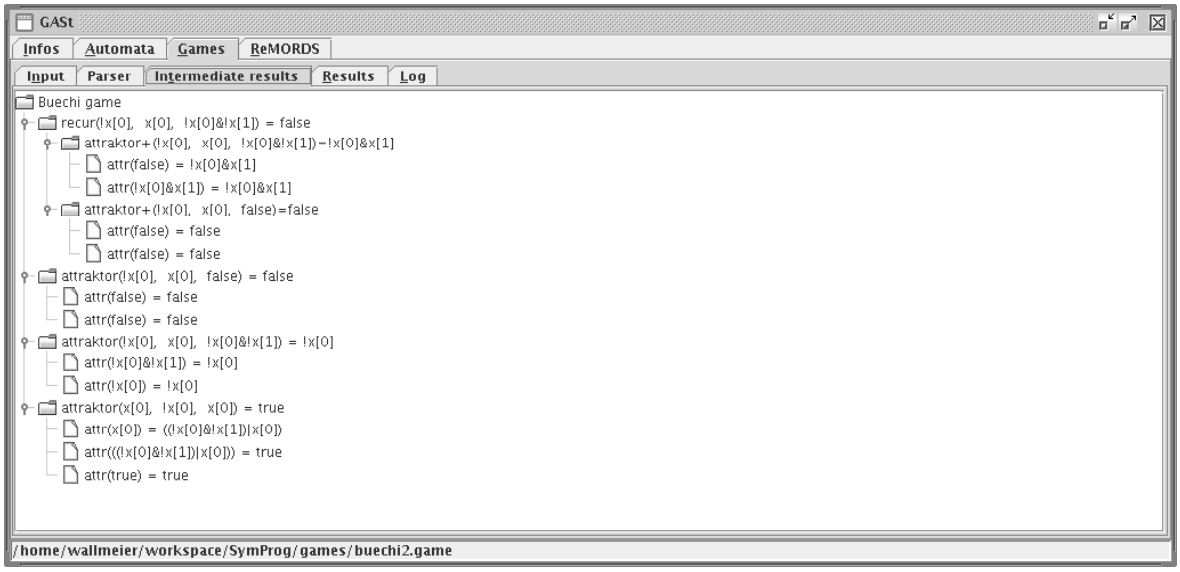


Abbildung 5.5: GUI unendliche Spiele: Zwischenergebnis-Seite

Auf der Zwischenergebnis-Seite in Abbildung 5.5 kann man anhand von Zwischenergebnissen gut erkennen, wie das Lösen eines konkreten Spiels funktioniert und so einen Algorithmus nachvollziehen. Im konkreten Fall sieht man, wie ein Büchi-Spiel mittels *Recur*, $Attraktor^+$ und *Attraktor* gelöst worden ist. Durch die Baumstruktur ist auch zu erkennen, wie die einzelnen Schritte zusammenhängen.

Auf der Ergebnis-Seite in Abbildung 5.6 wird die Lösung des Spiels ausgegeben. Hierfür werden neben den booleschen Formeln für die Gewinnbereiche der Spieler 0 und 1 auch die Mächtigkeiten dieser Gewinnbereiche angegeben. Des Weiteren werden auch die CPU-Zeiten angezeigt, die zum Parsen der Formeln und zur Lösung des Spiels gebraucht wurden.

Daneben gibt es im Falle einer symbolischen Lösung noch die Möglichkeit, die berechnete Gewinnstrategie zu überprüfen. Dazu kann im unteren Bereich der Seite ein Zustand angegeben werden, und das Programm berechnet die möglichen Nachfolgezustände ge-

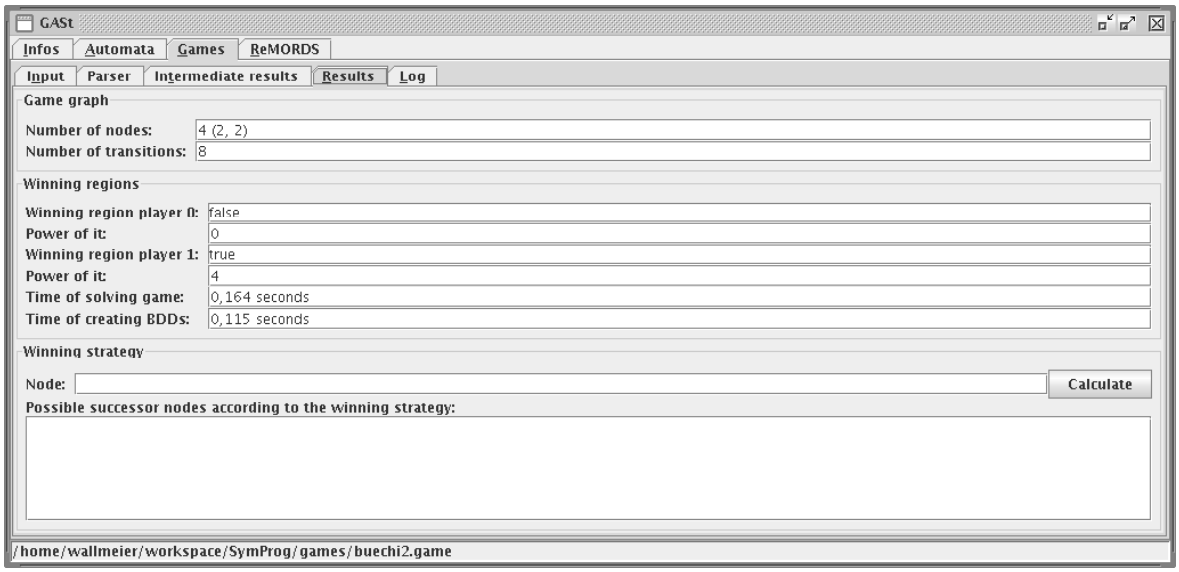


Abbildung 5.6: GUI unendliche Spiele: Ergebnis-Seite

mäß der Gewinnstrategie. Bei enumerativ gelösten Spielen wird stattdessen direkt die Gewinnstrategie ausgegeben.

5.1.3 Strategiesynthese für LSC-Spezifikationen

Die dritte Komponente der Experimentierplattform GASt dient zur Strategiesynthese von LSC-Spezifikationen. Sie basiert auf der Software REMoRDS (Requirements Engineering and Modeling of Reactive Distributed Systems), die im Rahmen der Doktorarbeit von Yves Bontemps [Bon05] an der Universität Namur, Belgien, entstanden ist. Die Vorgehensweise dabei ist, dass die LSC-Spezifikation eingelesen wird und diese dann in einen einfachen Streett-Automaten umgewandelt wird. Dieser wird schließlich durch Hinzufügen von Flags in ein einfaches Streett-Spiel transformiert. Dies ist notwendig, um sicherzustellen, dass beide Spieler abwechselnd ziehen. Der letzte Schritt besteht darin, einen Algorithmus zur Lösung des einfachen Streett-Spiels aufzurufen, den der Anwender auswählen kann. Dabei kommen die Algorithmen der zweiten Komponente von GASt zum Einsatz. Die synthetisierte Strategie kann dann als Implementierung des Systems genutzt werden. Für genauere Details der Transformation von LSC-Spezifikationen in einfache Streett-Spiele sei auf [Bon05] verwiesen.

Die Komponente unterstützt unter anderem die Funktionen, wie sie in Abbildung 5.7 zu sehen sind.

Syntax Check: Überprüft ob die Syntax der gegebenen Spezifikation korrekt ist.

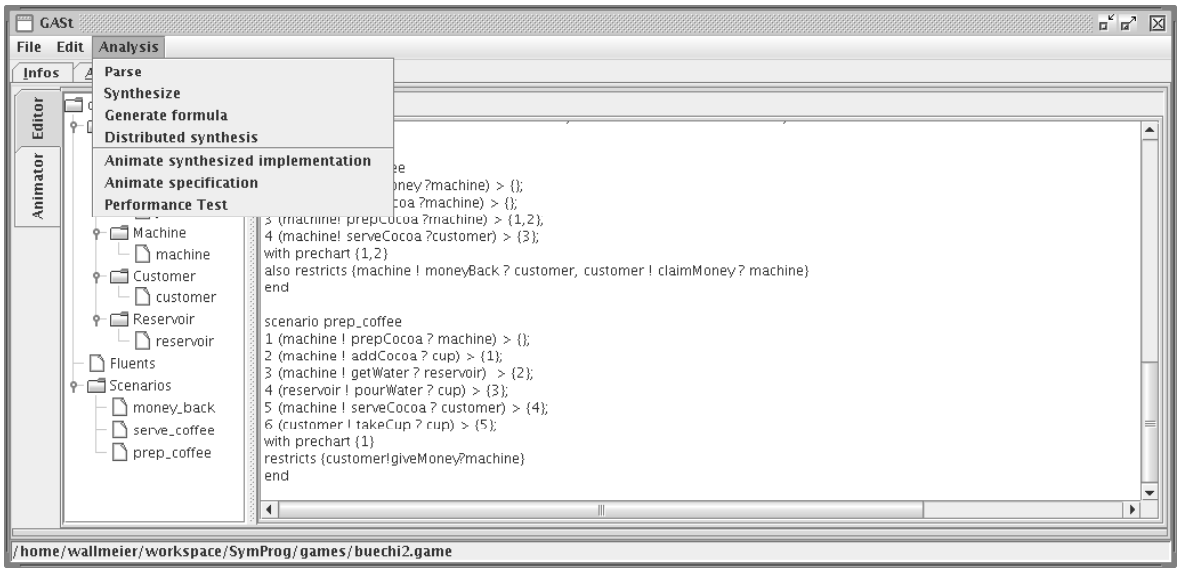


Abbildung 5.7: GUI REMoRDS: Auswahl-Seite

Animation: Führt die Spezifikation gemäß dem Algorithmus von Marelly und Harell [HM03] aus.

Realisierbarkeitsüberprüfung: Überprüft ob die gegebene LSC-Spezifikation realisierbar ist (für die Umwandlung der LSC-Spezifikation in ein einfaches Streett-Spiele siehe [Bon05]). Mittels des Animationsmoduls bekommt der Anwender auch Feedback über die Ergebnisse.

Die Eingabe der LSC-Spezifikation erfolgt textuell, siehe rechte Seite der Abbildung 5.7. Für die genaue Syntax sei an dieser Stelle auf Kapitel 6.2 in [Bon05] verwiesen.

5.1.3.1 Animation

Wenn die LSC-Spezifikation syntaktisch korrekt eingegeben und über das Menü übersetzt wurde, wird seine Struktur auf der linken Seite des Eingabebereichs dargestellt. Wählt der Benutzer ein Szenario aus und klickt im rechten Eingabebereich auf den Karteireiter “Visual” wird eine grafische Darstellung des Szenarios als universelles LSC (ULSC) angezeigt. Abbildung 5.8 stellt dies für das Beispiel einer Kaffeemaschine dar.

Neben der Visualisierung einzelner ULSCs hat der Benutzer auch die Möglichkeit, mit der LSC-Spezifikation zu “spielen”. Damit ist die interaktive Ausführung der Spezifikation gemeint. Dazu ist der Algorithmus von Harell und Marelly [HM03] implementiert worden. Die Bedienung ist einfach und intuitiv. Wieder für das Beispiel der Kaffeemaschine zeigt Abbildung 5.9, wie die Benutzeroberfläche für die Animation einer LSC-

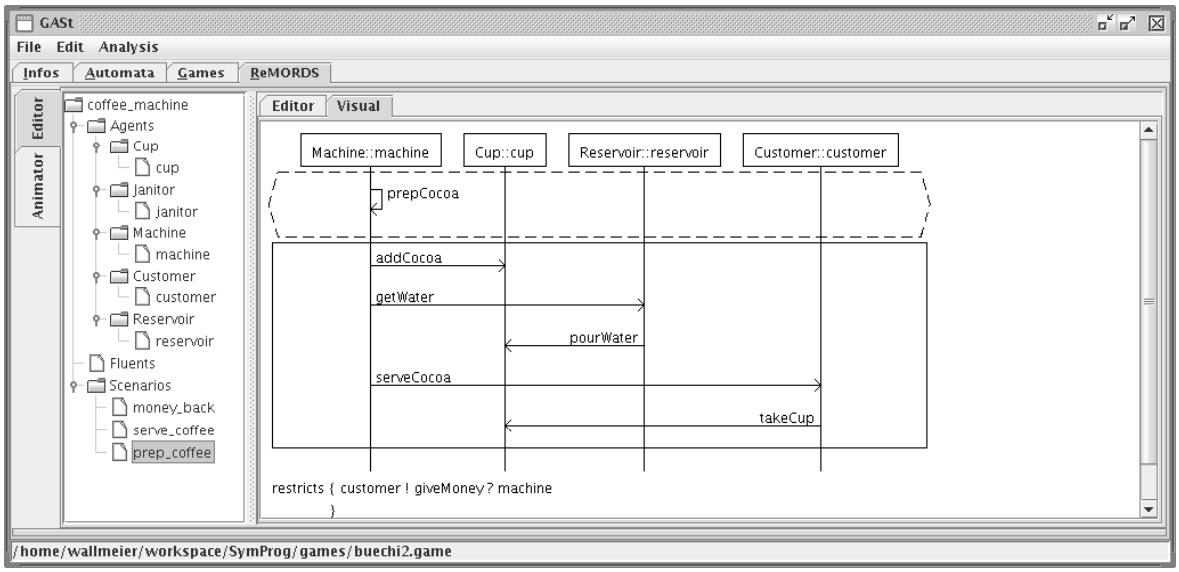


Abbildung 5.8: GUI REMoRDS: Visualisierung

Spezifikation aussieht. Um dorthin zu gelangen, wählt der Benutzer im Menü “Analysis” den Punkt “Animate specification” aus.

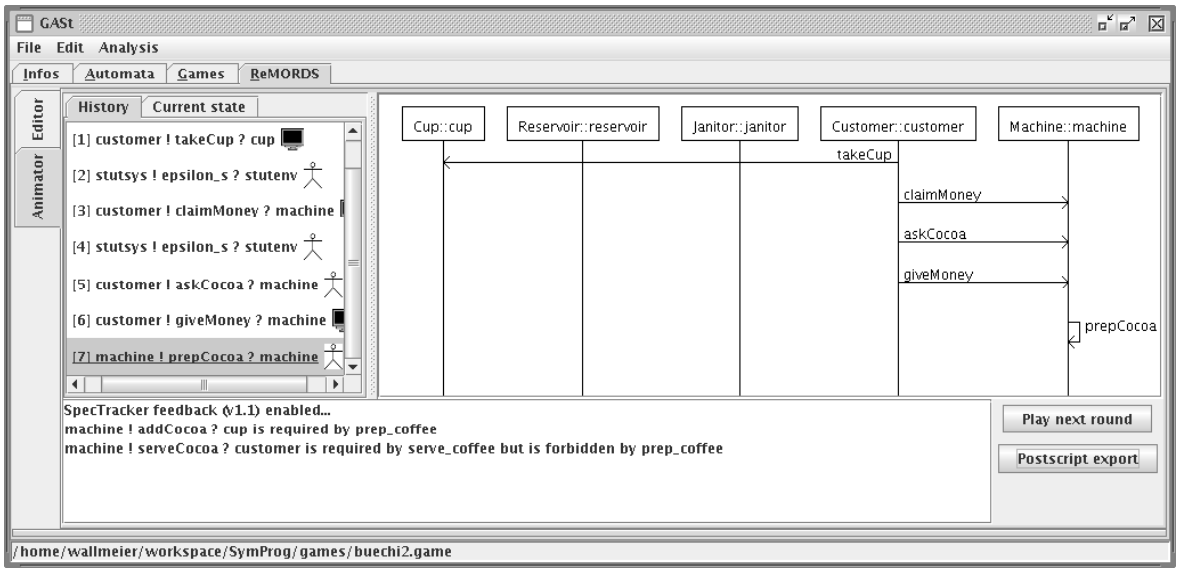


Abbildung 5.9: GUI REMoRDS: Animation

Die Oberfläche ist dabei in zwei Bereiche unterteilt, die über die entsprechenden Karteireiter ausgewählt werden: einerseits der “History”, die den kompletten bisherigen Lauf anzeigt, und andererseits dem Bereich “Current state”, der die partiellen Ausführungen aller ULSCs beinhaltet.

Im “History”-Modus wird im linken Bereich der Verlauf als Baum dargestellt, bei dem der aktuelle Knoten unterstrichen ist. Im rechten Bereich wird der Verlauf als Message-Sequence-Chart (MSC) dargestellt. Im unteren Bereich sieht der Benutzer, welche Ereignisse aktuell gefordert werden. Ist ein Ereignis gleichzeitig sowohl gefordert als auch verboten, so wird dieses auch dort dargestellt. Es liefert einen Hinweis darauf, dass es einen Konflikt in der LSC-Spezifikation gibt.

Um die Informationen genauer auswerten zu können, besteht auch die Möglichkeit, die Schnitte aller ULSCs im aktuellen Zustand zu betrachten. Dazu kann der Benutzer in der Ansicht “Current state” auf der linken Seite den Schnitt eines ULSC auswählen, den er betrachten möchte. Auf der rechten Seite wird dann das ULSC grafisch dargestellt und die Ereignisse dieses Schnitts, die bereits aufgetreten sind, sind im linken Bereich hervorgehoben – siehe auch Abbildung 5.10.

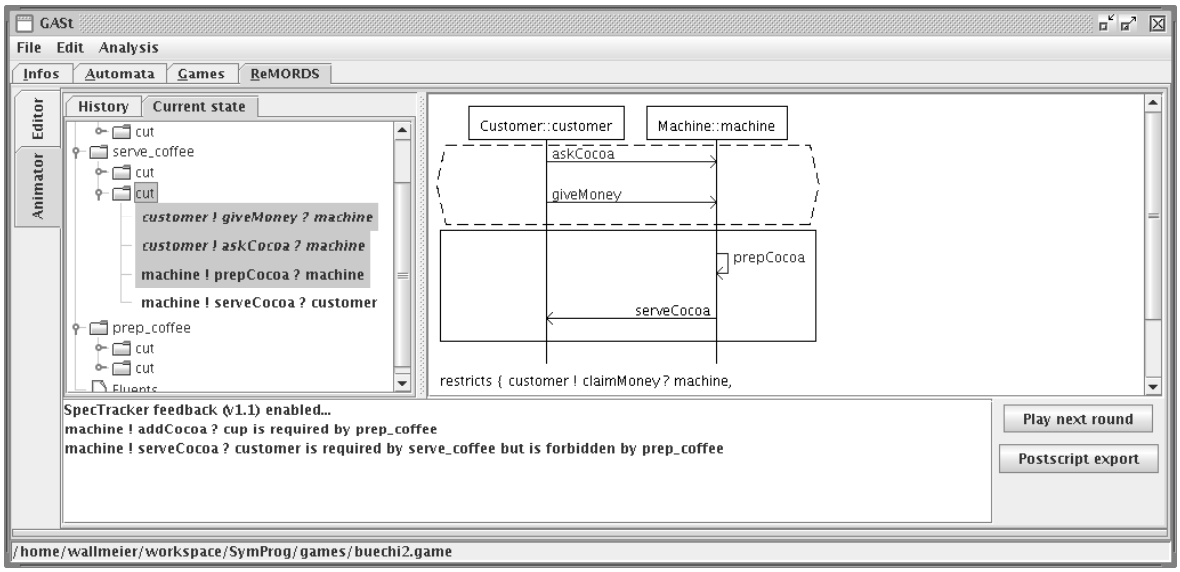


Abbildung 5.10: GUI REMoRDS: Visualisierung eines Zustands

Möchte man einen alternativen Pfad betrachten, kann man durch einen Doppelklick auf den entsprechenden Knoten in dem Baum auf der linken Seite den Lauf an dieser Stelle fortsetzen lassen.

Die Animation einer LSC Spezifikation ist somit sehr hilfreich, um Gegenbeispiele nachvollziehen zu können und um allgemein mit der Spezifikation experimentieren zu können.

5.1.3.2 Realisierbarkeitsüberprüfung

Bei der Realisierbarkeitsüberprüfung wird, wie bereits geschildert, die LSC-Spezifikation in ein einfaches Streett-Spiel umgewandelt und dieses anschließend gelöst. Den Algorithmus zur Lösung des einfachen Streett-Spiels kann der Benutzer über den Dialog, den er über das Menü “Edit” und dort dem Menüpunkt “Preferences” erreicht, im Kasten “Streett game solving” auswählen. Dort hat er die Auswahl zwischen dem *percur*-Algorithmus, der in Abschnitt 3.4.2.2 vorgestellt worden ist, oder einer Reduktion auf ein Paritätsspiel mit drei Farben. Hierbei können die Algorithmen zur Lösung eines Paritätsspiels zum Einsatz kommen, die in Abschnitt 3.4.2.4 verglichen worden sind. Möchte der Benutzer seine LSC-Spezifikation nun auf Realisierbarkeit überprüfen, wählt er im Menü “Analysis” den Punkt “Synthesis” aus. Anschließend bekommt er eine Rückmeldung, ob die Spezifikation realisierbar ist oder nicht – siehe für den ersten Fall die Abbildung 5.11.

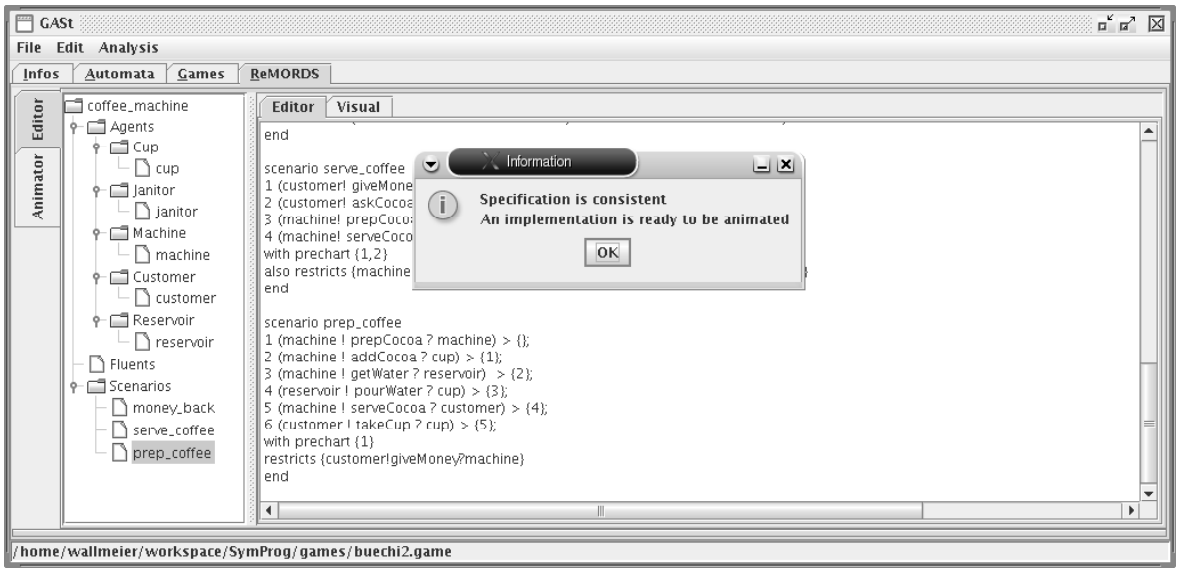


Abbildung 5.11: GUI REMoRDS: Synthese der LSC Spezifikation

Da die Strategie zu groß und zu komplex ist, um sie auf dem Bildschirm darstellen zu können, kann die eben vorgestellte Animationsmöglichkeit von LSC-Spezifikationen benutzt werden, um mit dem synthetisierten System zu experimentieren. Die Spiellösung des einfachen Streett-Spiels liefert entweder eine Strategie für den Controller, die das System korrekt (im Sinne von “gemäß der Spezifikation”) implementiert, oder einen Sabotageplan für die Umgebung, so dass das System die Spezifikation nicht erfüllen kann. In Abhängigkeit davon, ob die Spezifikation realisierbar ist, übernimmt der Benutzer des Programms entweder die Rolle des Controllers oder der Umgebung. Ist die

Spezifikation realisierbar, übernimmt der Benutzer die Rolle der Umgebung, während der Controller gemäß der Gewinnstrategie des einfachen Streett-Spiels agiert. Um mit dem synthetisierten System zu experimentieren, wählt der Benutzer im Menü "Analysis" den Menüpunkt "Animate synthesized implementation" aus und gelangt daraufhin in den Animationsmodus, der im letzten Abschnitt vorgestellt worden ist.

5.1.4 Zusammenspiel der Algorithmen

Um den Zusammenhang zwischen den unterschiedlichen Algorithmen, die in der Plattform GASt umgesetzt worden sind, zu verdeutlichen, soll Abbildung 5.12 dienen. Dabei ist für jeden Spieltyp eines unendlichen Zwei-Personen-Spiels die Bestimmung der Gewinnbereiche und der Gewinnstrategien umgesetzt worden. Jedes Spiel kann dazu auf die grundlegenden Algorithmen *Attraktor*, *Attraktor*$^+$ und *Recur* zurückgreifen. Die in Abbildung 5.12 unterstrichenen Spieltypen sind nur enumerativ implementiert.

5.2 Vorgehensweise bei der Implementierung

In diesem Abschnitt soll die Plattform GASt aus softwaretechnischer Sicht betrachtet werden. Ziel war es, eine einheitliche und einfach zu bedienende Programmoberfläche zu schaffen, die die Algorithmen der drei Komponenten von GASt vereinigt. Ein wichtiger Punkt bestand auch in der Anforderung, dass die Plattform einfach erweiterbar sein soll.

Ausgangspunkt von GASt war das Programm SymProg, welches im Rahmen der Diplomarbeit [Wal03] entstanden ist und zur symbolischen Synthese zustandsbasierter reaktiver Programme dient, d. h. zum Lösen von unendlichen Zwei-Personen-Spielen über einem symbolischen Zustandsraum. Somit handelt es sich bei GASt nicht um eine komplette Neuentwicklung, sondern um eine Weiterentwicklung von größerem Umfang. SymProg dient als Basis für die zweite Komponente von GASt und ist dafür deutlich erweitert worden:

- Neben dem symbolischen Zustandsraum wird jetzt auch der enumerative unterstützt.
- Weitere Spieltypen sind implementiert worden: verallgemeinerte Büchi-Spiele, Muller-Spiele, Streett-Spiele, einfache Streett-Spiele und Mean-Payoff-Spiele.
- Optimierung im klassischen Sinne (bezogen auf die Speichergröße für Gewinnstrategien) bei den verallgemeinerten Büchi-Spielen und Request-Response-Spielen (siehe auch Abschnitt 3.3). Für die Gewinnbedingungen "Staiger-Wagner" und "Request-Response" werden in der Diplomarbeit [Hol07] von Michael Holtmann

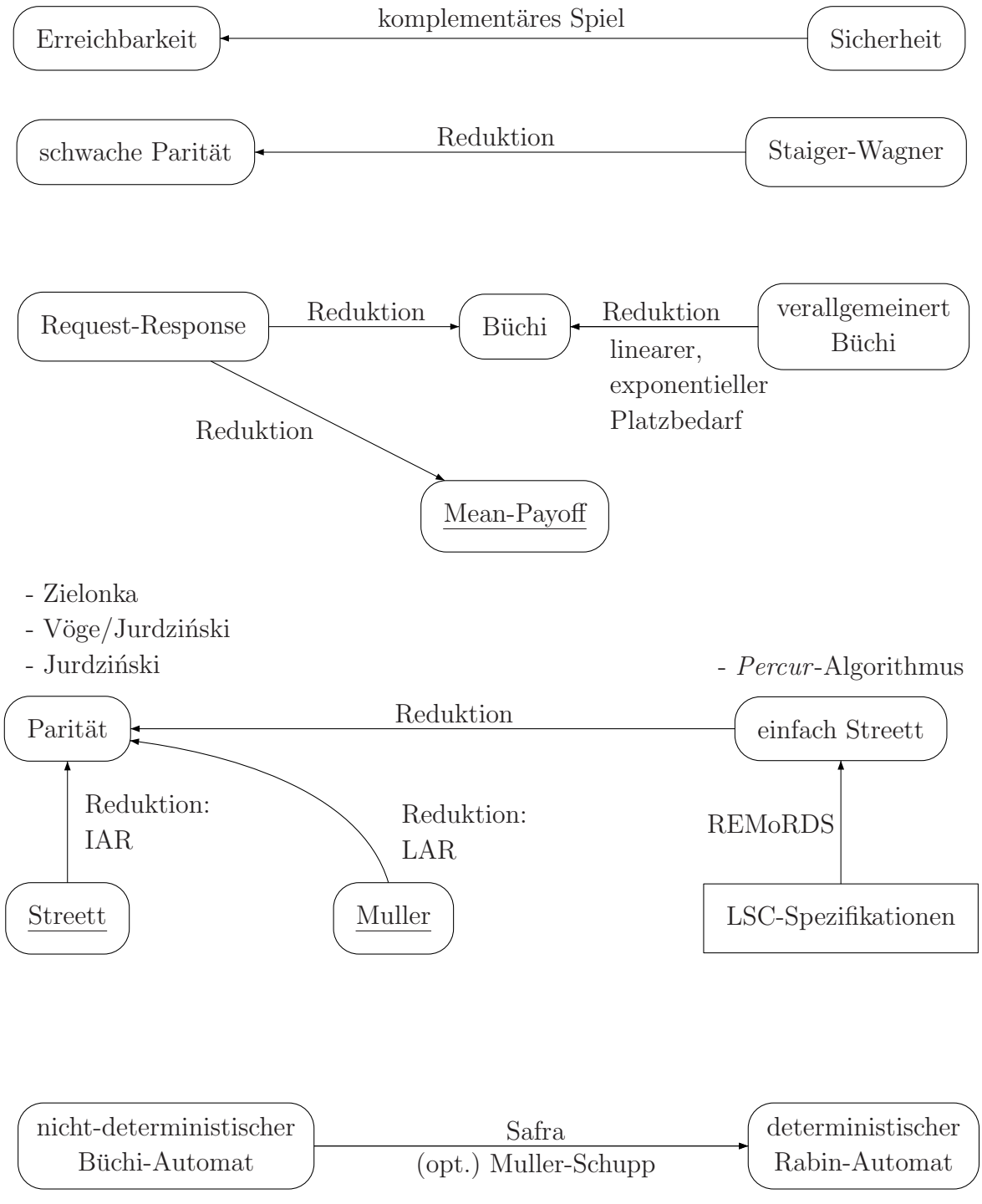


Abbildung 5.12: Übersicht über die implementierten Algorithmen

Reduktionen des Speichers entwickelt, die auch in der Experimentierplattform umgesetzt worden ist.

- Optimale Gewinnstrategien für Mean-Payoff- und Request-Response-Spiele, siehe Kapitel 4.

Für die anderen beiden Komponenten wurden bereits vorhandene Implementierungen in GASt integriert.

5.2.1 Einbindung der Komponenten

In GASt wurden insgesamt drei bereits bestehende Programme integriert:

1. Die bestehende Implementierung des Vöge/Jurdziński Algorithmus [SV00] in der Programmiersprache C zum Lösen von Paritätsspielen.
2. Die im Rahmen der Diplomarbeit von Christoph Schulte Althoff [Alt05] entstandene Anwendung OmegaDet (ebenfalls in C geschrieben) zur Determinisierung von Büchi-Automaten.
3. Das im Rahmen der Doktorarbeit von Yves Bontemps [Bon05] entstandene Programm REMoRDS (in Java geschrieben) zur Synthese von LSC-Spezifikationen.

Der Schwierigkeitsgrad zur Einbindung dieser drei Programme war recht unterschiedlich. Das erste Programm kann einfach von GASt aufgerufen werden, da es die Eingabe (der gefärbte Spielgraph des Paritätsspiels) in einer Datei erhält, die per Startparameter der Implementierung übergeben werden kann. Das Ergebnis des Paritätsspiels wird dann wieder in eine Datei geschrieben, die anschließend von GASt ausgelesen und ausgewertet werden kann.
Die Integration von OmegaDet war am anspruchsvollsten. Dies lag daran, dass OmegaDet die Möglichkeit der interaktiven Simulation der resultierenden, deterministischen Rabin-Automaten bietet. Somit kann es nicht einfach aus GASt aufgerufen und das Ergebnis ausgewertet werden, da dies durch den interaktiven Modus verhindert wird. Die Möglichkeit, OmegaDet komplett neu in Java zu schreiben, war nicht akzeptabel, da es eine ausgefeilte Speicherverwaltung benutzt. Umgesetzt wurde schließlich der Ansatz, das Programm um eine TCP/IP-Schnittstelle zu erweitern, mit der sich GASt dann verbinden kann, um OmegaDet steuern zu können. Dazu wird der Startparameter `-tcp` ausgewertet, der bewirkt, dass die Kommunikation mit dem Benutzer über den TCP-Port 5000 abgewickelt wird.
Die Einbindung von REMoRDS gestaltete sich einfacher, da es sich hierbei um eine Java-Anwendung handelt. Bei der Integration wurde Wert darauf gelegt, so wenig wie möglich an dem Programm selber zu ändern (bei größtmöglichem Code-Reuse), so dass weiterhin die separate Ausführung von REMoRDS möglich ist.

5.2.2 Softwareentwicklungsprozess

Die Erfahrungen bei der Entwicklung von SymProg [Wal03] führten dazu, dass auch bei der Entwicklung von GASt ein evolutionärer Entwicklungsprozess nach Sommerville [Som01] zum Einsatz kam. Ein evolutionärer Softwareentwicklungsprozess ist dadurch gekennzeichnet, dass zuerst eine initiale Version entwickelt wird (in diesem Fall kann man das Programm SymProg als eine solche ansehen), die im weiteren Verlauf weiterentwickelt und an die Ansprüche des Benutzers angepasst wird, bis ein adäquates System fertiggestellt ist. Abbildung 5.13 illustriert diesen Vorgang.

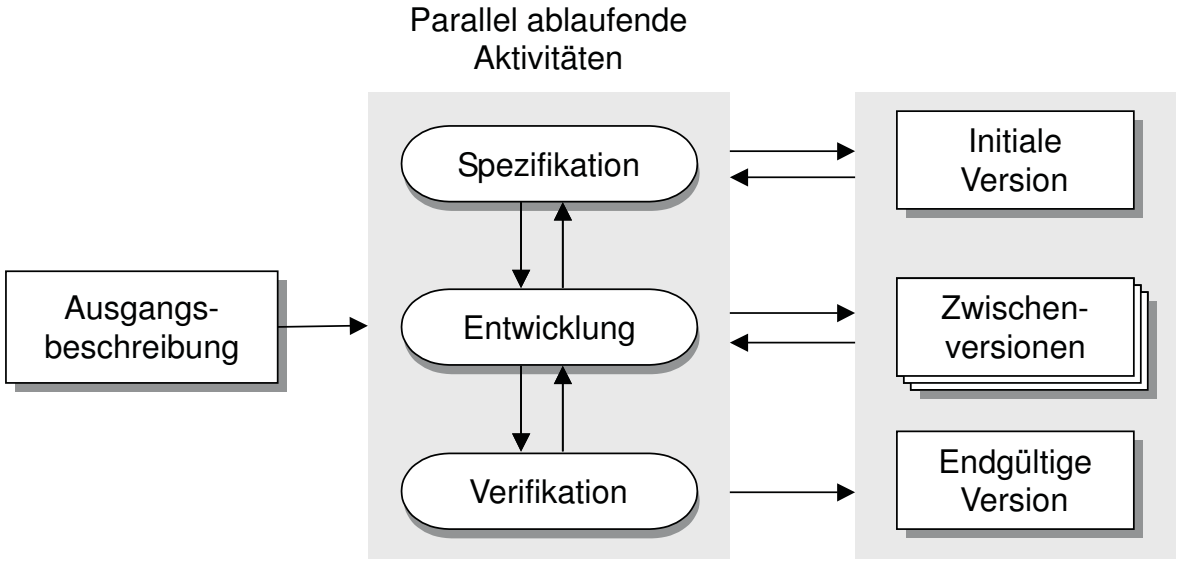


Abbildung 5.13: Prozessmodell "Evolutionary Development" nach [Som01]

Ein Merkmal dieses Entwicklungsprozesses ist, dass die Spezifikation, die Entwicklung und die Verifikation nicht als getrennte Aktivitäten durchgeführt werden, sondern parallel ablaufen. Angefangen wurde mit den Teilen des Systems, die bereits vollständig verstanden wurden, um das System schließlich um die Merkmale zu erweitern, die noch benötigt wurden. Dieser Ansatz wird nach Sommerville [Som01] auch "Exploratory Development" genannt (als Spezialfall des evolutionären Entwicklungsprozesses).

5.2.3 Die Architektur

In diesem Abschnitt wird die statische Architektur der Plattform GASt vorgestellt, d. h. welche Pakete bzw. Klassen existieren und in welcher Beziehung sie zueinander stehen. Für die externen Komponenten sei auf die entsprechenden Dokumentationen [Alt05] für OmegaDet bzw. [Bon05] für REMoRDS verwiesen, so dass sich die Beschreibung hier auf die zweite Komponente von GASt beschränkt. Abbildung 5.14 zeigt die Abhängigkeiten zwischen den Paketen von GASt.

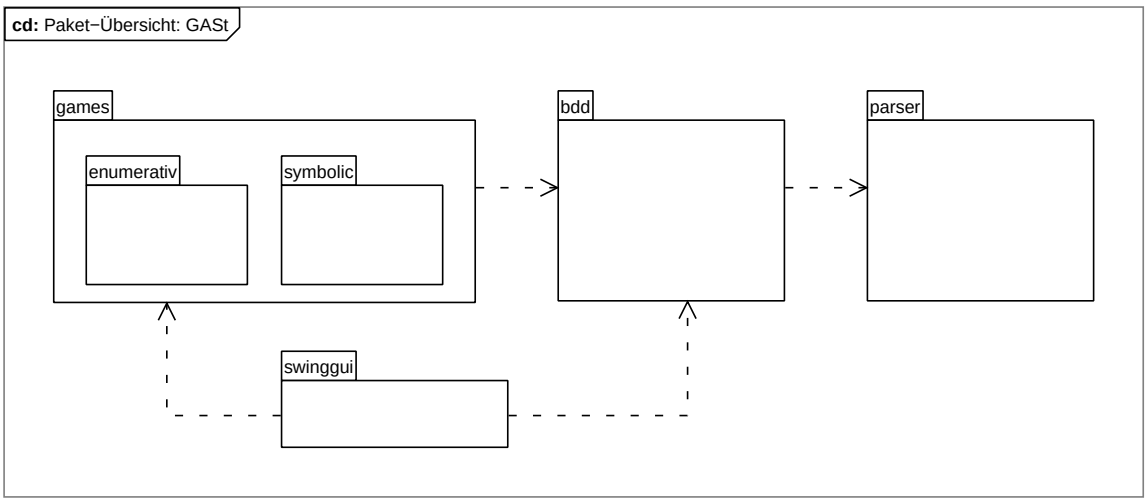


Abbildung 5.14: Pakete der Experimentierplattform GASt

Da sich an der Implementierung der BDDs und des Parsers der Eingabesprache nichts Wesentliches seit SymProg geändert hat, ist die Dokumentation der Pakete `bdd`, `parser` und `swinggui` in [Wal03] verwiesen. Das Paket `games` mit seinen Paketen `enumerativ` und `symbolic` stellt so etwas wie das Application Programming Interface (kurz API) dar, da es den algorithmischen Teil der implementierten Spiele beinhaltet. Eine grundlegende Anforderung hierbei war, dass die Programmoberfläche (auch GUI genannt) strikt von der Anwendungslogik (den implementierten Algorithmen) getrennt werden sollte. Die Programmoberfläche befindet sich ausschließlich im Paket `swinggui`. Das Paket `games` betrachten wir im folgenden Unterabschnitt noch genauer, wo wir auch darlegen, dass die Plattform GASt einfach erweiterbar ist.

5.2.3.1 Erweiterbarkeit

Vor der Aufnahme des enumerativen Zustandsraums hat die zweite Komponente von GASt, das ehemalige SymProg, zum Lösen von unendlichen Zwei-Personen-Spielen, ein komplettes Refactoring erfahren. Sämtliche Gemeinsamkeiten des symbolischen und enumerativen Zustandsraumes wurden in abstrakten Klassen bzw. Interfaces im Paket `games` gesammelt, die dann den Ausgangspunkt für die enumerative bzw. symbolische Implementierung der Zwei-Personen-Spiele bildeten. Konkret sind dabei die abstrakten Klassen `AbstractGame` und `AbstractGameGraph` sowie die Interfaces `ReducedGame`, `GameResult` und `ColoredGameGraph` entstanden. Das Klassendiagramm in Abbildung 5.15 zeigt einen Überblick der wichtigen Klassen/Interfaces des `games`-Pakets – dem algorithmischen Teil zum Lösen von unendlichen Zwei-Personen-Spielen.

Ein Hauptaugenmerk bei der Entwicklung von GASt war, dass das Einfügen von weiteren Spieltypen sich möglichst einfach gestalten soll. Trotzdem ist ein grundsätzliches Verständnis des Klassendiagramms von Abbildung 5.15 notwendig. Deshalb werden jetzt kurz die wichtigsten Klassen mit ihren Methoden vorgestellt.

`AbstractGame`

Die abstrakte Klasse `AbstractGame` ist die Basis für alle implementierten Spiele.

`getDebugOutput`	Diese Routine wird von der GUI verwendet, um die Informationen für die Zwischenergebnis-Seite abzurufen.
`getNumberOf-AvailableMethods`	Liefert die Anzahl der implementierten Algorithmen zur Lösung des Spiels.
`getMethod-Description`	Liefert den Namen des ausgewählten Algorithmus.
`hasAOptimization`	Hat der ausgewählte Algorithmus eine Optimierung?
`solve`	Diese Methode muss von jeder konkreten, abgeleiteten Klasse implementiert werden. Ein Aufruf dieser Methode bestimmt die Gewinnbereiche und -strategien beider Spieler.

`ReducedGame`

Interface für Spiele, die zur Lösung auf ein anderes Spiel reduziert werden.

`getResultGG`	Liefert den erweiterten Spielgraphen.
`getTimeCreatedGG`	Diese Routine wird von der GUI verwendet, um die CPU-Zeit für die Erstellung des erweiterten Spielgraphen zu ermitteln.
`setGameGraph-Listener`	Setzt einen `GameGraphListener`, der die Zeit für die Spiellösung abfragen kann.

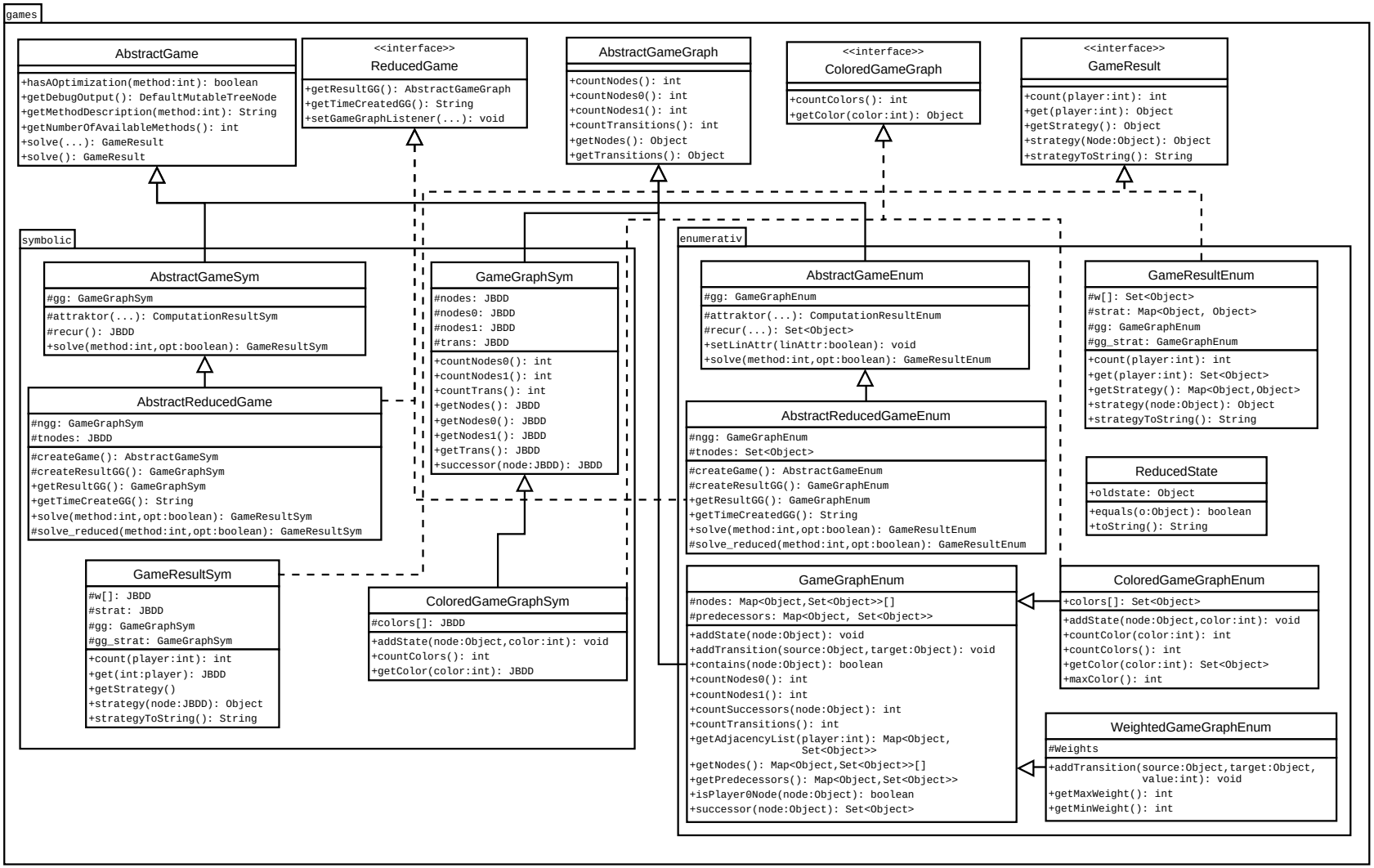


Abbildung 5.15: Das Paket games

`AbstractGameGraph`
Stellt die Grundlage zur Implementierung eines Spielgraphen dar.

`countNodes`	Liefert die Knoten des Spielgraphen.
`countNodes0()`	Liefert die Anzahl der Knoten von Spieler 0.
`countNodes1()`	Analog für Spieler 1.
`countTransitions`	Analog für die Transitionen.
`getNodes`	Gibt die Knoten von beiden Spielern zurück.
`getTransitions`	Liefert alle Transitionen.

`ColoredGameGraph`
Interface für einen gefärbten Spielgraphen

`countColors`	Liefert die Anzahl der Farben im Spielgraphen.
`getColor`	Liefert alle Zustände der ausgewählten Farbe.

`GameResult`
Stellt das Ergebnis eines Spiels dar.

`count`	Liefert die Mächtigkeit des Gewinnbereichs des ausgewählten Spieler.
`get`	Liefert die Zustände des Gewinnbereichs des ausgewählten Spielers.
`getStrategy`	Liefert die Gewinnstrategie des ausgewählten Spielers.
`strategy`	Liefert den Nachfolger des ausgewählten Knotens gemäß der Gewinnstrategie.
`strategyToString`	Liefert eine textuelle Darstellung der Gewinnstrategien.

Für die beiden Unterpakete `enumerativ` und `symbolic` werden bei der Vererbung implementierte abstrakte Methoden nicht nochmals aufgeführt.

`AbstractGameSym`
Dient als Ausgangspunkt für Spiele über einem symbolischen Zustandsraum, die direkt gelöst werden können.

`attraktor`	Diese Methode berechnet den Attraktor einer gegebenen Menge.
`recur`	Diese Methode berechnet die Recur-Menge einer gegebenen Menge.

`AbstractReducedGameSym`
Ist der Ausgangspunkt für Spiele, die mittels einer Spielreduktion gelöst werden.

`createGame`	Erzeugt das reduzierte Spiel.
`createResultGG`	Muss von einer abgeleiteten Klasse implementiert werden und soll den erweiterten Spielgraphen erzeugen.
`solve_reduced`	Soll das reduzierte Spiel erzeugen und lösen.

GameGraphSym
Stellt einen Spielgraphen dar, bei dem die Knoten und Transitionen durch boolesche Formeln gegeben sind.

getNodes0	Liefert die Knoten von Spieler 0.
getNodes1	Analog für Spieler 1.
successor	Liefert die Nachfolger des ausgewählten Knoten.

ColoredGameGraphSym
Stellt einen gefärbten, symbolischen Spielgraphen dar.

GameResultSym
Enthält eine Spiellösung.

Auch wenn es einige Gemeinsamkeiten in den Paketen enumerativ und symbolic gibt, wird an dieser Stelle auch das Paket enumerativ noch vorgestellt. Ein deutlicher Unterschied zwischen beiden Paketen besteht in der Art und Weise, wie ein Spielgraph repräsentiert wird.

AbstractGameEnum
Ausgangspunkt für ein Spiel über einem enumerativen Spielgraphen, welches direkt gelöst wird.

attraktor	Diese Methode berechnet den Attraktor zu einer gegebenen Menge.
recur	Diese Methode berechnet die Recur-Menge zu einer gegebenen Menge.
setLinAttr	Soll die Linearzeitvariante für den Attraktor benutzt werden (dies ist der Standard)?

AbstractReducedGameEnum
Der Ausgangspunkt für ein Spiel, welches zum Lösen auf ein anderes reduziert wird.

createGame	Erzeugt das reduzierte Spiel.
createResultGG	Muss von einer abgeleiteten Klasse implementiert werden und erzeugt den erweiterten Spielgraphen.
solve_reduced	Löst das reduzierte Spiel.

GameGraphEnum
Stellt einen enumerativen Spielgraphen dar (alle Knoten sind als Java-Objekte implementiert).

`addState`	Fügt den gegebenen Knoten zum Spielgraphen hinzu.
`addTransition`	Fügt die gegebene Transition zum Spielgraphen hinzu – Bedingung: beide Knoten existieren bereits im Spielgraphen.
`contains`	Überprüft, ob der gegebene Knoten bereits im Spielgraphen enthalten ist.
`countSuccessors`	Liefert die Anzahl der Nachfolger des gegebenen Knotens im Spielgraphen.
`getAdjacencyList`	Liefert alle Transitionen von allen Knoten des gegebenen Spielers.
`getPredecessors`	Liefert für alle Knoten die Vorgänger im Spielgraphen.
`isPlayer0Node`	Gibt `true` aus, wenn der gegebene Knoten Spieler 0 gehört.
`successor`	Liefert alle Nachfolger zu dem gegebenen Knoten.

`ColoredGameGraphEnum`
Stellt einen gefärbten, enumerativen Spielgraphen dar.

`countColor`	Bestimmt wie viele Knoten mit der gegebenen Farbe markiert sind.
`maxcolor`	Lieferte die höchste Farbe, mit denen Knoten gefärbt sind.

`WeightedGameGraphEnum`
Stellt einen Spielgraphen dar, bei dem die Kanten ein Gewicht haben.

`getMaxWeight`	Liefert das größte Kantengewicht.
`getMinWeight`	Analog dazu das kleinste Kantengewicht.

`GameResultEnum`
Beinhaltet eine Spiellösung

`ReducedState`
Dieses Interface sollte als Grundlage für Zustände im erweiterten Spielgraphen bei einer Spielreduktion benutzt werden.

`equals`	Entscheidet, ob der übergebene Zustand mit diesem übereinstimmt.
`toString`	Liefert eine textuelle Darstellung des Zustands.

5.2.3.2 Hinzufügen von neuen Spielen

Um weitere Spiele in GASt hinzuzufügen, sind mehrere Schritte notwendig:

1. Für den algorithmischen Teil des Spiels stehen vier verschiedene Klassen als Ausgangspunkt zur Verfügung, in Abhängigkeit davon, ob das Spiel direkt oder mittels Reduktion gelöst wird und ob ein enumerativer oder symbolischer Zustandsraum benutzt werden soll: `AbstractGameEnum`, `AbstractReducedGameEnum`,

`AbstractGameSym` und `AbstractReducedGameSym`. Diese abstrakten Klassen bieten Algorithmen wie Attraktor und Recur als Unterstützung zum Lösen des Spiels an. Wird das Spiel direkt gelöst, muss im Wesentlichen nur die Methode `solve` implementiert werden. Anders sieht es aus, wenn das Spiel mittels einer Reduktion gelöst wird. In diesem Fall muss immer die Methode `createResultGG` implementiert werden, die den erweiterten Spielgraphen erzeugt. Dabei sollte man die transformierten Knoten des gegebenen Spielgraphens in der Variablen `tnodes` speichern. Dann reicht es aus, die Methode `createGame` zu implementieren, die das reduzierte Spiel erzeugt. Reicht dies zur Lösung des Spiels nicht aus, besteht auch die Möglichkeit, die Methode `solve_reduced` zu überschreiben und an die eigenen Bedürfnisse anzupassen.

2. Damit das Spiel auch in der Programmoberfläche zur Verfügung steht, muss es dort noch integriert werden. Dazu ist es notwendig, für das Spiel eine Klasse von `AbstractGuiGame` im Paket `swinggui.game` abzuleiten. Für den Konstruktor muss eine passende Klasse gefunden werden, die das `Acceptance`-Interface implementiert. Zur Auswahl dafür stehen die Klassen `TableAcc`, für Gewinnbedingungen, die tabellarisch eingegeben werden können, und `SingleAcc`, für Gewinnbedingungen, die nur aus einer einzigen Menge bestehen. Es muss dann nur noch die Methode `checkAcc` implementiert werden, die die eingegebene Gewinnbedingung überprüft und eine Instanz des zugrundeliegenden Spiels erzeugt.

3. Der letzte Schritt besteht darin, das neu erstellte Spiel in der Programmoberfläche anzumelden. Dafür muss die Methode `registerGames` in der Klasse `GuiGamesAdmin` im Paket `swinggui.games` um eine Zeile erweitert werden, die gleich aufgebaut ist wie die vorhandenen, nur dass der Klassenname des neu erstellten Spiels eingetragen wird – siehe hierzu Listing 5.1.

5.2.3.3 Entwurfsmuster

Entwurfsmuster beschreiben eine in der Praxis bewährte Schablone für ein Entwurfsproblem. Sie stellen damit eine wiederverwendbare Vorlage zur Problemlösung dar. Der Durchbruch der Entwurfsmuster kam mit dem Buch [GHJV96] von Gamma, Helm, Johnson und Vlissides, welches Entwurfsmuster in drei Klassen entsprechend ihrem Zweck einteilt:

Erzeugungsmuster: Sie abstrahieren den Objekterzeugungsprozess. So kann man etwa die Anzahl von erzeugten Objekten einer Klasse limitieren wollen, oder man möchte den konkreten Typ der erzeugten Objekte in Abhängigkeit von den jeweiligen Bedingungen anpassen.

```
public void registerGames() {
  games = new Vector<AbstractGuiGame>();
  games.add(new ExistenceGuiGame());
  games.add(new SafetyGuiGame());
  games.add(new WeakParityGuiGame());
  games.add(new StaigerWagnerGuiGame());
  games.add(new RequestResponseGuiGame());
  games.add(new BuechiGuiGame());
  games.add(new GenBuechiGuiGame());
  games.add(new ParityGuiGame());
  games.add(new MullerGuiGame());
  games.add(new StreettGuiGame());
}
```

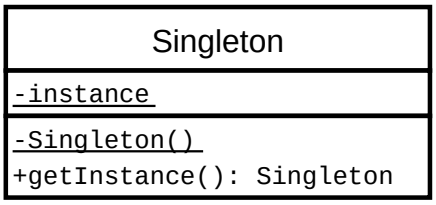
Listing 5.1: registerGames

Strukturmuster: Sie dienen zur Vereinfachung der Struktur zwischen Klassen. Komplexe Beziehungsgeflechte können beispielsweise über vermittelnde Klassen oder Schnittstellen logisch vereinfacht werden.

Verhaltensmuster: Sie betreffen das Verhalten der Klassen und beziehen sich auf die Zusammenarbeit und den Nachrichtenaustausch zwischen Klassen.

Bei der Entwicklung von GASt wurde versucht, möglichst viel mit Entwurfsmustern zu arbeiten. Verwendet wurden unter anderem die folgenden Muster:

- Als ein Erzeugungsmuster kam das Muster des Einzelstücks (Singleton) mehrfach zum Einsatz. Das Muster dient dazu, dass sichergestellt wird, dass nur ein Objekt zu einer Klasse existiert und ein einfacher Zugriff auf dieses Objekt möglich ist. Benutzt wurde dieses Muster bei den Klassen `JBDDMaster`, `GuiGamesAdmin` und `SymProg`. Das UML-Diagramm dazu sieht folgendermaßen aus:



- Als ein Verhaltensmuster kam die Schablonenmethode (template method) zur Anwendung. Diese ist dadurch gekennzeichnet, dass in einer abstrakten Klasse ein

Skelett eines Algorithmus in einer Operation definiert wird. Die konkrete Umsetzung der einzelnen Schritte wird an die Unterklasse delegiert. Somit ist es möglich, dass einzelne Schritte eines Algorithmus modifiziert oder überschrieben werden, ohne die grundlegende Struktur zu verändern. Dieses Muster wurde unter anderem in den Klassen `AbstractReducedGameEnum` und `AbstractReducedGameSym` eingesetzt. Dort wird in der Methode `solve_reduced` auf die abstrakte Methode `createGame` zurückgegriffen.

- Als ein weiteres Verhaltensmuster wurde das Besucher-Entwurfsmuster verwendet. Es dient der Kapselung einer auf Elementen einer Objektstruktur auszuführenden Operation. Die Definition weiterer Operationen ist somit ohne Änderung der betroffenen Elementklassen möglich. Das Besuchermuster lagert Operationen in externe Besucherklassen aus. Dazu müssen die zu besuchenden Klassen jedoch eine Schnittstelle zum Empfang eines Besuchers definieren. In GASt wird dieses Entwurfsmuster dazu verwendet, den Syntaxbaum, den der Parser aus der Eingabe erzeugt hat, in einen BDD umzuwandeln.

- Das Iterator-Entwurfsmuster gehört ebenfalls zu der Klasse der Verhaltensmuster. Es stellt Möglichkeiten zur Verfügung, auf Elemente einer aggregierten Struktur sequentiell zuzugreifen, ohne die Struktur zu enthüllen. Dieses Muster kommt zum Beispiel bei den erfüllenden Belegungen eines BDDs zum Tragen.

5.2.4 Qualitätssicherung

Zur Qualitätssicherung der Plattform GASt werden automatische Tests mittels des JUnit-Rahmenwerks benutzt, welches von Erich Gamma und Kent Beck (siehe [BG98]) entwickelt worden ist, um auf einfache Weise Regressionstests durchführen zu können. Unter Regressionstests versteht man Tests, die bei Veränderungen am Programm ausgeführt werden, um sicherzustellen, dass die Änderungen keine negativen Nebeneffekte haben. Dies reduziert den Testaufwand für eine neue Programmversion erheblich, da die bereits vorhandenen, automatisierten Tests wieder benutzt werden können.

Erst durch diese automatisierten Tests können Refactoring-Maßnahmen gefahrlos durchgeführt werden, da nur so sichergestellt werden kann, dass sich das beobachtbare Verhalten des Programms nicht ändert. Dies ist ein wesentlicher Punkt, da bei der Entwicklung von GASt bedingt durch den gewählten Softwareentwicklungsprozess viele Refactoring-Maßnahmen nötig wurden.

Die Testfälle sind in der Regel vor der Entwicklung des zu testenden Quellcodes geschrieben worden, damit die Testfälle gemäß der Spezifikation und nicht aufgrund des Quelltextes geschrieben werden. Dies entspricht der Philosophie des “Extreme Programming” [Bec98, Bec00]. Sie folgt der Devise:

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

Ausgehend von SymProg unterteilen sich die Tests auf zwei verschiedene Gruppen:

1. Test der BDDs (sowohl das zugrundeliegende BuDDy-Paket, als auch der Parser zur Erzeugung der BDDs)
2. Test der implementierten Spiele

Beide Gruppen wurden um zusätzliche Testfälle erweitert, um eine möglichst hohe Zweigüberdeckung zu erreichen. Bei den Tests für das BuDDy-Paket, dem die Dokumentation des Pakets zugrunde lag, handelt es sich nach Sommerville [Som01] um *Blackbox-Tests*, da die BDD-Klassen fast ausschließlich aus Aufrufen des C++-Codes mittels JNI[5] bestehen. Die Tests des Parsers sind hingegen *Whitebox-Tests* und basieren auf der Grammatik aus Abschnitt 5.1.2.2. Weiterhin existiert für jedes implementierte Spiel sowohl für den enumerativen als auch für den symbolischen Zustandsraum (sofern beide Varianten implementiert sind) jeweils eine eigene Sammlung von Testfällen.

[5] Java Native Interface

Kapitel 6

Zusammenfassung und Ausblick

Diese Arbeit beschäftigte sich mit dem Lösen unendlicher Zwei-Personen-Spiele. Eine notwendige Voraussetzung für die Spezifikation unendlicher Zwei-Personen-Spiele durch ω-Automaten ist die Determinisierung. Dazu wurden die Determinisierungsverfahren von Safra und Muller-Schupp vorgestellt und die Gemeinsamkeiten wie auch deren Unterschiede herausgearbeitet. Die Beispiele von Michel zeigten einen deutlichen Unterschied der beiden Verfahren mit einem Vorteil für die Konstruktion von Safra. Gründe für diesen Effekt wurden erläutert, und eine Verbesserung der Muller-Schupp-Konstruktion wurde eingeführt, aus der für einige Beispiele drastisch kleinere Automaten resultieren. Als ein Ergebnis der experimentellen Studien wurde eine stärkere Verbindung der beiden Verfahren als bisher bekannt entwickelt. Dabei zeigte sich auch, dass die Safra-Konstruktion in Bezug auf die resultierende Automatengröße überlegen ist.

Trotz ausgiebiger Forschung haben die Algorithmen zur Determinisierung nicht die Ebene erreicht, um realistische Anwendungen zu ermöglichen, im Gegensatz zu den vielen Implementierungen, die auf Büchi-Automaten und alternierenden Automaten basieren und im Bereich des Model Checking zum Einsatz kommen. Auf der anderen Seite ist die Determinisierung eine notwendige Voraussetzung in einigen Bereichen, wie etwa bei der Spezifikation unendlicher Spiele. Die Frage der Determinisierung muss weiter erforscht werden, bis praxistaugliche Verfahren vorliegen. Hierzu gehört die Konzeption geeigneter Vorverarbeitungs- bzw. Nachbearbeitungsschritte. Die Vorverarbeitung würde eine Analyse des gegebenen Büchi-Automaten beinhalten, möglicherweise eine Reduktion mit Methoden aus [EWS01] und eine Überprüfung, ob der Automat in einen äquivalenten co-Büchi-Automaten transformiert werden kann. Die co-Büchi-Darstellung hat den Vorteil, dass die einfachere Determinisierungsmethode von Miyano und Hayashi [MH84] benutzt werden kann (siehe auch [BJW05]). Die Nachbearbeitung würde sich mit der Reduktion oder sogar Minimierung des deterministischen Rabin-Automaten beschäftigen und auch den Parameter berücksichtigen, der hier nicht wei-

ter beachtet wurde, nämlich die Komplexität der Akzeptierbedingung (die Anzahl der Rabin-Paare). Einige dieser Aspekte sind in der Arbeit von Klein und Baier [KB05] betrachtet worden. Dort wurden einige Verbesserungen des Safra-Algorithmus vorgestellt und ein Nachbearbeitungsschritt entwickelt, der auf der Berechnung eines Bisimulationsquotienten beruht.

Im Bereich der unendlichen Zwei-Personen-Spiele wurden besonders die Spieltypen der Request-Response-Spiele und verallgemeinerten Büchi-Spiele sowie ihre Beziehung zueinander betrachtet. Dabei stellt die Request-Response-Bedingung eine einfache Form der Liveness-Bedingung dar. Weiterhin wurde die "klassische" Optimierung der Speichergröße von Automatenstrategien in Request-Response- und verallgemeinerten Büchi-Spielen untersucht. Die dabei erzielten Ergebnisse reduzieren die Speichergröße allerdings nur um konstante Faktoren.

Eine Anwendung der unendlichen Zwei-Personen-Spiele bestand in der Analyse von Live-Sequence-Charts-Spezifikationen. Die dabei zugrunde liegenden Spiele sind einfache Streett-Spiele, d. h. unendliche Zwei-Personen-Spiele, deren Gewinnbedingung durch eine Streett-Bedingung mit genau einem Paar gegeben ist. Diese Spiele entstehen in der Praxis, wenn man das Verhalten von reaktiven Systemen mit Fairness-Bedingungen an die Umgebung spezifizieren möchte. Derartige Spiele sind mächtiger als Büchi-Spiele und deutlich einfacher als normale Streett-/Rabin-Spiele. Insbesondere haben sie positionale Gewinnstrategien und können in quadratischer Zeit gelöst werden. Auf der praktischen Seite entstehen sie im Kontext der Szenario-basierten Synthese.

In dieser Arbeit wurde ein neuer Algorithmus zur Lösung von einfachen Streett-Spielen eingeführt, der ohne eine Reduktion auf andere Spiele auskommt und auch symbolisch umgesetzt werden kann. Das Laufzeitverhalten dieses Algorithmus wurde sowohl auf theoretischer als auch auf praktischer Ebene mit der Reduktion auf Paritätsspiele mit genau drei Farben verglichen. Dabei stellte sich heraus, dass dieser Algorithmus die bekannten Paritätsspielalgorithmen von Zielonka, Jurdziński und Vöge/Jurdziński schlägt.

Ein wesentlicher Teil dieser Arbeit beschäftigte sich mit der Optimierung der Request-Response-Gewinnstrategien. Hierbei ist nicht die "klassische" Optimierung in Bezug auf die Strategiegröße gemeint, sondern in Bezug auf die Güte des Gewinnens. Die RR-Gewinnbedingung fordert zunächst nur, dass, wenn eine Anforderungsmenge besucht worden ist, die passende Antwortmenge irgendwann aufgesucht wird. Wann dieses geschieht, ist nicht spezifiziert. Der erste Schritt bestand darin, ein geeignetes Maß für die Güte des Gewinnens zu definieren. Dazu wurden drei verschiedene Ansätze für die Definition eines Partiewertes vorgestellt und diskutiert: die durchschnittliche Anzahl offener Anforderungen, die durchschnittliche Wartezeit und eine quadratische Gewichtung der Wartezeit. Basierend auf der Partiewertdefinition konnte dann ein Strategiewert defi-

niert werden. Die wissenschaftliche Herausforderung bestand darin, das kontinuierliche Maß der Strategiewerte zu optimieren (anstelle des diskreten Maßes der Speichergröße bei der "klassischen" Optimierung). Es stellte sich heraus, dass optimale Gewinnstrategien in RR-Spielen mit endlichem Speicher auskommen und diese durch eine Reduktion auf ein Mean-Payoff-Spiel berechnet werden können. Offene Fragen in diesem Bereich sind:

- Kann der Parameter w_{max}, der die Speichergröße signalisiert, verbessert werden?
- Bei der bisherigen Betrachtung sind alle RR-Paare gleichberechtigt. Was muss geändert werden, wenn eine Gewichtung zwischen den Paaren vorliegt?
- Gibt es einen Zusammenhang zwischen Strategiegröße und Strategiegüte?
- Sind optimale Strategien für verallgemeinerte Büchi-Spiele effizienter als über den Ansatz der Request-Response-Spiele berechenbar?

Somit hat diese Arbeit erste Ergebnisse zur quantitativen Analyse von Request-Response-Spielen gezeigt. Aufgegriffen werden diese Ergebnisse aktuell von Florian Horn, der optimale Strategien in "finitary" Paritäts- bzw. Streett-Spielen untersucht.

Neben den theoretischen Ergebnissen war ein wesentlicher Bestandteil dieser Arbeit die Implementierung all dieser Algorithmen in der Experimentierplattform GASt. Dort wurden erstmalig eine große Anzahl klassischer Algorithmen der algorithmischen Theorie unendlicher Spiele umgesetzt, ebenso wie die in dieser Arbeit entwickelten Algorithmen.

Literaturverzeichnis

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

[Alt05] Christoph Schulte Althoff. Konstruktion deterministischer ω-Automaten: Eine vergleichende Analyse der Algorithmen von Safra und Muller/Schupp. Diplomarbeit, RWTH Aachen, April 2005.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.

[Bec98] Kent Beck. Extreme programming: A humanistic discipline of software development. *Lecture Notes in Computer Science*, 1382:1–6, 1998.

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

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[BHK03] Yves Bontemps, Patrick Heymans, and Hillel Kugler. Applying LSCs to the specification of an air traffic control system. In Sebastian Uchitel and Francis Bordeleau, editors, *Proc. of the 2nd Int. Workshop on "Scenarios and State Machines: Models, Algorithms and Tools"(SCESM'03), at the 25th Int. Conf. on Soft. Eng. (ICSE'03)*, Portland, OR, USA, May 2003. IEEE.

[BJW05] Bernard Boigelot, Sebastien Jodogne, and Pierre Wolper. An Effective Decision Procedure for Linear Arithmetic with Integer and Real Variables. *ACM Transaction on Computational Logic*, 6(3):614–633, 2005.

[Bon03] Yves Bontemps. Realizability of scenario-based specifications. Diplôme d'études approfondies, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), rue Grandgagnage, 21, B5000 - Namur (Belgium), September 2003.

[Bon05] Yves Bontemps. *Relating Inter-Agent and Intra-Agent Specifications (The Case of Live Sequence Charts)*. PhD thesis, Computer Science Dept, University of Namur, April 2005.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BS03] Yves Bontemps and Pierre-Yves Schobbens. Synthesizing open reactive systems from scenario-based specifications. In Felice Balarin and Johan Lilius, editors, *Proc. of the 3rd Int. Conf. on Applications of Concurrency to System Design (ACSD'03)*, pages 41–50, Guimarães, Portugal, June 2003. IEEE Computer Science Press.

[BuD] BuDDy – A Binary Decision Diagram Package. Available at `http://sourceforge.net/projects/buddy`.

[Büc62] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science*, pages 1–12, Stanford, CA, USA, 1962. Stanford University Press.

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

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

[CH06] Krishnendu Chatterjee and Thomas A. Henzinger. Finitary winning in omega-regular games. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2006.

[Chu62] Alonzo Church. Logic, arithmetic and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35, 1962.

[CS01] E. M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 24, pages 1635–1790. Elsevier Science, 2001.

[CTA] Center-TRACON Automation System. `http://www.ctas.arc.nasa.gov/`.

[dAHM03] Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. Discounting the future in systems theory. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 1022–1037. Springer, 2003.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[Dic13] Leonard E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Amer. Journal of Math.*, 35:413–422, 1913.

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

[EM79] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *Internal Journal of Game Theory*, 8:109–113, 1979.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[EWS01] K. Etessami, Th. Wilke, and R. A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In *Proc. 28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707, 2001.

[GH82] Yuri Gurevich and Leo Harrington. Trees, automata, and games. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 60–65, San Francisco, California, 5–7 May 1982.

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

[GKK88] V.A. Gurvich, A.V. Karzanov, and L.G. Khachiyan. Cyclic games and an algorithm to find minimax cycle means in directed graphs. *USSR Computational Mathematics and Mathematical Physics*, 28(5):85–91, 1988.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[GTW02] E. Grädel, W. Thomas, and Th. Wilke. *Automata, Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[GZ05] Hugo Gimbert and Wieslaw Zielonka. Games where you can play optimally without any memory. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 of *Lecture Notes in Computer Science*, pages 428–442. Springer, 2005.

[GZ06] Hugo Gimbert and Wieslaw Zielonka. Deterministic priority mean-payoff games as limits of discounted games. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 312–323. Springer, 2006.

[HM03] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[Hol07] Michael Holtmann. Memory Reduction for Strategies in Infinite Games. Diplomarbeit (revised version), RWTH Aachen, Dezember 2007.

[Hor07] Florian Horn. Faster algorithms for finitary games. In Orna Grumberg and Michael Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 472–484. Springer-Verlag, 2007.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In K.R. Apt, editor, *NATO ASI Series*, volume F-13, pages 477–498. Springer, January 1985.

[HRS05] Aidan Harding, Mark Ryan, and Pierre-Yves Schobbens. A new algorithm for strategy synthesis in LTL games. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2005.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[Hüt03] Patrick Hütten. Automatische Synthese von Controllern für Request-Response-Spezifikationen. Diplomarbeit, RWTH Aachen, Oktober 2003.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301, Lille, France, February 2000. Springer-Verlag.

[KB05] Joachim Klein and Christel Baier. Experiments with deterministic omega-automata for formulas of linear temporal logic. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *CIAA*, volume 3845 of *Lecture Notes in Computer Science*, pages 199–212. Springer, 2005.

[KK91] Nils Klarlund and Dexter Kozen. Rabin measures and their applications to fairness and automata theory. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 256–265, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

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

[LAS] The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

[Löd99] C. Löding. Optimal bounds for the transformation of omega-automata. In *Proc. of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, December 1999.

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

[McN65] Robert McNaughton. Finite-state infinite games. Technical report, Massachusetts Institute Of Technology, September 1965.

[McN66] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, 1966.

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

[Mel01] Helmut Melcher. *Hierarchische kompositionale Synthese von Steuerungen für reaktive Systeme*. PhD thesis, Uni Karlsruhe, 2001.

[MH84] S. Miyano and T. Hayashi. Alternating Finite Automata on ω-Words. *Theor. Comput. Sci.*, 32:321–330, 1984.

[Mic88] M. Michel. Complementation is more difficult with automata on infinite words. CNET, Paris, 1988.

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

[Mor82] Bernard M. E. Moret. Decision trees and diagrams. *ACM Computing Surveys*, 14(4):593–623, December 1982.

[Mos84] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In Andrzej Skowron, editor, *Proceedings of the 5th Symposium on Computation Theory*, volume 208 of *LNCS*, pages 157–168, Zaborów, Poland, December 1984. Springer.

[MS95] D. E. Muller and P. E. Schupp. Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of the Theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141(1&2):69–107, 1995.

[Mul63] David E. Muller. Infinite sequences and finite machines. In *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, Chicago, Illinois, 28–30 October 1963. IEEE.

[PP04] Dominique Perrin and Jean-Erric Pin. *Infinite Words: Automata, Semigroups, Logic and Games*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proc. of the sixteenth annual ACM Symposium on Principles of programming languages (POPL'89)*, pages 179–190, 1989.

[Pur95] Anuj Puri. *Theory of hybrid systems and discrete event systems*. PhD thesis, University of California, Berkeley, Dec., 1995.

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

[Rab72] M.O. Rabin. Automata on infinite objects and church's problem. *Amer.Math.Soc.Providence, RI*, 1972.

[Saf88] S. Safra. On the Complexity of ω-Automata. In *Proc. of the 29th Symp. on Foundations of Computer Science*, pages 319–327, Los Alamitos, CA, October 1988. IEEE Computer Society Press.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6. edition, 2001.

[Str82] R. S. Streett. Propositional dynamic logic of looping and converse is elementary decidable. *IC*, 54:121–141, 1982.

[SV00] Dominik Schmitz and Jens Vöge. Implementation of a strategy improvement algorithm for finite-state parity games. In Sheng Yu and Andrei Paun, editors, *CIAA*, volume 2088 of *Lecture Notes in Computer Science*, pages 263–271. Springer, 2000.

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

[Tho90] Wolfgang Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Formal models and semantics, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. *Lecture Notes in Computer Science*, 900:1–13, 1995.

[Tho97] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume III, pages 389–455. Springer, New York, 1997.

[VJ00] Vöge and Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV: International Conference on Computer Aided Verification*, 2000.

[Wal03] Nico Wallmeier. Symbolische Synthese zustandsbasierter reaktiver Programme. Diplomarbeit, RWTH Aachen, Januar 2003.

[WHT03] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In Oscar H. Ibarra and Zhe Dang, editors, *Proceedings of the 8th International Conference on Implementation and Application of Automata, CIAA 2003*, volume 2759 of *Lecture Notes in Computer Science*, pages 11–22. Springer, 2003.

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

[Zie05] Wieslaw Zielonka. An invitation to play. In J. Jedrzejowicz and A. Szepietowski, editors, *Proceedings of 30th International Symposium on Mathematical Foundations of Computer Science, MFCS 2005*, volume 3618 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2005.

[ZP96] Uri Zwick and Mike Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158:343–359, 1996.

Index

Symbols

A

B

U

W

Z

Lebenslauf

Zur Person

Name:	Nico Wallmeier
geboren:	25. April 1977 in Hilden
Staatsangehörigkeit:	deutsch

Bildungsgang

1983–1987	Gemeinschaftsgrundschule Jahnstraße Düsseldorf
1987–1996	Luisen-Gymnasium Düsseldorf
06/1996	Abitur
10/1997 – 01/2003	Studium der Informatik an der RWTH mit Nebenfach Elektrotechnik
01/2003	Diplom in Informatik, Titel der Arbeit: *Symbolische Synthese zustandsbasierter reaktiver Programme*
02/2003 – 01/2006	Stipendiat des Graduiertenkollegs *Software für mobile Kommunikationssysteme*
02/2006 – 07/2007	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik 7 “Logik und Theorie diskreter Systeme” (Prof. Dr. Wolfgang Thomas) Rheinisch-Westfälische Technische Hochschule Aachen