Diplomarbeit

Memory Reduction for Strategies in Infinite Games

von

Michael Holtmann

Überarbeitete Version

Lehrstuhl für Informatik 7
Logik und Theorie diskreter Systeme
Professor Dr. Wolfgang Thomas

Fakultät für Mathematik, Informatik und Naturwissenschaften
Rheinisch-Westfälische Technische Hochschule Aachen
März 2007

Name : Michael Holtmann
Geburtsdatum : 03.08.1979
Geburtsort : Düsseldorf

Straße : Karlsgraben 52-54
Postleitzahl : 52064
Stadt : Aachen
Telefon : 0241 / 5595753
E-Mail : Michael.Holtmann@rwth-aachen.de

Matrikelnummer : 229569

Betreuer: Dr. Christof Löding
Abgabefrist : 29.12.2006

Erklärung

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

Aachen, den 12.03.2007 Michael Holtmann

Erste Version: 27.12.2006
Überarbeitete Version: 12.03.2007

Abstract

In this thesis we deal with the problem of reducing the memory necessary for implementing winning strategies in infinite games. We present an algorithm that is based on the notion of game reduction. Its key idea is to reduce the problem of computing a solution to a given game to computing a solution to a new game. This new game has an extended game graph which contains the memory to solve the original game. Our algorithm computes an equivalence relation on the vertices of the extended game graph and from that deduces equivalent memory contents. The computations are carried out via a transformation to ω-automata. We apply our algorithm to Request-Response and Staiger-Wagner games where in both cases we obtain a running time exponential in the size of the given game graph. We compare our method to another technique of memory reduction, namely the minimisation algorithm for finite automata with output. For each of the two approaches we present an example for which it yields a substantially better result than the other approach.

Zusammenfassung

In dieser Arbeit behandeln wir das Problem, den für die Implementierung von Gewinnstrategien in unendlichen Spielen benötigten Speicher zu reduzieren. Dazu stellen wir einen Algorithmus vor, der auf der Methode der sogenannten Spielreduktion basiert. Die Idee der Spielreduktion ist, die Lösung eines gegebenen Spiels auf die Lösung eines neuen Spiels zu reduzieren. Die Knotenmenge des neuen Spielgraphen ist um den Speicher erweitert, der zur Lösung des Ausgangsspiels benötigt wird. Unser Algorithmus berechnet eine Äquivalenzrelation auf der Knotenmenge des neuen Spielgraphen und daraus äquivalente Speicherinhalte. Die dazu notwendigen Berechnungen werden auf der Ebene von ω-Automaten durchgeführt. Unseren Algorithmus wenden wir auf Request-Response- und Staiger-Wagner-Spiele an, wobei die Laufzeit in beiden Fällen exponentiell in der Größe des gegebenen Spielgraphen ist. Wir vergleichen unsere Methode mit einer weiteren Technik zur Speicherreduktion, nämlich dem Minimierungsalgorithmus für endliche Automaten mit Ausgabe. Für jeden der beiden Ansätze präsentieren wir ein Beispiel, bei dem er ein wesentlich besseres Ergebnis liefert als der jeweils andere Ansatz.

Acknowledgements

I am very grateful to all the people who have made their contributions to the success of this thesis both directly and indirectly.

First of all, I would like to thank my supervisor Dr. Christof Löding. The idea of transforming infinite games into ω-automata and then applying reduction algorithms to reduce the memory is due to him. The innumerable discussions and helpful suggestions have been of great value to my work.

Secondly, much gratitude goes to Nico Wallmeier for allowing me to work with his program. This eased the advancement with putting my ideas into practice.

I thank Prof. Wolfgang Thomas and Prof. Joost-Pieter Katoen for appraising my thesis.

Furthermore, I would like to thank again Nico Wallmeier as well as Birgit Kann, Frank Radmacher and Heiko Damme for their kind readiness to proofread my thesis. Their constructive remarks on the contents and wording helped me to bring out things more coherently.

Last but not least, I am deeply indebted to my friends and family who have supported me throughout the past years and without whom this would not have been possible.

Contents

Chapter 1

Introduction

1.1 Motivation

History. To identify the emergence of the theory of infinite games one has to go back to around 1960. At that time a new field of study arose, namely the theory of automata on infinite objects (ω-automata). The Swiss mathematician Richard Büchi introduced a new type of automaton (cf. [Büc62]), today known as Büchi automaton. It has the same components as a usual finite automaton, i.e. finite state set Q, input alphabet Σ, initial state q_0, transition relation Δ and a set F of final states. The important difference to a standard finite automaton is that a Büchi automaton works on infinite but not on finite words. A run of a Büchi automaton is accepting if and only if a final state is visited again and again. Büchi proved the first fundamental results for this type of automaton, e.g. equivalence to ω-regular expressions. Büchi, Rabin and Trakhtenbrot (et al.) related the theory of ω-automata to logic and computability theory. They solved decision problems both in classical mathematics and in what is known today as theoretical computer science (cf. [Rab69, TB73]). One of the most crucial insights due to their work is the decidability of the theory of the infinite binary tree by means of ω-automata (Rabin's tree theorem).

From Reactive Systems to Infinite Games. Various results in the theory of ω-automata have been used to find solutions to verification problems, e.g. the model checking problem for temporal logics (cf. [CE81, QS81]). Given a system and a specification the question is whether the system satisfies the specification. The model checker gets a model of the system and tests it against the specification which is usually given as a formula.

A different methodology to solve the verification problem are infinite games. Today they constitute a powerful tool for testing reactive systems. Reactive systems are programs like protocols, controllers and operating systems. They have two characteristic properties:

1. Reactiveness: Several components act in alternation or the program under

consideration (e.g. an operating system) communicates with its environment (e.g. a user).

2. Non-terminating behaviour: Programs like operating systems have to run indefinitely.

The framework of infinite games is based on a similar idea as ω-automata. We are given a transition system and an acceptance condition on the infinite paths through the system. Each path can be viewed as a run of an ω-automaton or as a play in an infinite game. Whereas for automata we are only interested in the set of accepting runs in infinite games we also consider the behaviour of the involved players. For this reason, in the early 1980s acceptance conditions were reformulated as winning conditions and the two concepts have been dealt with coexistently since then (cf. [GH82, McN93]).

An infinite game can have various forms. In this thesis we consider it to be a pair $\Gamma = (G, \varphi)$ made up of a (finite) game graph G and a winning condition φ. It is played by two players, called Player 0 and Player 1. G is a directed graph with a set of vertices Q partitioned into two sets where from each set exactly one of the two players is allowed to choose the next transition. A play is an infinite run through G starting from a designated vertex. It is won by Player 0 if and only if the sequence of visited vertices satisfies the winning condition φ and one player wins if and only if the other player loses. A solution to an infinite game is indicated by two things:

1. Winning region: The set of vertices from where the respective player can force to win.

2. Winning strategy: A specification of the behaviour a player has to act in accordance with in order to win.

Example 1.1. In a Büchi game we are given a designated subset F of the set of vertices in G. The winning condition for Player 0 is that this set F has to be visited infinitely often. Consider the following Büchi game with $F = \{1, 5\}$. Player 0 vertices are drawn as circles and Player 1 vertices as squares.

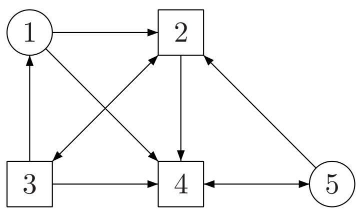


Figure 1.1: *Büchi game*

The reader may verify that Player 0 can force infinitely many visits to F from vertices $1, 4$ and 5 but not from vertices 2 and 3. Hence, his winning region is

$W_0 = \{1, 4, 5\}$. A possible winning strategy for Player 0 from a vertex in this set is to always move to vertex 4.

Winning Conditions and their Expressiveness. Besides the Büchi condition, today many different types of winning conditions (resp. acceptance conditions) are considered to model the versatile requirements demanded from reactive systems. We now give a short review of the most common ones where we hold off on most details until Chapter 2.

Muller introduced an automaton with an acceptance component $\mathcal{F}$ that consists of subsets of the state set of the considered automaton ($\mathcal{F} = \{F_1, \dots, F_k\}$). A run ρ of a Muller automaton is accepting if and only if the set of states visited infinitely often, i.e. $Inf(\rho)$, is one of the sets F_i. Muller used this condition to solve problems in designing of asynchronous circuits (cf. [Mul63, Mul64]). Büchi and Landweber showed that the problem of finding a solution to a Muller game is decidable (cf. [BL69]). Another important acceptance condition was introduced by Rabin (cf. [Rab72]). There, we are given k pairs of subsets of the state set, i.e. $\mathcal{F} = \{(E_1, F_1), \dots, (E_k, F_k)\}$ and a run ρ is accepting if and only if there exists an index $1 \leq i \leq k$ such that $Inf(\rho) \cap E_i = \emptyset$ and $Inf(\rho) \cap F_i \neq \emptyset$. A different type of acceptance condition, namely the parity condition, was introduced independently by Mostowski (cf. [Mos84]) and Emerson and Jutla (cf. [EJ91]). There, the acceptance component is a colouring c, i.e. a function assigning to each state a natural number. A run ρ of a parity automaton is accepting if and only if the highest colour seen infinitely often in ρ, i.e. $max(Inf(c(\rho)))$, is even. A parity automaton can be directly transformed into a Rabin automaton without changing the transition structure. The same can be done for a transformation from Rabin automata to Muller automata. On the other hand, to construct an equivalent parity automaton from a Rabin (or Muller) automaton one needs a number of states which is exponential in the number of states of the given automaton. Nevertheless, Büchi, Muller, Rabin and parity automata have all the same expressive power. A transformation from Muller automata to a special subclass of Rabin automata can be found in [Tho95]. Various other transformations can be found in [TL03].

One of the main theorems on infinite games is sometimes referred to as "Memoryless Determinacy" and was shown by Emerson, Jutla and Mostowski in 1991 (cf. [EJ91]). It states that parity games are determined and that both players have positional winning strategies from their winning regions. An infinite game is called determined if from each vertex exactly one of the two players can force to win. A strategy is called positional if the decision where to move from a given vertex only depends on this vertex. This is a special property of parity games which makes them more practical for some purposes. For the implementation of positional strategies no memory is necessary. If on the other hand we need to take into account finite information from the history of a play to be able to construct a winning strategy then we speak of a finite-state strategy. Finite-state strategies can be implemented by finite automata with output. In this context we call them strategy automata. The number of states of an automaton implementing a given strategy is the size of the memory used for implementing the strategy.

Aim of this Thesis. In this thesis we are dealing with the question of how to algorithmically reduce the size of the needed memory. One approach to solve this problem is the minimisation algorithm for strategy automata. Although it is very efficient it has a certain shortcoming for our purpose, namely that its result depends on the implemented strategy. We argue that this algorithm may yield an automaton of arbitrarily large size, already for very simple games. We present a new algorithm for memory reduction which tries to overcome this. It is based on the concept of game reduction. In a game reduction the problem of finding a solution to a given game is transferred to the problem of finding a solution to a new game. This methodology of simulating a winning condition by a different one was already foreshadowed above for the level of acceptance conditions. The idea of game reduction is to incorporate a finite memory into the given game graph such that at each vertex the players have enough information to take reasonable decisions where to move. The output of a game reduction algorithm is a new infinite game uniquely determined by the original one. The winning condition of the new game is easier than the given one in the sense that the new game is solvable by positional winning strategies. On the other hand the size of the new game graph is usually exponential in the size of the original game graph.

To avoid any confusion we have to address the ambiguity of the term "Reduction". If we speak of a "Game Reduction" then we mean an algorithm which transforms a given infinite game into a new one such that from a solution to the new game one can compute a solution to the original one. This idea was explained in the previous paragraph. If we simply speak of a "Reduction" then we mostly mean an algorithm that transforms a given automaton into a new equivalent one such that the new automaton has less states than the given one. The third term is "Memory Reduction". It refers to the problem of finding a solution to an infinite game which can be implemented with as little memory as possible. Hence, this notion of reduction is more general than the other two. Our new algorithm consists of four steps. The first step is a game reduction and the third step is a reduction of an automaton. Moreover, the algorithm as a whole conduces to memory reduction.

Our algorithm is applicable to any infinite game. Therefore, it is formulated and proven correctly on a general level. Though, in the core part of it we transform a given game into an ω-automaton (we call it "Game Automaton") and subsequently reduce the state space of this automaton. Hence, we are restricted to those types of acceptance conditions for which appropriate reduction algorithms do exist. We deal with two specific conditions (cf. Chapter 5). The first one is the Request-Response condition introduced in [Hüt03]. There, we are given a collection $\mathcal{F}$ of pairs of sets of vertices, i.e. $\mathcal{F} = \{(P_1, R_1), \ldots, (P_k, R_k)\}$ with $P_i, R_i \subseteq Q$. Player 0 wins a play if and only if any visit to a set P_i (the "request") is eventually followed by a visit to the corresponding R_i (the "response"). Request-Response games can be reduced to Büchi games. For the reduction of a Büchi automaton we use the notion of delayed simulation (cf. [EWS05]). The second type of winning condition dealt with in this thesis is named after the authors of [SW74], the Staiger-Wagner winning condition. As for the Muller condition we are given a set $\mathcal{F} = \{F_1, \dots, F_k\}$ with $F_i \subseteq Q$. Player 0 wins a play if and only if the set of visited vertices, i.e. $Occ(\rho)$, is exactly one of the sets F_i. Staiger-Wagner games can be reduced to the weak version of parity games. The corresponding weak parity automaton can be efficiently minimised with the algorithm presented in [Löd01].

The Results. One objective of this thesis is to relate our new approach of memory reduction to the minimisation algorithm for strategy automata. Both algorithms reduce the memory that is necessary to implement winning strategies in infinite games. The problem with this comparison is that the algorithms have different starting points. The new algorithm first reduces a game graph and then on the reduced game graph computes a solution. This approach more or less amounts to simplifying the complexity of a solution to a game or of the effort that is to be made to compute it. The minimisation algorithm for strategy automata works completely different. There, a solution to a game is already given and the algorithm reduces the memory that is necessary to implement it. We have to take this into consideration when commenting on any results about both algorithms.

We make both a practical and a theoretical analysis of our algorithm. For the practical part we have implemented both the minimisation algorithm for strategy automata and our new algorithm for the Request-Response and Staiger-Wagner winning condition. In the new algorithm the game reduction step takes exponential time and space for both winning conditions. For the reduction of the game automaton we can use in both cases the minimisation algorithm for standard DFA. However, we have to execute certain precomputations. These precomputations as well as the subsequent reduction step are computable very efficiently. The problem with respect to the computational costs is that they are performed on a transition structure which is exponentially large in the size of the game we want to solve. This is a big drawback compared to the minimisation algorithm for strategy automata. Moreover, one issue that we came across during the implementation is that considering non-reachable vertices when computing the game reduction can improve the eventual results. Doing so our algorithm is slowed down even more. Altogether, the graph returned by the game reduction should not have more than $15,000$ nodes.

We present results revealing the pros and cons of both approaches to memory reduction. We give an example where Player 0 has a positional winning strategy but the algorithm to solve the game yields a strategy automaton of exponential size. Since the strategy implemented by this automaton has a certain complexity and the minimisation algorithm for strategy automata does not allow for a simplification of the strategy it fails at reducing the automaton to polynomial size. However, our new algorithm reduces the memory even to constant size and thereby makes it possible to compute a very simple winning strategy for Player 0. We also give an example where the situation is converse. There we construct a game in which we can easily compute a positional winning strategy for Player 0. But due to a certain weakness of game automata our algorithm yields a strategy automaton of exponential size. The reason for this is that game automata are more general than infinite games and do not take into account the behaviour of the two players. Since our new approach is applied prior to solving a game and the minimisation algorithm for strategy automata is applied after a game has been solved both techniques can be integrated to one big algorithm for memory reduction. Furthermore, our results show that both algorithms have a right to exist.

1.2 Organisation of this Thesis

In the second chapter we provide the basic knowledge necessary to understand the rest of this work. We introduce the basic notions for infinite games and strategies. Thereto, we formally introduce some of the most common winning conditions and define strategy automata. A frequently used method to compute positional winning strategies is the attractor construction and we present it at the beginning of the section on solving infinite games. Afterwards we focus on those types of games that are relevant for this thesis and present detailed algorithms how to solve them. As a basic concept we introduce the method of game reduction.

In Chapter 3 two algorithms for reducing the memory necessary for implementing winning strategies in infinite games are presented. The first algorithm is the minimisation algorithm for finite automata with output. We argue why this algorithm has certain drawbacks for our purposes. To overcome them a new approach to memory reduction is presented. The idea is to view an infinite game as an ω-automaton reading infinite sequences that are plays in a given game. We call this a game automaton. We explain how reduction algorithms for game automata can be used to reduce the size of the underlying game graphs. We compute an equivalence relation on the state set and from that determine equivalent memory contents. In consequence of this our algorithm returns a strategy automaton with probably less states. At the end of this chapter the correctness of our algorithm is proven in the main theorem.

Our algorithm uses game reductions from Request-Response games to Büchi games and from Staiger-Wagner games to weak parity games. Hence, we need reduction algorithms for ω-automata for the Büchi acceptance condition and for the weak parity acceptance condition. In the fourth chapter we present such algorithms. The first one is used to reduce general Büchi automata and uses the notion of delayed simulation (cf. [EWS05]). We explain how this algorithm is integrated into our method of memory reduction. For that we have to verify a technical property of delayed simulation which we call compatibility. The second algorithm deals with a special subclass of deterministic Büchi automata, namely deterministic weak Büchi automata (DWA). For DWA there exists an efficient minimisation algorithm which utilises the minimisation algorithm for standard DFA. We show that the class of DWA is equivalent to the class of weak parity game automata and give a very simple procedure to construct a DWA from a weak parity game automaton, and vice versa. Finally, we also prove compatibility for the utilised equivalence relation (as we have done for the delayed simulation relation). This is necessary if we want to apply our main theorem from Chapter 3.

In Chapter 5 we apply our algorithm to Request-Response and Staiger-Wagner games. For that we use the game reduction algorithms from Chapter 2 and the reduction algorithms for ω-automata presented in Chapter 4. First we motivate each algorithm by an example and then give a detailed listing of the steps to be performed in both cases. For each algorithm we present the results on the considered example and thereby give a transition to the subsequent chapter.

The sixth chapter gives an overview of the different issues that we came across during the implementation. Thus, it deals with the practical aspects of this thesis. First of all, we discuss two different approaches to the computation of equivalent memory contents. The "extended" version is much more time-consuming than the "normal" version but normally yields better results, i.e. strategy automata of smaller size. In the sequel of this chapter we give a short introduction to the tool GASt and describe how we have implemented our memory reduction algorithms for Request-Response and Staiger-Wagner games and the minimisation algorithm for strategy automata.

In Chapter 7 we compare our new approach to memory reduction with the minimisation algorithm for strategy automata. In the core part of it we prove that there exist infinite games where our algorithm yields winning strategies that can be implemented with a very small memory, but, without applying it we would obtain winning strategies that need a very large memory to be implemented. We also present an example where our algorithm fails at reducing the memory even though the game under consideration is very simple.

Chapter 8 summarises our results and gives some ideas about possibilities how to further improve our algorithm.

Chapter 2

Preliminaries

In this chapter we introduce some basic notions necessary to follow the rest of this work. We define both infinite games and strategies. Furthermore, we present detailed algorithms for solving those types of games which are of greater importance for us.

2.1 Games and Strategies

2.1.1 Infinite Games

An infinite game is played by two players. We denote them by Player 0 and Player 1. The game arena is a graph G where from each vertex exactly one of the two players is allowed to move. A play goes until infinity and its winner is decided by a winning condition φ.

Definition 2.1. An *infinite game* is a tuple $\Gamma = (G, \varphi)$. $G = (Q, E)$ is the game graph where Q is a disjoint union $Q_0 \dot{\cup} Q_1$ of Player 0 and Player 1 vertices and we have the edge relation $E \subseteq Q \times Q$. A *play* is a sequence $\rho = q_0 q_1 q_2 \ldots \in Q^\omega$ of vertices of G such that $(q_i, q_{i+1}) \in E$ for every $i \in \mathbb{N}$. To guarantee infinite sequences through G we assume that every vertex has at least one successor. Player 0 wins a play ρ if and only if the winning condition φ is satisfied. We denote the set of plays winning for Player 0 by Win_0 (Win_1 analogously for Player 1) and the set of all plays of Γ by $\Psi(\Gamma)$.

All common winning conditions can be roughly classified in three different categories each one of them having its own characteristics. Only some of them are relevant for us. The first type is a condition on the occurence set $Occ(\rho)$, i.e. the set of vertices visited at least once. The second type is a condition on the set of vertices which are visited infinitely often in a play ρ. This set is called the infinity set and is denoted by $Inf(\rho)$. The third condition is somehow different. It involves a colouring c which assigns to each vertex a natural number. In the following we give examples for all three types of winning conditions:

1. The simplest one is the guaranty winning condition. There, we are given a designated set $F \subseteq Q$ of final vertices and Player 0 wins a play ρ if and only if the set F is visited at least once, i.e. $Occ(\rho) \cap F \neq \emptyset$. The Staiger-Wagner winning condition is a disjunction of strict[1] guaranty winning conditions. We are given a family $\mathcal{F} = \{F_1, \dots, F_k\}$ of final sets ($F_i \subseteq Q$) and Player 0 wins a play ρ if and only if $Occ(\rho) \in \mathcal{F}$, i.e. one of the sets F_i must be hit exactly. An example of a Staiger-Wagner game can be seen in Figure 3.5.

2. Büchi games and Muller games are to $Inf(\rho)$ as Guaranty games and Staiger-Wagner games are to $Occ(\rho)$. In Büchi games we are given a set $F \subseteq Q$ and Player 0 wins a play ρ if and only if the set F is visited infinitely often, i.e. $Inf(\rho) \cap F \neq \emptyset$. An example of a Büchi game can be found on page 4 in the introduction. The Muller winning condition is a disjunction of strict Büchi conditions. We are given a family $\mathcal{F} = \{F_1, \dots, F_k\}$ of final sets ($F_i \subseteq Q$) and Player 0 wins a play ρ if and only if the infinity set coincides with one of the sets F_i, i.e. $Inf(\rho) \in \mathcal{F}$.

3. For the third type of winning condition we do not consider sets of states but a property on the individual states. Therefore, we are given a colouring $c : Q \to \{0, \dots, k\}$ where k is a natural number. For a run $\rho = \rho(0)\rho(1)\rho(2)\dots$ we define the sequence of colours of ρ as $c(\rho) = c(\rho(0))c(\rho(1))c(\rho(2))\dots$. We introduce the parity winning condition in the strong and in the weak version. For Player 0 to win a play ρ with the strong parity condition we require that the highest colour seen infinitely often is even, i.e. $\max(Inf(c(\rho)))$ is even. For the weak version we require this condition only for the occurrence set but not for the infinity set: $\max(Occ(c(\rho)))$ must be even.

One class of winning condition which is of special interest for us is between categories one and two. For this reason we introduce it separately from all others. It is called the Request-Response winning condition (cf. [Hüt03]). There, we are given a family $\mathcal{F} = \{(P_1, R_1), \dots, (P_k, R_k)\}$ of pairs of sets $P_i, R_i \subseteq Q$. Player 0 wins a play ρ if and only if every visit to some P_i is eventually followed by a visit to R_i (for $i = 1, \dots, k$). The visit to P_i is called the request and the visit to R_i is called the response:

$$\text{Player 0 wins } \rho :\Longleftrightarrow \forall r (\rho(r) \in P_i \Longrightarrow \exists s \geq r : \rho(s) \in R_i)$$

An example of a Request-Response game can be found on page 71.

2.1.2 Strategies and Strategy Automata

Solving games means to compute the winning regions of the two players as well as winning strategies from these winning regions. By the winning region of Player 0 we mean a subset W_0 of the set of vertices Q of the game graph from where Player 0 has a winning strategy (analogously for Player 1). By a strategy for Player 0 we

[1]In this context "strict" means that two sets are exactly the same but not just overlap.

mean a function f assigning to each game position of Player 0 an edge to choose from this game position.

Definition 2.2. A *strategy* for Player 0 is a partial function $f : Q^*Q_0 \dashrightarrow Q$ that assigns to any play prefix $q_0 \dots q_i$ with $q_i \in Q_0$ a state q_{i+1} such that $(q_i, q_{i+1}) \in E$. The play $\rho = q_0q_1q_2\dots$ is *played according to the strategy* f if for every $q_i \in Q_0$ it holds $q_{i+1} = f(q_0 \dots q_i)$. The function f is called a *winning strategy from* q for Player 0 if any play ρ starting in vertex q that is played according to f is won by Player 0. The *winning region* W_0 of Player 0 is the set of all vertices from where Player 0 has a winning strategy. Winning strategies and winning regions for Player 1 can be defined analogously.

Remark 2.3. Not both players can have a winning strategy from the same vertex, i.e. $W_0 \cap W_1 = \emptyset$ (and $\mathrm{Win}_0 \cap \mathrm{Win}_1 = \emptyset$).

Proof. Assume $W_0 \cap W_1 \neq \emptyset$. Then there exists $q \in Q$ such that each of the two players has a winning strategy from q, say f for Player 0 and g for Player 1. Assume both players play according to their winning strategy building up the play ρ. Since f is winning for Player 0 ρ satisfies φ. Analogously, since g is winning for Player 1 ρ satisfies $\neg\varphi$. Contradiction. □

This raises the question whether each vertex of a game graph belongs to one or the other player's winning region, i.e. whether $W_0 \dot\cup W_1 = Q$ holds. If a game has this property then we say that it is *determined*. In a determined game Γ we can partition the set $\Psi(\Gamma)$ of all possible plays into Win_0 and Win_1, i.e. $\Psi(\Gamma) = \mathrm{Win}_0 \dot\cup \mathrm{Win}_1$. All games presented in Section 2.1.1 are determined. For detailed proofs of the determinacy of various games see [TL03]. We address the problem of computing winning strategies for these games in Section 2.2. There are some exotic games played on infinite game graphs which are non-determined. An example is the Gale-Stewart game which can also be found in [TL03].

The main issue of this thesis is to find winning strategies for specific games which need as little memory as possible to be implemented. As introduced above there is a variety of winning conditions for infinite games. Some winning conditions are more complex than others in the sense that it is more costly to express them in a specific formalism. A natural requirement for winning strategies (or strategies in general) should be that they are computable. In [TL03] there is an interesting example of a game on an infinite game graph where Player 0 has a winning strategy which is not computable. The proof is accomplished by a reduction from the halting problem for Turing machines. All games that are of interest for us have computable winning strategies. We differentiate strategies by the amount of memory needed for implementing them. A strategy f for Player 0 is called positional (cf. Definition 2.4) if the choice for the next transition depends only on the current vertex. Such a strategy can be simplified to $f : Q_0 \to Q$ or likewise be expressed by a subset $E_f \subseteq E$ containing for each $q \in Q_0$ one edge (q, p) with $f(q) = p$. Some winning conditions require to memorise particular parts of the history of a play to take an appropriate decision. These are called *finite-state* strategies and are implemented by finite automata with output.

Definition 2.4. A *strategy automaton* for Player 0 has the form $\mathcal{A} = (S, Q, s_0, \sigma, \tau)$ with

$$\begin{aligned} S &: \text{finite set of states} \\ Q &: \text{input alphabet} \\ s_0 &: \text{initial state} \\ \sigma : S \times Q \to S &: \text{memory update function} \\ \tau : S \times Q_0 \to Q &: \text{transition choice function (for Player 0)} \end{aligned}$$

The strategy automaton $\mathcal{A}$ defines a strategy $f_\mathcal{A}$ for Player 0. Memory updates (determined by σ) are made for moves of both players whereas the output function τ is defined only for vertices of Player 0. Extending σ from reading states to reading finite sequences of states (play prefixes) we obtain $\sigma^* : S \times Q^* \to S$ with $\sigma^*(s, \epsilon) = s$ and $\sigma^*(s, wq) = \sigma(\sigma^*(s, w), q)$. From this we obtain the strategy $f_\mathcal{A}$ computed by $\mathcal{A}$ as $f_\mathcal{A}(q_0 \dots q_i) := \tau(\sigma^*(s_0, q_0 \dots q_{i-1}), q_i)$ for $q_i \in Q_0$. A strategy f is called an *automaton strategy* if there exists a strategy automaton $\mathcal{A}$ with $f = f_\mathcal{A}$.[2] f is called *positional* if and only if it can be implemented by a strategy automaton with only one state. By the *size* of a strategy we mean the minimal number of states among all automata implementing this strategy.

It is important to note that for a strategy f there can exist several automata which have all as many states as the size of f. If we are given a strategy automaton $\mathcal{A}_f$ implementing the strategy f and redirect some of its transitions to new target states then the resulting automaton outputs a different function than before. But this does not need to have any effects on the strategy f. This is due to the fact that f is normally a partial function and not all transitions in $\mathcal{A}_f$ are relevant for implementing it. In Section 3.1 we show that this has a certain consequence for the minimisation of strategy automata.

Before we give solutions to some basic game types we introduce a fundamental notion used to solve Büchi and weak parity games.

2.1.3 Attractor Strategies

A *Guaranty game* has a designated set F and a play is winning for Player 0 if and only if Player 0 can force a visit to F at least once. Thus, we are interested in a set of vertices from where Player 0 has it in his hands to move into F no matter what Player 1 does. We call this set the *0-Attractor* of F and denote it by $Attr_0(F)$. The subscript 0 indicates that the attractor refers to Player 0. In guaranty games the 0-Attractor of F is exactly the winning region of Player 0 while in Büchi games the situation is a bit more difficult as we show later. We can compute the 0-Attractor inductively. For that we introduce a superscript i denoting that a visit to F is possible in at most i moves. At the induction start we have

[2]To obtain the equality $f = f_\mathcal{A}$ it might be necessary to extend the domain of f to Q^+.

$$Attr_0^0(F) = F$$

and for $i \in \mathbb{N}$

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

We have $Attr_0^0(F) \subseteq Attr_0^1(F) \subseteq Attr_0^2(F) \subseteq \dots$ and after at most $|Q|$ steps, i.e. for $i = |Q|$, the sequence becomes stationary. We set

$$Attr_0(F) := \bigcup_{i=0}^{|Q|} Attr_0^i(F).$$

Now we can define certain *attractor strategies* for the two players (independent of the type of a given game). If we have a node q in $Attr_0(F) \setminus F$ then by definition of $Attr_0(F)$ Player 0 can reduce the distance to F. By that we mean that Player 0 from q can force a visit to a vertex q' such that $\min\{i \mid q' \in Attr_0^i(F)\} < \min\{i \mid q \in Attr_0^i(F)\}$. Either $q \in Q_0$ which means that Player 0 can choose an appropriate edge himself or $q \in Q_1$ and all edges from q lead "towards" F. Conversely, if $q \in Q_0$ and $q \in Q \setminus Attr_0(F)$ then Player 0 cannot choose an edge into $Attr_0(F)$ (otherwise q would belong to $Attr_0(F)$ by definition). If $q \in Q_1$ then there is at least one edge from q leading back to $Q \setminus Attr_0(F)$. This means that Player 1 can avoid visiting $Attr_0(F)$ (or keep the distance to it "at infinity"). This behaviour, reducing the distance to F and avoiding $Attr_0(F)$, is called the attractor strategies for the two players:

Player 0 : From a vertex in $Attr_0^{i+1}(F) \setminus Attr_0^i(F)$ move to $Attr_0^i(F)$
Player 1 : From a vertex in $Q \setminus Attr_0(F)$ move back to $Q \setminus Attr_0(F)$

We can give an efficient algorithm for computing the 0-Attractor $Attr_0(M)$ for any set $M \subseteq Q$.

Remark 2.5. Let $G = (Q, E)$ be a game graph and $M \subseteq Q$. Then $Attr_0(M)$ can be computed in linear time in the size of G.

Proof. The following backward breadth-first search computes $Attr_0(M)$ (the set of marked vertices) in time $O(|Q| + |E|)$:

Algorithm 2.6. (Attractor Algorithm)
Input: Game graph $G = (Q, E)$ and a set $M \subseteq Q$
Output: $Attr_0(M)$

```
1. For any $q \in Q_1$: $n(q) :=$ outdegree of $q$
2. Perform backward breadth-first search starting from $M$ as follows:
    (a) Mark all $q \in M$
    (b) Mark $q \in Q_0$ if there exists $(q,r) \in E$ with $r$ already marked
    (c) At visit of $q \in Q_1$ from marked vertex: $n(q) := n(q) - 1$
    (d) At visit of $q \in Q_1$: mark $q$ if $n(q) = 0$
```

□

2.2 Solving Games of different Complexity

In this section we present various game types which are of our interest. Later we describe techniques to reduce the memory requirements for winning strategies for some of these games.

2.2.1 Büchi Games

A *Büchi game* $\Gamma = (G, \varphi)$ is played on a game graph $G = (Q, E)$ and has the following winning condition φ for Player 0:

$$\rho \in \mathrm{Win}_0 :\Longleftrightarrow Inf(\rho) \cap F \neq \emptyset$$

By definition of a Büchi game the winning region W_0 for Player 0 is the set of vertices from where Player 0 can force infinitely many visits to F. The computation of this set is done in two steps. First we compute all $q \in F$ from where Player 0 can force infinitely many (re)visits to F. We call this set the *recurrence region* for Player 0. We compute it by induction on the number i of revisits. We denote the subset of vertices of F from where Player 0 can force at least i revists to F by $Recur_0^i(F)$ and the subset of vertices of F from where Player 0 can force infinitely many (re)visits to F by $Recur_0(F)$. Clearly, this yields:

$$F \supseteq Recur_0^1(F) \supseteq Recur_0^2(F) \supseteq \ldots$$

and

$$Recur_0(F) = \bigcap_{i \in \mathbb{N}} Recur_0^i(F)$$

Before we can compute $Recur_0(F)$ inductively we introduce the *proper attractor* $Attr_0^+(F)$. It is a slight modification of the attractor introduced in Section 2.1.3 and denotes the set of vertices from where Player 0 can force a visit to F in at least one step. Since this is not possible in zero steps we set

$$Attr_0^0(F) := \emptyset$$

and define for $i \in \mathbb{N}$

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

This yields the proper attractor:

$$Attr_0^+(F) = \bigcup_{i \in \mathbb{N}} Attr_0^i(F)$$

In the second step we use the former result to define the recurrence region:

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

With these definitions we can now prove that Büchi games are solvable. We can compute the winning regions W_0 and W_1 for the two players and respective positional winning strategies. In the proof we show that Player 0 can use his attractor strategy to win. For the strategy of Player 1 see the remarks after Theorem 2.7.

Theorem 2.7. *Büchi games are determined. The winning regions for the two players are $W_0 = Attr_0(Recur_0(F))$ and $W_1 = Q \setminus Attr_0(Recur_0(F))$. Both players have positional winning strategies on their respective winning regions.*

Proof. By definition, on $Recur_0(F) \cap Q_0$ Player 0 can choose an edge back to $Attr_0^+(Recur_0(F))$ and from vertices of $Recur_0(F) \cap Q_1$ all edges lead back to $Attr_0^+(Recur_0(F))$. Furthermore, any vertex contained in either $Recur_0(F) \cap Q_0$ or $Recur_0(F) \cap Q_1$ is also contained in $Attr_0^+(Recur_0(F))$ itself. Hence, playing the attractor strategy from $Recur_0(F)$ Player 0 can force the play to return to $Recur_0(F)$ infinitely often. Since $Recur_0(F) \subseteq F$ by definition F is visited infinitely often. This yields:

$$Attr_0(Recur_0(F)) \subseteq W_0$$

Now we show the set inclusion in the other direction. Since we already know that $W_0 \cap W_1 = \emptyset$ it suffices to show

$$Q \setminus Attr_0(Recur_0(F)) \subseteq W_1$$

which then also verifies our assertion that Büchi games are determined. We show that Player 1 from $Q \setminus Attr_0(Recur_0^i(F))$ can force F to be visited no more than i times, i.e. finitely often. We complete the proof by induction on i. Take $q \in Q \setminus Attr_0(Recur_0^i(F))$. If $i = 0$ then $q \in Q \setminus Attr_0(F)$. With the attractor strategy Player 1 can avoid a visit to $Attr_0(F)$ and thus to F (F is visited at most 0 times). For the induction step assume $q \in Q \setminus Attr_0(Recur_0^{i+1}(F))$. If $q \in Q \setminus Attr_0(Recur_0^i(F))$ then, by induction hypothesis, Player 1 can force F to be visited at most $i \leq i + 1$ times. Otherwise $q \in (Q \setminus Attr_0(Recur_0^{i+1}(F))) \setminus (Q \setminus Attr_0(Recur_0^i(F))) = Attr_0(Recur_0^i(F)) \setminus Attr_0(Recur_0^{i+1}(F))$. But if a play does not visit $Attr_0(Recur_0^{i+1}(F))$ then Player 0 cannot force a visit to $Recur_0^{i+1}(F)$ either. This means that if a play reaches the set F then it will be visiting F outside $Recur_0^{i+1}(F)$. By induction hypothesis this implies that Player 1 can force that F is visited at most $i + 1$ times, namely for a first visit and at most i revisits. After that he can avoid any further visit to F. □

Note that the above definitions are constructive. By this we mean that we can compute the proper attractor as well as the recurrence region. We have argued that Player 0 from W_0 wins by playing his attractor strategy with respect to $Recur_0(F)$. This means, if Player 0 from W_0 continuously decreases the distance to $Recur_0(F)$ then he wins. However, the attractor strategy for Player 1 as introduced in Section 2.1.3 is not a winning strategy for Player 1. This is due to the fact that there can be final vertices in $Q \setminus Attr_0(Recur_0(F))$. Hence, avoiding $Attr_0(Recur_0(F))$ is not sufficient for Player 1 to win. To compute a winning strategy for Player 1 in a given

Büchi game we have to solve a safety game (cf. Example 2.8) for Player 1 for the set $Q \setminus F$ first. This means Player 1 wins if and only if the set $Q \setminus F$ is never left. Let W_1' be the winning region of Player 1 in this safety game. Then $Attr_1(W_1')$ is the winning region W_1 of Player 1 in the given Büchi game for which he has the following positional winning strategy. On $Attr_1(W_1') \setminus W_1'$ he plays the attractor strategy to reach W_1' and on W_1' he plays the positional winning strategy from the safety game.

2.2.2 Weak Parity Games

For weak parity games we introduce a colouring $c : Q \to \{0, \dots, k\}$ for $k \in \mathbb{N}$. A play ρ is won by Player 0 if and only if the highest colour seen in ρ is even:

$$\rho \in \mathrm{Win}_0 :\Longleftrightarrow \max(Occ(c(\rho(0))c(\rho(1))c(\rho(2))\dots)) \text{ is even}$$

Example 2.8. As examples of a weak parity game we present two special versions of it. The first one is a *Guaranty game*. There, Player 0 wins a play if and only if a certain set of states is eventually visited. Let F denote this set of "good" states. Then we get:

$$\rho \in \mathrm{Win}_0 :\Longleftrightarrow \exists i : \rho(i) \in F$$

We can equivalently formulate this as a weak parity winning condition. For that we define a colouring c_1 as follows:

$$c_1(q) := \begin{cases} 2 & \text{, if } q \in F \\ 1 & \text{, if } q \notin F \end{cases}$$

A dual to that is a *safety game*. There we require that something "bad" never happens. If we formulate it from the reverse point of view we get the following winning condition for Player 0:

$$\rho \in \mathrm{Win}_0 :\Longleftrightarrow \forall i : \rho(i) \in F$$

In a weak parity game the following colouring c_2 captures the above winning condition for safety games:

$$c_2(q) := \begin{cases} 0 & \text{, if } q \in F \\ 1 & \text{, if } q \notin F \end{cases}$$

The weak parity winning condition is simple in the sense that the winner of a play ρ is determined after a finite prefix of ρ (even if this prefix can be arbitrarily long). This is due to the fact that only the occurrence set $Occ(c(\rho))$ but not the infinity set $Inf(c(\rho))$[3] determines the winner. Both types of games are solvable by positional winning strategies for both players. Here, we consider only weak parity games.

Theorem 2.9. *Weak parity games are determined. The winning regions W_0 and W_1 can be computed and both players have positional winning strategies on their respective winning regions.*

[3]$Inf(c(\rho))$ is considered in strong parity games.

Proof. We are given the game graph $G = (Q, E)$ and the colouring $c : Q \to \{0, \dots, k\}$. We assume without loss of generality (w.l.o.g.) that k is even. Otherwise we swap the role of the two players. We define C_i to be the set of vertices with colour i:

$$C_i := \{q \mid q \in Q, c(q) = i\}$$

We calculate sets A_i for i decreasing from k down to 0. For even (resp. odd) i the set A_i denotes the set of vertices from where Player 0 (resp. Player 1) can force exactly one of the following:

1. For **even** i: Player 0 can force a visit to C_i and Player 0 cannot force that an even colour higher than i is the highest colour seen in the play and Player 1 cannot force the latter for an odd colour higher than i.
   For **odd** i: Player 1 can force a visit to C_i and Player 1 cannot force that an odd colour higher than i is the highest colour seen in the play and Player 0 cannot force the latter for an even colour higher than i.

2. At least one of the sets $A_{i+2}, A_{i+4}, \dots, A_k$ is visited.

Item 2. means that the opponent allows to visit a number even higher than i without changing the winner of the play. If e.g. a play is in A_{10} then it is up to Player 1 whether the play proceeds to C_{10} and no higher colour is seen afterwards (item 1.) or whether Player 1 even allows to visit one of the sets $A_{12}, A_{14}, \dots$ (item 2.). Note that for $i = k$ only item 1. is possible. The reader may verify that the definition underneath covers the same matter. Because we start from index k and compute $A_k, \dots, A_0$ in this order we can capture the above condition by simply computing the appropriate attractor[4] sets. When we consider A_i we only have to exclude all vertices which are covered by any A_j with an index j higher than i. Figure 2.1 shows a visualisation of the computation of the sets $A_k, \dots, A_0$. There, G_i denotes the subgraph of G where the sets $A_{i+1}, \dots, A_k$ and all adjacent edges are deleted.[5] Note that each $q \in Q$ is contained in exactly one A_i, i.e. $\bigcup_{i=0}^{k} A_i = Q$ and $A_i \cap A_j = \emptyset$ for all $i \neq j \in \{0, \dots, k\}$.

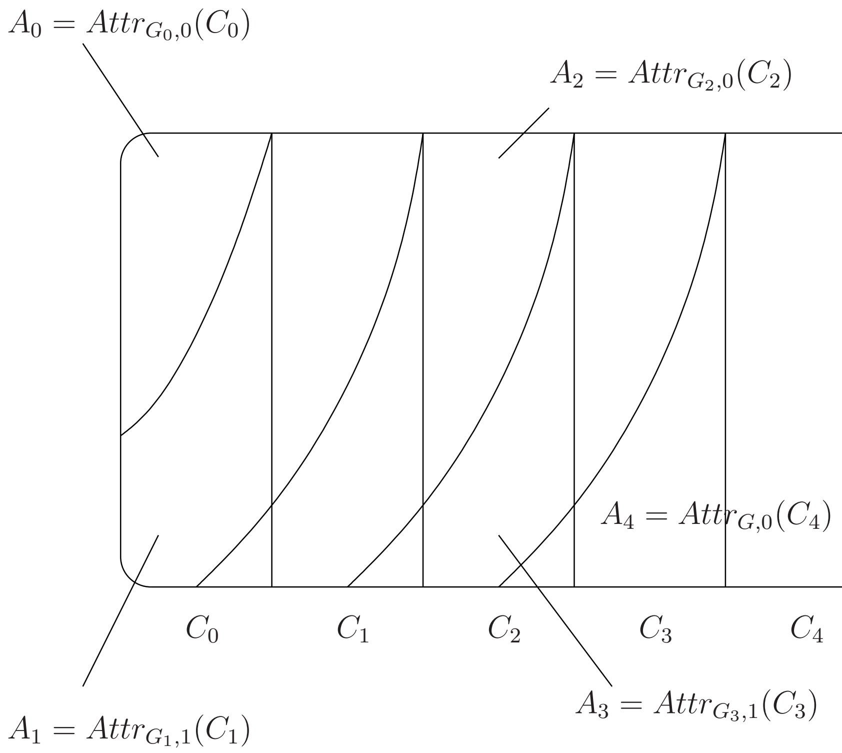
The computation of the attractor sets can be done as described in Section 2.1.3. For decreasing values of $i = k, \dots, 0$ we define:

$$A_k := Attr_{G,0}(C_k)$$
$$A_i := \begin{cases} Attr_{G_i,0}(C_i) & \text{, if } i \text{ is even} \\ Attr_{G_i,1}(C_i) & \text{, if } i \text{ is odd} \end{cases}$$

We now show that the union of the A_i for even (resp. odd) i is exactly the winning region of Player 0 (resp. Player 1). Furthermore, we give a description of the positional winning strategies for the two players.

[4] not the proper attractor

[5] $Attr_{G_i,0}$ is defined analogously to $Attr_{G,0}$.



$A_0 = Attr_{G_0,0}(C_0)$
$A_2 = Attr_{G_2,0}(C_2)$
$A_4 = Attr_{G,0}(C_4)$
C_0
C_1
C_2
C_3
C_4
$A_3 = Attr_{G_3,1}(C_3)$
$A_1 = Attr_{G_1,1}(C_1)$

Figure 2.1: *Computation of $A_k, \ldots, A_0$ for $k = 4$*

Lemma 2.10. *If $A_k, \ldots, A_0$ are computed as described above then we get:*

$$W_0 = \bigcup_{i \text{ even}} A_i \text{ and } W_1 = \bigcup_{i \text{ odd}} A_i$$

Both players have positional winning strategies on their respective winning regions.

Proof. We accomplish the proof by induction on the highest colour k. If $k = 0$ then all vertices have colour zero and we have $W_0 = \bigcup_{i \text{ even}} A_i = A_0 = Q$ and $W_1 = \bigcup_{i \text{ odd}} A_i = \emptyset$. Any positional strategy is winning for Player 0 from W_0. Now, let $k > 0$ be even. For odd k the proof is analogous with the roles of the players interchanged. It is clear that Player 0 wins from $A_k = Attr_{G,0}(C_k)$ because Player 0 from A_k can force a visit to a vertex with the highest colour k which is even. This can be done with a positional (attractor) strategy f_k and we get $A_k \subseteq W_0$. Let G_{k-1} be the game graph obtained from G by deleting A_k and all edges from or to vertices in A_k. Since in G_{k-1} only the colours $0, \ldots, k-1$ appear we can apply the induction hypothesis to it. Thus, we obtain winning regions $W_0' = \bigcup_{i \text{ even}} A_i$ for Player 0 and $W_1' = \bigcup_{i \text{ odd}} A_i$ for Player 1 where i ranges over $0, \ldots, k-1$. Additionally, we have positional winning strategies f'_0 for Player 0 on W'_0 and f'_1 for Player 1 on W'_1. We show that

$$W_0 = W'_0 \cup A_k \text{ and } W_1 = W'_1.$$

In the induction step we construct positional winning strategies f_0 (resp. f_1) on W_0 (resp. W_1) for Player 0 (resp. Player 1). The winning strategy f_0 is constructed as follows: On G_{k-1} Player 0 plays according to f'_0 and on A_k according to the positional winning strategy f_k. If a play remains in W'_0 on G_{k-1} then Player 0 wins because f'_0 is a winning strategy in the subgame from there. If the play enters A_k then Player 0 can force a visit to C_k by playing the positional winning strategy f_k. So Player 0 wins the game from $W_0 = W'_0 \cup A_k$ with a positional winning strategy. If a play starts in W'_1 then Player 1 wins by playing $f_1 := f'_1$. This is due to the fact that f'_1 is a winning strategy for Player 1 on W'_1 in the subgame and the play never reaches A_k. Otherwise there was a vertex q in the subgame from where Player 0 can force a visit to A_k. But then q would have been contained in A_k itself and would not be contained in G_{k-1} because A_k was deleted from G to obtain G_{k-1}. So, any play starting in W'_1 is won by Player 1 and we obtain $W_1 = W'_1$. Furthermore, f_1 is positional because it was copied from the positional strategy f'_1 of the subgame. □

As we have just proven both players have positional winning strategies on their respective winning regions. Furthermore, any $q \in Q$ is contained in exactly one A_i. This yields the determinacy of weak parity games and we are done with the proof of Theorem 2.9. □

In this and in the preceding section we have solved two types of games where both players have positional winning strategies on their winning regions. Now we draw our attention to two types of games where this is not possible. For Request-Response games as well as for Staiger-Wagner games one has to use memory to be able to implement a winning strategy. However, it is possible to solve these games by finite-state strategies. To do so we first transform a given game into a new game and then solve this new game by positional winning strategies. From the latter we can construct a solution to the original game. As a drawback in the worst case the new game graph is exponentially larger than the original game graph.

2.2.3 Game Reduction

In order to solve Request-Response games and Staiger-Wagner games we introduce the notion of game reduction. Reducing a game $\Gamma = (G, \varphi)$ means to compute a new game $\Gamma' = (G', \varphi')$ from Γ where G' is composed of finitely many copies of G each memorising a certain history of a play ρ on G. This memory component is used to construct a strategy automaton implementing a non-positional winning strategy for Player 0 from W_0 in Γ. If we are given two games Γ and Γ' such that Γ' is the output of a game reduction algorithm with Γ as input then we say that Γ' is the *reduced game* of Γ.

Definition 2.11. Let Γ and Γ' be games with game graphs $G = (Q, E)$ and $G' = (Q', E')$ and winning conditions φ and φ'. (G, φ) is *reducible* to (G', φ') (short: $(G, \varphi) \leq (G', \varphi')$) if the following hold:

1. $Q' = S \times Q$ for a finite set S and $q \in Q_i \iff (s, q) \in Q'_i$
2. Every play ρ of (G, φ) is transformed into a play ρ' of (G', φ') such that
   (a) $\exists s_0 \in S, \forall q \in Q$: $\rho(0) = q \implies \rho'(0) = (s_0, q)$
   (b) Let $(s, q) \in Q'$:
      i. $(q, q') \in E \implies \exists (s', q') \in Q' : ((s, q), (s', q')) \in E'$
      ii. $((s, q), (s_1, q_1)) \in E', ((s, q), (s_2, q_2)) \in E' \implies s_1 = s_2$
   (c) $((s, q), (s', q')) \in E' \implies (q, q') \in E$
3. ρ is winning for Player 0 in $(G, \varphi) \iff \rho'$ is winning for Player 0 in (G', φ')

For our definition of game reduction we use a model different to that in [TL03]. The main point is item 2.(b).ii. It means that from a given vertex (s, q) the memory content reached via any of the outgoing edges is uniquely determined by s and q. In other words, the memory update only depends on the source vertex of an edge. This is also why we need to have the unique initial memory content s_0. For technical reasons related to the algorithm we are going to present in Section 3.2 we have chosen this definition. It is equivalent to the definition from [TL03][6] in the sense that one can be transformed into the other. We show that with a given game reduction we can solve even more difficult games.

Theorem 2.12. *Let $\Gamma = (G, \varphi)$ and $\Gamma' = (G', \varphi')$ be games and $(G, \varphi) \leq (G,' \varphi')$ according to the definition above. If Player 0 wins (G', φ') from vertex (s_0, q) with a positional winning strategy f' then Player 0 wins (G, φ) from vertex q by an automaton strategy f.*

Proof. We construct the strategy automaton $\mathcal{A}_f = (S, Q, s_0, \sigma, \tau)$ implementing the winning strategy f for Player 0 in Γ. S and Q are given by Q'. s_0 is the unique initial memory content from item 2.(a) of Definition 2.11.

$\sigma : S \times Q \to S$: $\sigma(s, q) := s'$ where s' is the unique memory content reached from (s, q) according to item 2.(b).ii of Definition 2.11

$\tau : S \times Q_0 \to Q$: $\tau(s, q) := q'$ where q' is the second component of the vertex (s', q') reached from (s, q) in G' by applying the positional winning strategy f'

$\mathcal{A}_f$ builds up a play ρ on G. ρ is constructed from input letters $q \in Q$ of $\mathcal{A}_f$ which together with the memory contents induce the corresponding play ρ' on G'. Since

[6]In [TL03] the memory update also depends on the Q-component of the target vertex of an edge but not on the Q-component of its source vertex.

we have a game reduction from (G, φ) to (G', φ') and ρ' is won by Player 0 by the positional winning strategy f' it follows from 3. that f is winning for Player 0 in (G, φ). □

Theorem 2.12 reveals how we can obtain a solution to Γ from a solution to Γ'. The winning strategy for Player 0 is given by the strategy automaton $\mathcal{A}_f$. A winning strategy for Player 1 can be found in an analogous way. The winning regions W_0 and W_1 can be directly read off the solution to Γ'. Vertex q belongs to W_0 if and only if vertex (s_0, q) belongs to W'_0.

2.2.4 Request-Response Games

In this section we approach the problem of solving Request-Response games. There, we are given a set $\mathcal{F} = \{(P_1, R_1), \ldots, (P_k, R_k)\}$ of k pairs of subsets of the state set. A play ρ is winning for Player 0 if and only if any visit to a set P_i is eventually followed by a visit to R_i. The presented results (Theorems 2.13 and 2.14) are taken from [WHT03]. First we show a lower bound for the memory necessary to implement winning strategies for Request-Response games. To win such a game Player 0 needs a memory of a size exponential in the number of request-response conditions. After that we give a game reduction algorithm from Request-Response games to Büchi games. A solution to Büchi games has been presented in Section 2.2.1 and together with the game reduction it can be used to solve any Request-Response game.

Theorem 2.13. *There is a family G_k of game graphs with request-response winning condition and a special vertex v_0 such that the following holds:*

1. *The number of vertices of G_k is linear in k.*

2. *The number of request-response conditions is linear in k.*

3. *Player 0 has a winning strategy from v_0, i.e. $v_0 \in W_0$.*

4. *Every strategy automaton implementing a winning strategy for Player 0 from v_0 has at least $k \cdot 2^k$ states.*

Proof. The game graph G_k can be seen in Figure 2.2. It has $7k + 2 \in O(k)$ vertices. We thereby validate the first item from Theorem 2.13. We have $3k \in O(k)$ request-response conditions (cf. item 2.) being devided in three kinds of pairs, namely $(\underline{P_i}, \underline{R_i})$, $(\overline{P_i}, \overline{R_i})$ and (P_i, R_i) for $i = 1, \ldots, k$. For every vertex $v \in Q$ the sets where v is contained in are written next to v. R_i^* is an abbreviation for the conjunction of $\underline{R_j}$ and $\overline{R_j}$ for $j \neq i$ (e.g. $s_1 \in \underline{R_2}, \ldots, s_1 \in \underline{R_k}$ and $s_1 \in \overline{R_2}, \ldots, s_1 \in \overline{R_k}$). The pairs $(\underline{P_i}, \underline{R_i})$ are requested in vertices $\underline{v_i}$ and responded to in vertices $\overline{v_i}$ and $\underline{z_i}$ and additionally in the vertices s_j for all indices $j \neq i$. Analogously, the pairs $(\overline{\overline{P_i}}, \overline{R_i})$ are requested in vertices $\overline{v_i}$ and responded to in vertices $\underline{v_i}$ and $\overline{z_i}$ and additionally in the vertices s_j for all indices $j \neq i$. The third kind of pairs is (P_i, R_i) which is requested in vertex v_{i-1}. Additionally, (P_1, R_1) is requested in vertex v_k.

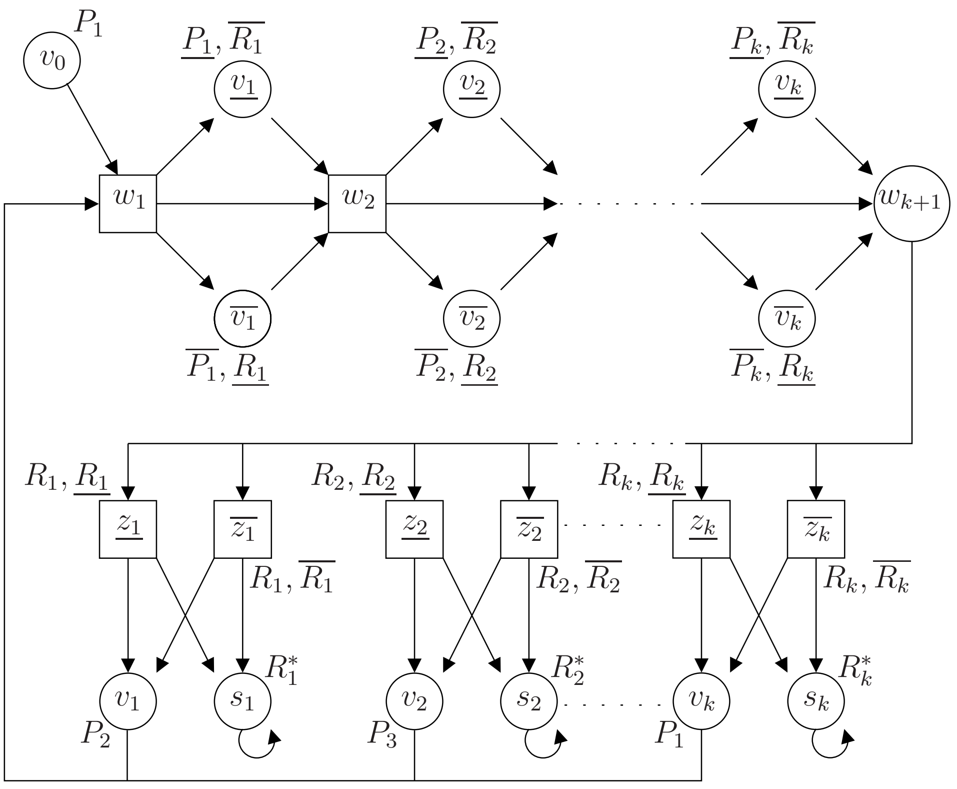


Figure 2.2: *Game Graph G_k*

We shall show that the vertex v_0 is in the winning region of Player 0 and that every strategy automaton implementing a winning strategy from v_0 needs at least $k \cdot 2^k$ states. First we explain how Player 0 can win from v_0. Starting from there and reaching w_{k+1} for the first[7] time (P_1, R_1) has been requested in v_0. If Player 0 from w_{k+1} chooses to enter one of the vertices $\underline{z_j}$ or $\overline{z_j}$ for $j \neq 1$ then (P_1, R_1) is not responded to and Player 1 can move to vertex v_j requesting additionally[8] the pair $(P_{(j \bmod k)+1}, R_{(j \bmod k)+1})$. This causes Player 0 to lose the play for the following reason: The next time the play reaches w_{k+1} both (P_1, R_1) and (P_j, R_j) are active. No matter where Player 0 moves from w_{k+1}, if Player 1 afterwards moves to a sink state $s_{j'}$ then he wins the play because at least one of the pairs (P_1, R_1) or (P_j, R_j) are active and never be responded to later.

From this observation we can generalise the following: While the play does not

[7]The case is analogous to other situations that arise later in the play with different pairs (P_i, R_i) activated.

[8]If Player 0 moves to $\underline{z_k}$ or $\overline{z_k}$ then the pair (P_1, R_1) would be re-activated in v_k but in this case Player 1 can win by moving directly to s_k because then (P_1, R_1) remains active until infinity. A pair (P, R) is called active at a certain time instance if and only if P has been visited sometime before such that R has not been visited since.

reach a sink vertex s_i at least one pair (P_i, R_i) is active when reaching w_{k+1} and a response to (P_i, R_i) results in a request of the pair $(P_{(i \bmod k)+1}, R_{(i \bmod k)+1})$. Player 0 has to make sure that at each time only one of the pairs (P_i, R_i) is active and when w_{k+1} is reached he has only two possible vertices to move to, namely either $\underline{z_i}$ or $\overline{z_i}$. Which one he has to choose is explained below.

Anytime the play proceeds from w_1 to w_{k+1} at most one of the pairs $(\underline{P_i}, \underline{R_i})$ or $(\overline{P_i}, \overline{R_i})$ is activated after having reached w_{k+1}. This is due to the fact that we start with zero requests and the fact that each request of $(\underline{P_i}, \underline{R_i})$ in $\underline{v_i}$ causes $(\overline{P_i}, \overline{R_i})$ to be responded to, and vice versa. If $(\underline{P_i}, \underline{R_i})$ (resp. $(\overline{P_i}, \overline{R_i})$) is requested then Player 0 has to move to $\underline{z_i}$ (resp. $\overline{z_i}$) from w_{k+1}. If he moves to $\overline{z_i}$ (resp. $\underline{z_i}$) then Player 1 can move to s_i and $(\underline{P_i}, \underline{R_i})$ (resp. $(\overline{P_i}, \overline{R_i})$) is never responded to. If Player 0 chooses the "correct" vertex $\underline{z_i}$ (resp. $\overline{z_i}$) then the requests (P_i, R_i) and $(\underline{P_i}, \underline{R_i})$ (resp. $(\overline{P_i}, \overline{R_i})$) are responded to and there is no use for Player 1 to move to s_i. If neither $(\underline{P_i}, \underline{R_i})$ nor $(\overline{P_i}, \overline{R_i})$ is requested then Player 0 has a free choice between $\underline{z_i}$ and $\overline{z_i}$.

Now it should be clear how Player 0 can win from v_0 validating item 3. from the theorem. Furthermore, it is clear that the behaviour of Player 1 exactly determines which strategy Player 0 has to play in order to win. Any strategy automaton has to memorise two things. Firstly, it has to store the index i for the pairs (P_i, R_i). Note that this index proceeds cyclically from 1 to k. Secondly, it has to store which of the pairs $(\underline{P_i}, \underline{R_i})$, $(\overline{P_i}, \overline{R_i})$ are requested. Since Player 1 also has the opportunity to move directly from w_i to w_{i+1} the active pairs from the latest visits to $\underline{v_i}$ and $\overline{v_i}$ have to be memorised as well because they can get important later when index i is reached. Altogether, we have a memory of size $k \cdot 2^k$ as claimed in item 4.: k for memorising the index i for the pairs (P_i, R_i) and 2^k for memorising whether the pair $(\underline{P_i}, \underline{R_i})$ is requested[9] or not. $\square$

The core of a game reduction algorithm from Request-Response games to Büchi games is to find a set $F \subseteq S \times Q$ that simulates the request-response condition by the simpler Büchi condition. We use the method from [WHT03] where we have adapted the algorithm to our definition of game reduction. This means the memory update only depends on the source vertex of an edge. The algorithm involves a blow-up of the game graph which is exponential in the number of request-response conditions. As we have seen in Theorem 2.13 in the worst case this blow-up is unavoidable. For an improvement for the average case we refer to Section 4.3 of [Hüt03].

Theorem 2.14. *Let $\Gamma = (G, \varphi)$ be a Request-Response game where G has n vertices and $\mathcal{F} = \{(P_1, R_1), \ldots, (P_k, R_k)\}$. Then there exists a Büchi game $\Gamma' = (G', \varphi')$ such that G' has $2^{k+1} \cdot k \cdot n$ vertices and $\Gamma \leq \Gamma'$.*

Proof. We assume the winning condition for the Request-Response game as presented in Section 2.1.1. That means we have a conjunction of conditions of the form: "Whenever P_i is visited, then now or later R_i must be visited". The idea is to wait for a response to P_i and increase the value of i cyclically as soon as the set

[9]This implies that $(\overline{P_i}, \overline{R_i})$ is not requested and if $(\overline{P_i}, \overline{R_i})$ is requested then $(\underline{P_i}, \underline{R_i})$ is not requested.

R_i has been visited. If we have k equal to one then we have to add a dummy pair $(P_2, R_2) = (\emptyset, \emptyset)$ to be able to detect a completed cycle. If we have a request P_i responded to we set a "response" flag f which means visiting a final state. If we are waiting for a response that has no active request we move on directly to the next index and visit a final state. In infinity the flag is set again and again if and only if all requests are responded to eventually. We define $G' = (Q', E')$ as follows:

- $Q' := 2^{\{1,\dots,k\}} \times \{1,\dots,k\} \times \{0,1\} \times Q$
- $((P, i, f, q), (P', i', f', q')) \in E' :\Longleftrightarrow$
  - $P' := (P \cup \{i \mid q \in P_i\}) \setminus \{i \mid q \in R_i\}$
  - $i' := \begin{cases} i & \text{, if } i \in P' \\ (i \bmod k) + 1 & \text{, otherwise} \end{cases}$
  - $f' := \begin{cases} 0 & \text{, if } i = i' \\ 1 & \text{, otherwise} \end{cases}$
  - $(q, q') \in E$
- $F := 2^{\{1,\dots,k\}} \times \{1,\dots,k\} \times \{1\} \times Q$

As initial memory content we choose $s_0 = (\emptyset, 1, 0)$ (cf. item 2.(a) of Definition 2.11). Then it can easily be shown that the Büchi game with game graph G' satisfies items 1. to 3. from Definition 2.11. Thus, we have established a game reduction from Request-Response games to Büchi games and are done. For further details we refer to [WHT03]. □

2.2.5 Staiger-Wagner Games

One type of game (or winning condition) which is of important interest in this thesis is the Staiger-Wagner winning condition. In a Staiger-Wagner game $\Gamma = (G, \varphi)$ we are given a family $\mathcal{F} \subseteq 2^Q$ of subsets of the set of game graph vertices. A play ρ is winning for Player 0 iff $Occ(\rho) \in \mathcal{F}$. $Occ(\rho)$ is the set of vertices visited in the play ρ, i.e. $Occ(\rho) = \{q \mid \exists i : \rho(i) = q\}$. As we shortly see winning strategies for Staiger-Wagner games require strategy automata of a size exponential in the number of vertices of the given game graph.

Theorem 2.15. *There is a family G_n of game graphs with Staiger-Wagner winning condition such that the number of vertices of G_n is linear in n and any strategy automaton implementing a winning strategy from a certain vertex of G_n has at least 2^n states.*

Proof. We consider the game graph G_n with $6n + 1 \in O(n)$ vertices as shown in Figure 2.3 with the following Staiger-Wagner winning condition φ:

$$\rho \text{ is winning for Player } 0 :\Longleftrightarrow \forall i (i \in Occ(\rho) \iff i' \in Occ(\rho))$$

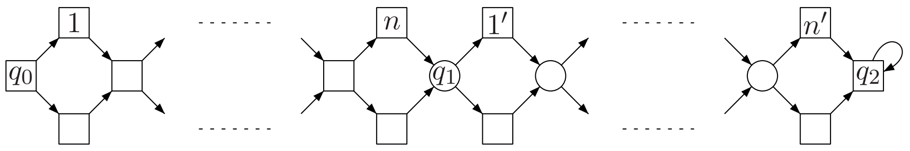


Figure 2.3: *Game Graph G_n*

There is an automaton winning strategy for Player 0 from q_0 memorising the states visited up to vertex q_1. Since the set $\{1, \ldots, n\}$ has 2^n subsets that many different sets may have been visited when Player 0 retains control in vertex q_1. Thus, a strategy automaton needs 2^n states for these 2^n different sets. For a contradiction, assume there is a strategy automaton $\mathcal{A}$ with less than 2^n states implementing a winning strategy $f_{\mathcal{A}}$ for Player 0 from q_0. Then there must be two play prefixes $u_1 \neq u_2$ which lead the automaton to the same state when reaching vertex q_1. The rest of the two plays is the same as determined by $\mathcal{A}$. Since $u_1 \neq u_2$ at most one of the two plays can be winning for Player 0. A contradiction, because $f_{\mathcal{A}}$ is a winning strategy for Player 0 from q_0 and, accordingly, both plays must be won by Player 0. $\square$

We now deal with the question of how to solve Staiger-Wagner games. For that we give a game reduction to weak parity games taken from [TL03]. As we have just seen the memory needed to win Staiger-Wagner games can be exponential in the size of G. This is due to the fact that in the worst case during a play ρ any subset of the vertex set can be visited. Hence, there are $|2^Q|$ many possibilities for $Occ(\rho)$. The idea for the reduction is to accumulate the visited states in a set R, i.e. $R \in 2^Q$. Consequently, for the game graph obtained from the game reduction we take as vertices the set $Q' = 2^Q \times Q$. Since at the beginning no vertices have been visited so far we start the play ρ' at vertex $(\emptyset, q)$ if ρ starts at q. As the set of edges we introduce transitions to update the memory component R. Taking $E' = \{((R, q), (R \cup \{q\}, q')) \mid (q, q') \in E,\ R \subseteq Q\}$ we are done with parts 1. and 2. of Definition 2.11. It remains to define the colouring function c. We take:

$$c((R, q)) = \begin{cases} 2 \cdot |R \cup \{q\}| - 1 & \text{, if } R \cup \{q\} \notin \mathcal{F} \\ 2 \cdot |R \cup \{q\}| & \text{, if } R \cup \{q\} \in \mathcal{F} \end{cases}$$

Since we never delete states from R the sequence of seen "memory states" (first components of the vertices in ρ') is weakly increasing. Hence, the sequence of seen colours is also weakly increasing and becomes stationary when we reach $R' = Occ(\rho)$. From that point onwards the unique maximal colour is seen and it is even iff $R' \in \mathcal{F}$, i.e. ρ is winning for Player 0 in $(G, \varphi) \iff \rho'$ is winning for Player 0 in (G', φ'). This completes part 3. of the definition and we are done.

Chapter 3

Reduction of Strategy Automata

We now deal with the main issue of this thesis. Our aim is to reduce the memory necessary for implementing winning strategies in infinite games. In this chapter we give two algorithms to solve this problem. As a first approach, in Section 3.1 we use a minimisation algorithm for deterministic finite automata with output (cf. [TL03]). We are given a strategy automaton implementing a finite-state strategy obtained from a game reduction algorithm and transform it into an equivalent one that has probably less states. We argue why this method of memory reduction has a certain drawback and try to overcome it with the second algorithm.

The second algorithm is a new approach to memory reduction. There, we also make use of game reduction algorithms. We are given the game $\Gamma = (G, \varphi)$ with the game graph $G = (Q, E)$ and the game $\Gamma' = (G', \varphi')$ with the game graph $G' = (Q', E')$ such that $\Gamma \leq \Gamma'$. Our idea is to simplify the game graph G' before computing a winning strategy on it. To achieve that we transform Γ' into a deterministic automaton $\mathcal{A}_{\Gamma'}$ accepting exactly those ω-words that are winning plays for Player 0 in the original game Γ. The state set of this automaton includes the set $S \times Q$ from the game graph G' (cf. Definition 2.11). We compute an equivalence relation $\approx$ on this set and from that obtain an equivalence relation $\approx_S$ on S. The algorithms we use for computing the equivalence relation $\approx$ are presented in the next chapter and are taken from [EWS05] and [Löd01]. The quotient automaton with respect to $\approx_S$ recognises the same language as the initial automaton $\mathcal{A}_{\Gamma'}$. Finally, we transform this quotient automaton back into an infinite game $\Gamma'' = (G'', \varphi'')$. $G'' = (Q'', E'')$ has as set of vertices $Q'' = S' \times Q$ where $S' = S/_{\approx_S}$. This means Q'' consists of less copies of Q than Q'. We show that the reduction of the set S to S' preserves the properties of a game reduction. This means $\Gamma \leq \Gamma''$ and we can extract a strategy automaton implementing a winning strategy for Player 0 in Γ from a positional winning strategy on G''. This automaton has less states than the automaton extracted from the positional winning strategy on G'. Hence, we have reduced the size of the memory for a winning strategy from $|S|$ to $|S'|$.

In Chapter 7 we compare the size of the strategy automata obtained by the two algorithms and present examples where one or the other approach yields better results.

3.1 The Marking Algorithm

Minimising an automaton means to compute a new automaton that has the minimal number of states among all automata that are equivalent to the given one. For DFA equivalence means to accept the same language. For a given DFA one can compute a minimal equivalent DFA very efficiently, namely in time $O(n \cdot \log(n))$ (cf. [Hop71])[1]. The result is an automaton which is uniquely determined (up to isomorphism). As we will shortly see analogous results can be achieved for finite automata with output. These are called Mealy automata. The only difference is that usual DFA have a set F of final states whereas Mealy automata have an output function, e.g. τ' from Definition 3.1 below. For this reason equivalence has to be reformulated. Two Mealy automata $\mathcal{A}_1, \mathcal{A}_2$ are said to be equivalent if and only if they compute the same function, i.e. $f_{\mathcal{A}_1} = f_{\mathcal{A}_2}$, where the output function τ_1 (resp. τ_2) of $\mathcal{A}_1$ (resp. $\mathcal{A}_2$) is used to define $f_{\mathcal{A}_1}$ (resp. $f_{\mathcal{A}_2}$) inductively.

We aim at using the minimisation algorithm for Mealy automata for minimising strategy automata. However, strategy automata and Mealy automata are nearly the same. The only difference is that we always consider the output function of a strategy automaton as a strategy in an infinite game. From now on we use only the term strategy automaton. In Theorem 3.2 we show that for a function, say f, which is computable by a strategy automaton, say $\mathcal{A}_f$, there exists a unique minimal automaton implementing it. It is important to note that this is only true for the function f whereas it is not necessarily true for the strategy that $\mathcal{A}_f$ implements for an infinite game. We come back to this at the end of this section but first we present the theorem on minimisation of strategy automata.

Before we can apply a marking algorithm to a strategy automaton we have to modify the domain of the output function τ from $S \times Q_0$ to $S \times Q$ by introducing a dummy output for Player 1 vertices.

Definition 3.1. Let $\mathcal{A} = (S, Q, s_0, \sigma, \tau)$ be a strategy automaton. We define the *extended strategy automaton* $\mathcal{A}' = (S, Q, s_0, \sigma, \tau')$ where $\tau' : S \times Q \to Q$ with

$$\tau'(s, q) = \begin{cases} \tau(s, q) & \text{, if } q \in Q_0 \\ * & \text{, if } q \in Q_1 \end{cases}$$

For given $\mathcal{A}'$ we can now specify the computed strategy:

$$\begin{aligned} f_{\mathcal{A}'}(\epsilon) &= \epsilon \\ f_{\mathcal{A}'}(uq) &= f_{\mathcal{A}'}(u) \underbrace{\tau'(\sigma(s_0, u), q)}_{\in\, Q \cup \{*\}} \end{aligned}$$

[1]We present a simpler and slower marking algorithm with a time complexity that is cubic in the number of states.

The extension of τ to τ' is necessary to guarantee the correctness of the minimisation algorithm. We thereby obtain an output function which is length preserving, i.e. $\forall \alpha, \alpha' \in Q^* : |\alpha| < |\alpha'| \Longrightarrow |\tau'(\alpha)| < |\tau'(\alpha')|$ where $\tau'(\alpha) := \tau'(\alpha_0) \dots \tau'(\alpha_{n-1})$ is the statewise concatenation of τ'-values. This would not work if we had e.g. an output function allowing outputs of length two for some arguments. We are now able to give a minimisation algorithm for $\mathcal{A}'$.

Theorem 3.2. *Let $\mathcal{A}'$ be defined as in Definition 3.1. Then $\mathcal{A}'$ can be transformed in polynomial time into a minimal automaton that is equivalent to $\mathcal{A}'$. This automaton is uniquely determined up to isomorphism.*

Proof. The idea for the proof is analogous to minimisation of usual DFA. The first step of the algorithm is to eliminate the non-reachable states of $\mathcal{A}'$. For this we execute a simple depth-first search to determine all reachable states. The time complexity for this is linear in the size of the automaton graph, i.e. the number of states plus the number of transitions (cf. [CLRS01]). The second step is the computation of the equivalence relation $\sim_{\mathcal{A}'}$ on S. We define two states s, s' to be equivalent if and only if considering each of them as initial state of the given automaton we obtain equal output functions:

$$s \sim_{\mathcal{A}'} s' :\Longleftrightarrow f_{\mathcal{A}'_s} = f_{\mathcal{A}'_{s'}}$$

Since $\sim_{\mathcal{A}'}$ is an equivalence relation we can deduce equivalence classes as usual. For $s \in S$ we denote by $[s]$ the equivalence class of s and by $[S]$ the set of equivalence classes of S, i.e. $S/_{\sim_{\mathcal{A}'}} = \{[s] \mid s \in S\}$. The relation $\sim_{\mathcal{A}'}$ is also used to define $[\sigma] : [S] \times Q \to [S]$. The new output function τ' needs only slight notational modifications. Note that both the following definitions are independent of the representative s:

$$\begin{aligned} [\sigma]([s], q) &:= [\sigma(s, q)] \\ [\tau']([s], q) &:= \tau'(s, q) \end{aligned}$$

We obtain the reduced automaton $\mathcal{A}'_{red} = ([S], Q, [s_0], [\sigma], [\tau'])$ of $\mathcal{A}'$ computing the function $f_{\mathcal{A}'}$, i.e. $f_{\mathcal{A}'_{red}} = f_{\mathcal{A}'}$.

If we assume that a function f is computable by an automaton then we can define a canonical automaton $\mathcal{A}_f$ uniquely determined by f which is equivalent to $\mathcal{A}'_{red}$ for every automaton $\mathcal{A}'$ computing f. $\mathcal{A}_f$ has the minimal number of states among all automata computing f. For $f : Q^* \to (Q \cup \{*\})^*$ and $u, v \in Q^*$ we define:

$$\begin{aligned} u \sim_f v :\Longleftrightarrow \forall w \in Q^* : \ & f(uw) \text{ and } f(vw) \text{ have the same suffix} \\ & \text{after the prefixes } f(u) \text{ and } f(v) \end{aligned}$$

$\sim_f$ is an equivalence relation. For $u \in Q^*$ the equivalence class of u is denoted by $\langle u \rangle$. We define $\mathcal{A}_f$ as follows:

$$\begin{aligned} \text{States} &: \sim_f\text{-classes} \\ \text{Initial state} &: \langle \epsilon \rangle \\ \text{Memory update function} &: \sigma_f(\langle u \rangle, q) := \langle uq \rangle \\ \text{Output function} &: \tau_f(\langle u \rangle, q) := \text{last letter of } f(uq) \end{aligned}$$

$\mathcal{A}_f$ computes the function f. If $\mathcal{A}'_{red}$ computes f by assumption then it has at least $index(\sim_f)$ many states where $index(\sim_f)$ denotes the number of equivalence classes of the equivalence relation $\sim_f$. Since $\mathcal{A}_f$ has exactly $index(\sim_f)$ states it consequently has the minimal number of states among all automata computing f. Finally we can give an isomorphism from $\mathcal{A}'_{red}$ to $\mathcal{A}_f$. All states of $\mathcal{A}'_{red}$ are reachable because of the first step of the algorithm. Thus, for each $[s] \in [S]$ there is a $u_{[s]}$ such that $[s] = [\sigma]([s_0], u_{[s]})$. The mapping $\alpha : [s] \to \langle u_{[s]} \rangle$ is the desired isomorphism.

We finally give a concise version of the marking algorithm in pseudo-code. Note that there are at most $|S|^2$ pairs of equivalent states and each call of the **while**-loop needs $O(|S| \cdot |Q|)$ steps to compute for the remaining pairs whether to be marked or not. Thus, Algorithm 3.3 also validates the polynomial time (and space) bound.

Algorithm 3.3. (Marking Algorithm)
Input: Strategy automaton $\mathcal{A} = (S, Q, s_0, \sigma, \tau)$ with $f_{\mathcal{A}} : Q^* \to (Q \cup \{*\})^*$
Output: Set of pairs of equivalent states

```
[Check one-step behaviour of pairs of states]
for all $(s, s') \in S \times S$ do
  if there is a $q \in Q$ such that $\tau(s, q) \neq \tau(s', q)$
    then mark $(s, s')$
[Check multi-step behaviour]
while some pair of states is newly marked do
  for all $(s, s') \in S \times S$ do
    if there is a $q \in Q$ such that $(\sigma(s, q), \sigma(s', q))$ is already marked
      then mark $(s, s')$
```

□

We have just seen that for a given strategy automaton $\mathcal{A}$ computing the output function f the minimisation algorithm yields the unique minimal automaton $\mathcal{A}_f$ among all automata computing f. Now we show that this is not necessarily true for the strategy g implemented by $\mathcal{A}_f$. This is due to the fact that g usually is a partial function $g : Q^* \dashrightarrow Q$. It is only defined for those state sequences from Q^* that are possible play prefixes in the considered game (cf. Definition 2.2). This means only some transitions of $\mathcal{A}_f$ are relevant for the definition of g. The transitions that are irrelevant can be redirected arbitrarily without changing the strategy implemented by the automaton. Hence, there can exist numerous automata implementing the strategy g that are minimal with respect to the functions they compute. To get this clearer consider the game graph from Figure 3.1. We use the Staiger-Wagner winning condition as introduced in Section 2.1.1 with the set $\mathcal{F} = \{\{0, 1\}, \{1, 2\}\}$.

In this game the winning region of Player 0 is $W_0 = \{0, 1\}$ and he has a winning strategy of size two as follows. If the play starts in vertex 0 then Player 0 moves to vertex 1 and stays there. If the play starts in vertex 1 then he moves to vertex 2. This strategy is implemented by the strategy automaton $\mathcal{A}_1$ in Figure 3.2. A

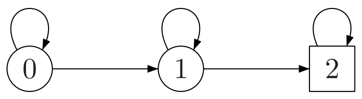


Figure 3.1: *Staiger-Wagner game*

symbol left of a vertical line is an input letter and a symbol right of that line is an output letter.

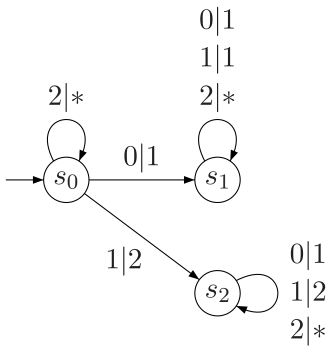


Figure 3.2: *Strategy Automaton $\mathcal{A}_1$*

This automaton is minimal in the sense that for each pair of states the output functions computed from these states are distinct. Hence, Algorithm 3.3 yields three equivalence classes. If we consider the winning strategy for Player 0 in the game from Figure 3.1 then some of the transitions of $\mathcal{A}_1$ are irrelevant. This is particularly true for the transition from s_2 reading input letter 0. If state s_2 is reached in the automaton then Player 0 in the game moves from vertex 1 to vertex 2. Consequently, only input letter 2 is read for the rest of the run because Player 0 can only move from vertex 2 back to vertex 2. This means, if we redirect the transition with input letter 0 to state s_1 then the obtained automaton $\mathcal{A}_2$ still implements the same strategy. The new automaton can be seen in Figure 3.3. The automaton $\mathcal{A}_2$ is no longer minimal because from states s_0 and s_2 the same output functions are computed. Algorithm 3.3 yields the equivalence classes $\{s_0, s_2\}$ and $\{s_1\}$. The corresponding quotient automaton can be seen in Figure 3.4.

We have proven that for a strategy automaton $\mathcal{A}_f$ implementing the output function f and the strategy g Algorithm 3.3 yields the unique minimal automaton $\mathcal{A}_f$ among all automata computing f. The above example shows that $\mathcal{A}_f$ is not necessarily the minimal automaton implementing g because $\mathcal{A}_f$ not only depends on g but also on f. This is one reason why the minimisation algorithm for strategy automata is not always suitable for memory reduction. In the following section we present an algorithm to overcome this. As a further motivation we explain a second shortcoming of using the minimisation algorithm for strategy automata for memory reduction.

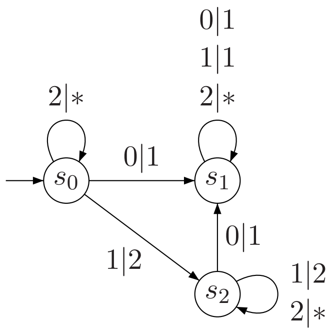


Figure 3.3: *Strategy Automaton $\mathcal{A}_2$*

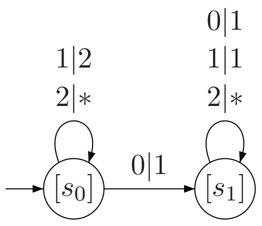


Figure 3.4: *Quotient Automaton of $\mathcal{A}_2$*

3.2 Reduction of Game Graphs

We now describe a new algorithm to reduce the memory necessary for implementing winning strategies in infinite games. In Section 2.2.3 we have explained how to construct a strategy automaton $\mathcal{A}$ from a game reduction. There, we are given two games, say $\Gamma = (G, \varphi)$ and $\Gamma' = (G', \varphi')$, such that Γ is reducible to Γ' (cf. Definition 2.11). The transition structure of G' determines the memory update function σ and a positional winning strategy for Γ' determines the output function τ of $\mathcal{A}$. τ specifies the function $f_{\mathcal{A}}$ computed by $\mathcal{A}$. To minimise the size of $\mathcal{A}$ we can apply the algorithm from Section 3.1. There, we have already argued why this does not yield a minimal automaton for the implemented strategy but only for the computed function.

Now we bring forward a further argument why this approach to memory reduction sometimes is too weak. The problem is that the minimal automaton $\mathcal{A}_f$ not only depends on the output function f but also on the strategy g implemented by it.[2] Even if we assume that $\mathcal{A}_f$ has the minimal number of states necessary to implement the strategy g then this does not mean that it has the minimal number of states necessary for implementing any winning strategy for the given game Γ. This

[2]We have already seen this in Section 3.1 but there we have focused on the definition of the irrelevant transitions.

is due to the fact that g can be too complicated. To make this clear let us have a look at the following Staiger-Wagner game with $\mathcal{F} = \{\{0,1\},\{0,2\},\{0,1,2,3\}\}$.

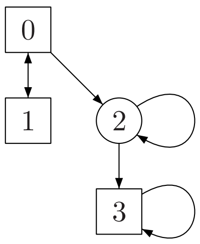


Figure 3.5: *A simple Staiger-Wagner game*

In this example Player 0 has the winning region $W_0 = \{0,1\}$. If a play starting from W_0 reaches vertex 2 then Player 0 has to distinguish between two situations. If Player 1 has chosen to move directly from vertex 0 to vertex 2 without an indirection via vertex 1 then to win Player 0 has to stay in vertex 2 forever. In this case we have $Occ(\rho) = \{0,2\}$. If vertex 1 has been visited at least once then Player 0 has to move to vertex 3 ($Occ(\rho) = \{0,1,2,3\}$). To implement this strategy[3] f we give an automaton $\mathcal{A}_f$ of size two. $\mathcal{A}_f$ is optimal in the sense that there exists no strategy automaton of smaller size implementing a winning strategy for Player 0 in this game. This means that Player 0 has no positional winning strategy in Γ. The reader may verify that $\mathcal{A}_f$ implements the winning strategy f for Player 0 from his winning region W_0.

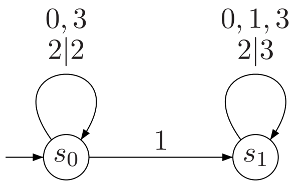


Figure 3.6: *Strategy automaton $\mathcal{A}_f$*

We can think of a different strategy which is also winning for Player 0. What we change is the behaviour in case of a visit to vertex 1. When reaching vertex 2 Player 0 delays the move from vertex 2 to vertex 3 for exactly n times. This strategy f_n is implemented by the strategy automaton $\mathcal{A}_n$ in Figure 3.7.

The dashed line shall indicate more states with outgoing transitions defined as for s_1 and s_n. If $\mathcal{A}_n$ is in state s_i ($1 \leq i \leq n$) and vertex 2 is read then it outputs vertex 2 and moves on to vertex s_{i+1}. Note that for $n = 0$ we get the automaton $\mathcal{A}_f$

[3]From now on f denotes a strategy in an infinite game but no longer an output function.

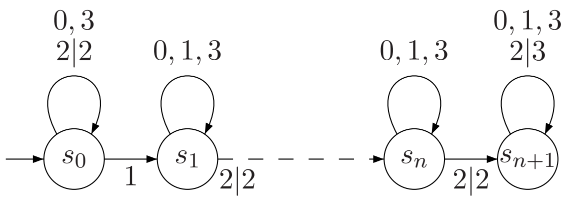


Figure 3.7: *Strategy automaton $\mathcal{A}_n$*

from above, i.e. $\mathcal{A}_0 = \mathcal{A}_f$. It is clear that for any $n \in \mathbb{N}$ the strategy f_n implemented by $\mathcal{A}_n$ is winning for Player 0. This is because a play is an infinite sequence but Player 0 lingers in vertex 2 only for finitely many times. After n revisits to vertex 2 the play proceeds to vertex 3. Thereby, if ρ starts in W_0 we get $Occ(\rho) = \{0, 1, 2, 3\}$ or $Occ(\rho) = \{0, 2\}$. It is important to note that for any n the automaton $\mathcal{A}_n$ is minimal. This is due to the fact that $\mathcal{A}_n$ has to count the number of revisits to vertex 2 and none of the $n+2$ states can be abstained from. On the other hand the strategy f_n is much too complicated. Any of the states s_i for $1 \leq i \leq n$ could be left out in $\mathcal{A}_n$. The resulting automaton, some $\mathcal{A}_j$ with $j \leq n$, would still implement a winning strategy but one that was different from f_n. Since we can choose any value for n and $\mathcal{A}_n$ is minimal we conclude that minimised strategy automata can get arbitrarily large. The reason for this is that the strategy that is to be minimised can be much too complicated and the minimisation algorithm does not allow for a simplification of it. This is a major drawback.

The above observations give rise to the question whether it is possible to algorithmically simplify a given strategy before applying the minimisation algorithm. Before we present an algorithm that does so we need to address the ambiguity of the word "Reduction". In particular we need to distinguish two different meanings:

1. Game Reduction: The methodology of a game reduction is similar to that of reduction of decidability problems (cf. [Hro01]). We are given a game $\Gamma = (G, \varphi)$ where games with a winning condition like φ are not solvable by positional winning strategies in general. To solve Γ we use an algorithm that transforms it into a new game $\Gamma' = (G', \varphi')$. Usually, the size of G' is exponentially large in the size of G. On the other hand φ' is simpler than φ in the sense that Γ' can be solved by positional winning strategies. From a solution to Γ' we can compute a solution to Γ. An example for a game reduction is the transformation from Request-Response games to Büchi games (cf. Theorem 2.14).

2. Reduction: If we speak of a reduction (without the word "Game" in front) of an automaton $\mathcal{A}$ then we mean the problem of finding an equivalent automaton $\mathcal{B}$ which has less states than the given automaton $\mathcal{A}$. In this setting equivalence means to accept the same language, i.e. $L(\mathcal{A}) = L(\mathcal{B})$. In our algorithm we view game graphs as automata, and vice versa. Therefore, we also speak of reductions of game graphs but not only of automata. However, the reduction of

a game graph is always achieved via a transformation to the level of automata.

In the sequel we use both the terms "Game Reduction" and "Reduction" depending on the context.

We now present an algorithm that reduces the memory component S of a game reduction (cf. Definition 2.11). The reduction itself is accomplished in an automata-theoretic framework. For that we transform the output of a game reduction, i.e. a game Γ', into an ω-automaton. The reduction of the automaton will be carried out in such a way that the properties of a game reduction are preserved. This includes that the set of vertices of the new game graph is still the cartesian product of a finite memory S' and the set of vertices Q of the original game graph. The reduced automaton is transformed back into an infinite game and, thereby, we obtain a reduction of a game graph. The new game has the same structural properties as Γ' which means that we can extract a new strategy automaton from it (cf. Theorem 2.12). Our algorithm is depicted in Figure 3.8.

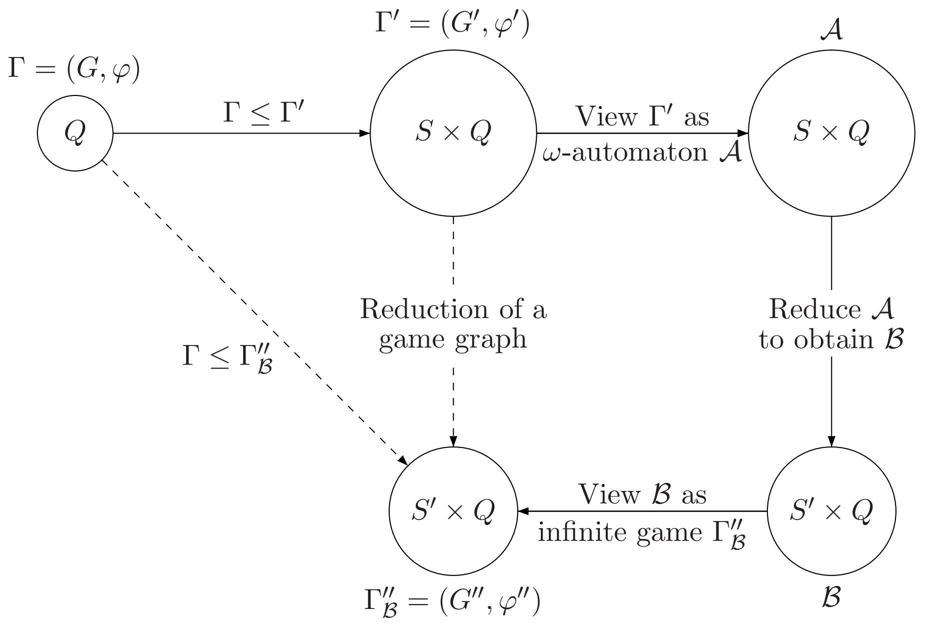

$\Gamma = (G, \varphi)$
Q
$\Gamma \leq \Gamma'$
$\Gamma' = (G', \varphi')$
$S \times Q$
View Γ' as ω-automaton $\mathcal{A}$
$\mathcal{A}$
$S \times Q$
Reduction of a game graph
Reduce $\mathcal{A}$ to obtain $\mathcal{B}$
$\Gamma \leq \Gamma''_{\mathcal{B}}$
$S' \times Q$
View $\mathcal{B}$ as infinite game $\Gamma''_{\mathcal{B}}$
$S' \times Q$
$\Gamma''_{\mathcal{B}} = (G'', \varphi'')$
$\mathcal{B}$

Figure 3.8: *Reduction of a game graph*

Since the resulting memory S' is smaller than S we can compute a strategy automaton from the new game which has less states than the strategy automaton computed from Γ'. The algorithm works as described below.

Algorithm 3.4. (Memory Reduction)
Input: Infinite game $\Gamma = (G, \varphi)$
Output: Strategy automaton $\mathcal{A}_f$ for Player 0 from W_0

1. We establish a game reduction from Γ to a new game $\Gamma' = (G', \varphi')$. G' has as vertices the set $S \times Q$ for a finite set S where Q is given by G.

2. The game Γ' is viewed as ω-automaton $\mathcal{A}$. This means we introduce two auxiliary states q_0 and q_{sink}, we assign labels to the transitions and we understand the winning condition φ' as acceptance component F of the automaton $\mathcal{A}$. This step is explained in more detail in the next Section.

3. We reduce the ω-automaton $\mathcal{A}$. From an equivalence relation $\approx$ on $S \times Q$ we compute the equivalence relation $\approx_S$ on S and construct the corresponding quotient automaton $\mathcal{B} := \mathcal{A}/_{\approx_S}$. $\mathcal{B}$ has as vertices the set $S' \times Q$ where $S' := S/_{\approx_S}$. This step is described very detailed in Section 3.2.2.

4. In this step we take the reverse of step two. That means we transform the automaton $\mathcal{B}$ from step three back into an infinite game $\Gamma''_{\mathcal{B}} = (G'', \varphi'')$ (cf. Definition 3.8).

5. The claim of Theorem 3.14 is that the game $\Gamma''_{\mathcal{B}}$ obtained in step four has the same structural properties as the game Γ'. By this we mean that Γ is reducible to $\Gamma''_{\mathcal{B}}$. This is indicated by the left dashed arrow in Figure 3.8. We solve the game $\Gamma''_{\mathcal{B}}$ and use its solution to construct the strategy automaton $\mathcal{A}_f$ for Player 0 from his winning region W_0. $\mathcal{A}_f$ is the output of the algorithm.

In step four the algorithm computes a game $\Gamma''_{\mathcal{B}} = (G'', \varphi'')$ with $\Gamma \leq \Gamma''_{\mathcal{B}}$. To prove that steps two, three and four really preserve the properties of a game reduction we need to claim an additional quality of the equivalence relation $\approx$ used in step three, namely what we call *compatibility* (cf. Definition 3.9). Since G'' has as vertices the set $S' \times Q$ and $S' = S/_{\approx_S}$ we get that G'' includes at most as much memory as G'. This means $|S'| \leq |S|$ where, of course, we want that $|S'| < |S|$. In this case the strategy automaton extracted from the game reduction computed by our algorithm has less states than the strategy automaton extracted from the initial game reduction. This is exactly what we are aiming at. Next we describe steps two and four where we go from game graphs to ω-automata and back.

3.2.1 From Games to Automata

For the reduction of the set of vertices of a game graph we make use of known algorithms for automata. Therefore we first introduce a notation to treat game graphs as automata, and vice versa. The automata we are looking for should read infinite words, namely state sequences that are possible plays on a given game graph. We take the plays of $\Gamma = (G, \varphi)$ and define a language[4] $L_0(\Gamma)$ as the set of those plays of Γ that are won by Player 0, i.e. $L_0(\Gamma) = \{\rho = q_0q_1q_2\ldots \in Q^\omega \mid \forall i : (q_i, q_{i+1}) \in E \text{ and } \rho \text{ is winning for Player 0 in } \Gamma\}$. We use Q' obtained from a game reduction[5] as state set for the automaton and Q as input alphabet. The transition relation allows to move from (s, q) to (s', q') reading input letter q' if and only if there is a corresponding edge in G'.

[4]We distinguish the language $L_0(\Gamma) \subseteq Q^\omega$ from the set of plays $\text{Win}_0 \subseteq \Psi(\Gamma) \subseteq Q^\omega$.
[5]plus two additional states

Definition 3.5. Let $\Gamma = (G, \varphi)$ and $\Gamma' = (G', \varphi')$ be two infinite games such that Γ is reducible to Γ' (cf. Definition 2.11). For Γ' we define the (deterministic) *game automaton* $\mathcal{A}_{\Gamma'} = ((S \times Q)\dot{\cup}\{q_0, q_{sink}\}, Q, q_0, \delta, F)$. The transition function $\delta = \delta_1 \,\dot{\cup}\, \delta_2 \,\dot{\cup}\, \delta_3 \,\dot{\cup}\, \delta_4$ is defined as follows:

1. One transition from the auxiliary[6] initial state q_0 to each vertex representing an initial game position:
   $\delta_1 : \{q_0\} \times Q \to \{s_0\} \times Q$
   $\delta_1(q_0, q') := (s_0, q')$ where s_0 is the unique initial memory content
2. Regular transitions obtained from G' labelled by the Q-component of their target state:
   $\delta_2 : (S \times Q) \times Q \dashrightarrow S \times Q$
   $\delta_2((s, q), q') := (s', q')$ where $((s, q), (s', q')) \in E'$
3. Transitions from game positions (s, q) to the sink state q_{sink} for catching input words which are not possible plays on G:
   $\delta_3 : (S \times Q) \times Q \dashrightarrow \{q_{sink}\}$
   $\delta_3((s, q), q') := q_{sink}$ where $(q, q') \notin E$
4. One transition from q_{sink} back to q_{sink} for each input letter $q' \in Q$:
   $\delta_4 : \{q_{sink}\} \times Q \to \{q_{sink}\}$
   $\delta_4(q_{sink}, q') := q_{sink}$

A run of $\mathcal{A}_{\Gamma'}$ is called regular if and only if it does not contain q_{sink}. Note that a regular run of the automaton mimics the corresponding play of Γ'. This means for $\rho' \in (S \times Q)^\omega$:

$$q_0\rho' \text{ is a regular run of } \mathcal{A}_{\Gamma'} \text{ if and only if } \rho' \text{ is a play of } \Gamma'$$

The acceptance component F is defined according to the winning condition of Γ'. For a play ρ' of Γ' we define:

$$q_0\rho' \text{ is accepting if and only if } \rho' \text{ is winning for Player 0}$$

Any non-regular run is defined to be rejecting.

In Chapter 5 we show that the automata we use there have indeed the acceptance properties of Definition 3.5. From Definitions 2.11 and 3.5 we can immediately deduce the following observation.

Corollary 3.6. *Let $\Gamma = (G, \varphi)$ and $\Gamma' = (G', \varphi')$ be games such that $\Gamma \leq \Gamma'$. Let ρ be a play of Γ, ρ' be the corresponding play of Γ' and $q_0\rho'$ be the corresponding run of $\mathcal{A}_{\Gamma'}$. Then the following are equivalent:*

1. *ρ is winning for Player 0 in Γ*

[6] q_0 is introduced to enable $\mathcal{A}_{\Gamma'}$ to read plays as a whole, i.e. including the initial vertex.

2. *ρ' is winning for Player 0 in Γ'*

3. *$q_0\rho'$ is an accepting run of $\mathcal{A}_{\Gamma'}$*

The definition of δ_2 is the essential part of an automaton that reads plays of a given game graph. Parts one, three and four are only necessary to get the automaton deterministic. We could also label the transitions with the Q-component of the source state but not of the target state and use δ_2 only. This would yield a nondeterministic automaton reading exactly those state sequences which are possible plays in Γ. The acceptance or non-acceptance of the automaton would not be influenced because it is affected by the visited states only and not by the letters read. But, doing so we would not be able to apply the minimisation algorithm from [Löd01] since it requires a deterministic automaton.

Remark 3.7. Let $\Gamma' = (G', \varphi')$ be a game obtained from $\Gamma = (G, \varphi)$ by a game reduction according to Definition 2.11 and let $G' = (Q', E')$. Let $\mathcal{A}_{\Gamma'}$ be the induced game automaton of Γ' according to Definition 3.5. Then $\mathcal{A}_{\Gamma'}$ accepts the winning plays for Player 0 in Γ, i.e. $L(\mathcal{A}_{\Gamma'}) = L_0(\Gamma)$.

Proof. The equality follows from Definition 3.5 and Corollary 3.6. Let $\rho = q_1q_2\ldots$ be a play of Γ, $\rho' = (s_1, q_1)(s_2, q_2)\ldots$ be the corresponding play of Γ' and $q_0\rho' = q_0(s_1, q_1)(s_2, q_2)\ldots$ be the corresponding run of $\mathcal{A}_{\Gamma'}$. Then, by Corollary 3.6 we get:

$$\rho \text{ is winning for Player 0} \iff q_0\rho' \text{ is accepting}$$

Furthermore, by Definition 3.5 a transition of $\mathcal{A}_{\Gamma'}$ is labelled by the Q-component of its target state. Hence, $q_0\rho' = q_0(s_1, q_1)(s_2, q_2)\ldots$ is the unique run on the ω-word $\alpha = q_1q_2\ldots = \rho$. □

We have just explained for a reduced game Γ' how to transform it into an automaton. We also need a formalisation for the reverse direction. There, we require the given automaton to be of a special format, namely that of automaton $\mathcal{A}_{\Gamma'}$ above. We introduce the term *automaton game*, a notational variant of a game automaton:

Definition 3.8. Let $\mathcal{B} = ((S \times Q)\dot{\cup}\{q_0, q_{sink}\}, Q, q_0, \delta, F)$ be a game automaton. Then $\Gamma' = (G', \varphi')$ is said to be the *automaton game* of $\mathcal{B}$ and given by:

1. The game graph $G' = (Q', E')$:
   - $Q' = S \times Q$ with $(s, q) \in Q'_i :\iff q \in Q_i$ $(i = 0, 1)$
   - $E' = \{((s, q), (s', q')) \mid \delta((s, q), q') = (s', q')\}$
2. The winning condition φ':
   - ρ' is won by Player 0 $:\iff q_0\rho'$ is an accepting run of $\mathcal{B}$

As we have seen in this section a game graph can easily be considered as an automaton accepting a language of plays which are winning for Player 0. This is due to the fact that an automaton can be seen as game arena with only one player trying to build up a successful run. There is no opponent ruling out certain plays by his strategy. Moreover, the partition of the set of vertices into $Q_0 \dot{\cup} Q_1$ is no longer necessary. Thus, automata acceptance is in a sense easier to achieve than winning a game.

Note that game graphs and game automata are in some sense the same structures. The only differences between those two are that a game automaton has two extra vertices q_0 and q_{sink} and that its edges are labelled by Q. But these differences are just on a notational level. Since one can be directly obtained from the other we treat game automata as a variant of game graphs. In the sequel it may happen that the terms "game graph" and "game automaton" are mixed up depending on the context the term is used in. However, we actually mean the same thing even if we mostly use the term "game automaton".

3.2.2 Reduction of Game Automata

The main objective of this section is to find an equivalence relation between states which has the following properties:

1. Induce a quotient automaton of substantially smaller size,
2. Preserve the properties of a game reduction and
3. Be computable efficiently[7]

At first sight items one and three seem conflicting because we deal with game automata of a size exponential in the size of the initial game graph G. Additionally, it seems difficult to achieve item 1. in general because minimisation algorithms are known only for a small number of ω-automata. In Chapter 4 we present two algorithms for state space reduction of ω-automata. The first one uses the notion of delayed simulation and can be applied to nondeterministic Büchi automata. We consider only deterministic automata for which the reduction algorithm has the time complexity[8] $O(m \cdot \log(n))$. A special subclass of deterministic Büchi automata are deterministic weak Büchi automata. There we have an exact minimisation algorithm in our hands that needs time $O(n \cdot \log(n))$. This algorithm can also be used to minimise weak parity game automata that have a special format. Since the presented algorithms have to be modified to be applicable to game automata they do not guarantee the same reduction results for our case as for standard ω-automata. The major problem is that the reduction algorithms start with a game automaton of exponential size. Moreover, we have to choose a more restrictive definition for the equivalence of states to assure that the properties of a game reduction are preserved (cf. item 2. above). Item 1., however, will be verified or falsified in the implementation.

We assume that the automaton under consideration has the state space $S \times Q$ and we carry out the reduction only on this set. Thus, the two helping states q_0 and

[7]in polynomial time

[8]m is the number of edges and n the number of states of the given automaton.

q_{sink} are never touched. Before we can formulate our algorithm in full detail we have to examine item 2. more closely. We must get a clear picture of the requirements to be met by the equivalence relation used for state space reduction. We could compute the equivalence relation on the state space as given, i.e. on $S \times Q$. But it becomes immediately clear from our motivation that we do not need to consider pairs of states which differ in their Q-component. If we merge (s_1, q_1) and (s_2, q_2) for $q_1 \neq q_2$ then we no longer have several copies of one unique game graph but numerous different ones. Facing that we would no longer have a game graph satisfying item 1. of Definition 2.11. We are merely interested in a reduction of the memory component S and not in a modification of Q. Thus, we introduce an equivalence relation $\approx_S$ on S which is uniquely determined by $\approx$. We introduce a special property for equivalence relation $\approx$ called *compatibility*. Later we show that if $\approx$ (which is used to compute $\approx_S$) is compatible then the quotient game automaton with respect to $\approx_S$ satisfies the definition of a game reduction.

Definition 3.9. Let $\mathcal{A} = ((S \times Q)\dot{\cup}\{q_0, q_{sink}\}, Q, q_0, \delta, F)$ be a game automaton with acceptance component F and let $\approx$ be an equivalence relation on $S \times Q$. We say that $\approx$ is *compatible* with $\mathcal{A}$ if and only if the following hold:

1. For all $s_1, s_2 \in S, q, q' \in Q$:
   $(s_1, q) \approx (s_2, q) \Longrightarrow \delta((s_1, q), q') \approx \delta((s_2, q), q')$
2. Let $\rho = q_0(s_1, q_1)(s_2, q_2)\dots$ and $\rho' = q_0(s'_1, q'_1)(s'_2, q'_2)\dots$ be any two runs of $\mathcal{A}$ such that $(s_i, q_i) \approx (s'_i, q'_i)$ for all $i > 0$. Then ρ is accepting if and only if ρ' is accepting.

Definition 3.10. Let $\mathcal{A}$ be a game automaton and $\approx$ a compatible equivalence relation on $S \times Q$. We define the (deterministic) quotient game automaton $\mathcal{A}/_\approx = ((S \times Q)/_\approx\dot{\cup}\{q_0, q_{sink}\}, Q, q_0, \delta/_\approx, F/_\approx)$ where

$$\begin{aligned}
\delta/_\approx(q_0, q') = [(s_0, q')] &:\Longleftrightarrow \delta(q_0, q') = (s_0, q'),\\
\delta/_\approx([(s, q)], q') = [(s', q')] &:\Longleftrightarrow \delta((s, q), q') = (s', q'),\\
\delta/_\approx([(s, q)], q') = q_{sink} &:\Longleftrightarrow \delta((s, q), q') = q_{sink} \text{ and}\\
\delta/_\approx(q_{sink}, q') = q_{sink} &:\Longleftrightarrow \delta(q_{sink}, q') = q_{sink}.
\end{aligned}$$

$[(s, q)]$ denotes the equivalence class of (s, q), i.e. the set of all $(s', q') \in S \times Q$ with $(s', q') \approx (s, q)$. $(S \times Q)/_\approx$ denotes the quotient state space of $S \times Q$, i.e. the set of all equivalence classes of $S \times Q$ with respect to $\approx$. The acceptance component $F/_\approx$ is defined on the abstract level of runs. We define a run $\rho' = q_0[(s'_1, q'_1)][(s'_2, q'_2)]\dots$ of $\mathcal{A}/_\approx$ to be accepting if and only if there exists an accepting run $\rho = q_0(s_1, q_1)(s_2, q_2)\dots$ of $\mathcal{A}$ with $(s_i, q_i) \approx (s'_i, q'_i)$ for each $i > 0$.

Item 1. of Definition 3.9 is necessary to get a well-defined transition function for $\mathcal{A}/_\approx$. Now we show that a compatible equivalence relation preserves the recognised language of a game automaton.

Remark 3.11. Let $\mathcal{A}$ be a game automaton and $\approx$ a compatible equivalence relation on $S \times Q$. Then $\mathcal{A}$ and $\mathcal{A}/_\approx$ are equivalent, i.e. $L(\mathcal{A}) = L(\mathcal{A}/_\approx)$.

Proof. For the equality we show set inclusion in both directions. For the direction from left to right let $\alpha = q_1 q_2 \ldots \in L(\mathcal{A})$ and $\rho = q_0(s_1, q_1)(s_2, q_2) \ldots$ be the run of $\mathcal{A}$ on α. Let $\rho' = q_0[(s_1, q_1)][(s_2, q_2)] \ldots$ be the corresponding run of $\mathcal{A}/_{\approx}$ on α. Since ρ is accepting it follows from the definition of $F/_{\approx}$ that ρ' is accepting as well. Hence, $\alpha \in L(\mathcal{A}/_{\approx})$. For the other direction consider $\alpha' = q_1' q_2' \ldots \in L(\mathcal{A}/_{\approx})$ and the run $\rho' = q_0[(s_1', q_1')][(s_2', q_2')] \ldots$ of $\mathcal{A}/_{\approx}$ on α'. Since ρ' is accepting there exists an accepting run $\rho = q_0(s_1, q_1)(s_2, q_2) \ldots$ of $\mathcal{A}$ such that $(s_i, q_i) \approx (s_i', q_i')$ for all $i > 0$. From item 2. of Definition 3.9 it follows that any run $\rho'' = q_0(s_1'', q_1'')(s_2'', q_2'') \ldots$ of $\mathcal{A}$ with $(s_i'', q_i'') \approx (s_i', q_i')$ for all $i > 0$ is accepting. Thus, this is in particular true for the run on α' which means $\alpha' \in L(\mathcal{A})$. Altogether, we get $\alpha \in L(\mathcal{A}) \iff \alpha \in L(\mathcal{A}/_{\approx})$ for any $\alpha \in Q^\omega$. □

Now we use $\approx$ to define an equivalence relation $\approx_S$ on S which is the basis for the reduction of game graphs. For two memory contents s_1, s_2 to be $\approx_S$-equivalent we require that for each $q \in Q$ the pairs $(s_1, q), (s_2, q)$ are equivalent with respect to some equivalence relation $\approx$ depending on the acceptance component F of $\mathcal{A}$.

Definition 3.12. Let $\mathcal{A} = ((S \times Q) \dot\cup \{q_0, q_{sink}\}, Q, q_0, \delta, F)$ be a game automaton and let $\approx$ be a compatible equivalence relation on $S \times Q$. We define the equivalence relation $\approx_S$ on the memory component S as follows:

$$s_1 \approx_S s_2 :\Longleftrightarrow \forall q \in Q : (s_1, q) \approx (s_2, q)$$

We get the quotient automaton $\mathcal{A}/_{\approx_S} = ((S/_{\approx_S} \times Q) \dot\cup \{q_0, q_{sink}\}, Q, q_0, \delta/_{\approx_S}, F/_{\approx_S})$. Given a total order $\prec_S$ on S, $([s], q) \in S/_{\approx_S} \times Q$ and $(q, q') \in E$ we define

$$\delta/_{\approx_S}(([s], q), q') := ([s_{\min}], q')$$

where

$$s_{\min} := \min\{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } \delta((\hat{s}, q), q') = (\hat{s}', q')\}.$$

The acceptance component $F/_{\approx_S}$ is defined on the abstract level of runs. Let $\rho' = q_0([s_1], q_1)([s_2], q_2) \ldots$ be a run of $\mathcal{A}/_{\approx_S}$. We define ρ' to be accepting if and only if the corresponding run $\rho = q_0[(s_1, q_1)][(s_2, q_2)] \ldots$ of $\mathcal{A}/_{\approx}$ is accepting.

In Chapter 4 we define particular acceptance conditions[9] for the automata used there and show that they satisfy the definition of $F/_{\approx}$ and $F/_{\approx_S}$ from above. The definition of $\approx_S$ assures that the resulting quotient automaton $\mathcal{A}/_{\approx_S}$ has as state set the cartesian product $S' \times Q$ of a finite memory component $S' = S/_{\approx_S}$ and the set of vertices Q given by game graph G. We show that $\mathcal{A}$ and $\mathcal{A}/_{\approx_S}$ are equivalent.

Lemma 3.13. *Let $\mathcal{A}$ be a game automaton and let $\approx$ be a compatible equivalence relation on $S \times Q$. Then $\mathcal{A}$ and $\mathcal{A}/_{\approx_S}$ are equivalent, i.e. $L(\mathcal{A}) = L(\mathcal{A}/_{\approx_S})$.*

[9]We use Büchi and weak parity acceptance conditions.

Proof. Let $\rho' = q_0([s_1], q_1)([s_2], q_2)\ldots$ be the run of $\mathcal{A}/_{\approx_S}$ on $\alpha = q_1q_2\ldots$ and let $\rho = q_0[(s_1, q_1)][(s_2, q_2)]\ldots$ be the run of $\mathcal{A}/_{\approx}$ on α. By Definition 3.12 ρ' is accepting if and only if ρ is accepting. If $\alpha \in L(\mathcal{A}/_{\approx_S})$ then $\alpha \in L(\mathcal{A}/_{\approx})$ and if $\alpha \notin L(\mathcal{A}/_{\approx_S})$ then $\alpha \notin L(\mathcal{A}/_{\approx})$. Hence, $\mathcal{A}/_{\approx}$ and $\mathcal{A}/_{\approx_S}$ are equivalent, i.e. $L(\mathcal{A}/_{\approx}) = L(\mathcal{A}/_{\approx_S})$. Furthermore, we have seen in Remark 3.11 that $\mathcal{A}$ and $\mathcal{A}/_{\approx}$ are equivalent, i.e. $L(\mathcal{A}) = L(\mathcal{A}/_{\approx})$. Thus, $L(\mathcal{A}) = L(\mathcal{A}/_{\approx_S})$ holds. □

In Chapter 4 we introduce two reduction algorithms for Büchi automata and DWA. As we will see DWA are equivalent to a certain subclass of weak parity automata and we can use the former to reduce the latter. For the Büchi case we use the notion of mutual simulation for computing an equivalence relation $\approx$ after modifying the set of accept states. For the case of DWA we have to compute a colouring on the state space first. After that we can apply a simple minimisation algorithm for DFA. Before we give the technical details for both cases we show that our method yields the desired result. This means that if steps one to four are performed as depicted in Figure 3.8 then Γ is reducible to Γ''.

3.2.3 Correctness Proof

Let us sum up what we have done so far. First we have introduced the notion of game reduction which we used to solve games where positional strategies do not suffice to win those games. From a game reduction we were able to extract a strategy automaton for Player 0 implementing a winning strategy by solving the respective game. Our aim was to find strategies needing less memory than the ones that were implemented by these strategy automata. The Marking Algorithm presented in Section 3.1 refers to a fixed strategy and, thus, was not appropriate for our intention. As an intermediate step we took a simple transformation from game graphs to automata and back. The reason for this was that we wanted to apply known reduction algorithms for ω-automata. However, we actually established a reduction for game graphs (via automata). Now, the first question is whether we can easily extract a strategy automaton from the game graph resulting from step four as we could for game reductions. The second question is of what size these automata will be in general, i.e. how much memory we need to implement non-positional strategies using this method. We start out by showing that the game graph G'' obtained in step four has indeed similar properties as the game graph G' obtained in step one (cf. Figure 3.8).

Theorem 3.14. *Let $\Gamma = (G, \varphi)$ and $\Gamma' = (G', \varphi')$ be games and Γ be reducible to Γ'. Let $\mathcal{A}$ be a game automaton for Γ' and $\approx$ a compatible equivalence relation on $S \times Q$. Then Γ is reducible to the unique automaton game Γ'' of $\mathcal{A}/_{\approx_S}$.*

Proof. From Definition 3.12 it follows that $\mathcal{A}/_{\approx_S}$ has the state set $(S' \times Q) \dot\cup \{q_0, q_{sink}\}$ for some finite set S'. Hence, $\mathcal{A}/_{\approx_S}$ is a game automaton and we can transform it into a unique automaton game Γ'' (cf. Definition 3.8).

Now we show that Γ is reducible to $\Gamma'' = (G'', \varphi'')$. We have to check items one to three from Definition 2.11. Item 1. of the definition is immediately satisfied by Definition 3.8 and our remarks above.

For item 2.(a) of the definition we first have to find a unique initial memory content. We take the equivalence class of s_0, i.e. $[s_0] \in S/_{\approx_S}$ and obtain as initial game positions the set $\{([s_0], q) \mid q \in Q\}$. We now have to check the new transition relation E'' of G''. To show item 2.(b).i let $([s], q) \in Q''$ and $(q, q') \in E$:

$$\begin{aligned}
([s], q) \in Q'' &\overset{\text{Def. 3.8,3.12,3.5}}{\Longrightarrow} (s, q) \in Q' \\
&\overset{(q,q')\in E, \Gamma\leq\Gamma'}{\Longrightarrow} \exists (s', q') \in Q' : ((s, q), (s', q')) \in E' \\
&\overset{\text{Def. 3.5,3.12}}{\Longrightarrow} \exists ([s_{\min}], q') \in Q'' : \delta/_{\approx_S}(([s], q), q') = ([s_{\min}], q') \\
&\overset{\text{Def. 3.8}}{\Longrightarrow} \exists ([s_{\min}], q') \in Q'' : (([s], q), ([s_{\min}], q')) \in E''
\end{aligned}$$

The reverse direction, i.e. item 2.(c), is justified as follows:

$$\begin{aligned}
(([s], q), ([s_{\min}], q')) \in E'' &\overset{\text{Def. 3.5}}{\Longrightarrow} \delta/_{\approx_S}(([s], q), q') = ([s_{\min}], q') \\
&\overset{\text{Def. 3.12}}{\Longrightarrow} \exists \hat{s} : \hat{s} \approx_S s \text{ and } \delta((\hat{s}, q), q') = (s_{\min}, q') \\
&\overset{\text{Def. 3.8}}{\Longrightarrow} ((\hat{s}, q), (s_{\min}, q')) \in E' \\
&\overset{\Gamma\leq\Gamma'}{\Longrightarrow} (q, q') \in E
\end{aligned}$$

To complete the proof of item 2. we have to show the uniqueness of the memory update, i.e. 2.(b).ii of Definition 2.11. Let $(([s], q), ([s_1], q_1)), (([s], q), ([s_2], q_2)) \in E''$. By Definition 3.8 this means $\delta/_{\approx_S}(([s], q), q_1) = ([s_1], q_1)$ and $\delta/_{\approx_S}(([s], q), q_2) = ([s_2], q_2)$. We have to show that $[s_1] = [s_2]$ holds. For a contradiction, assume $[s_1] \neq [s_2]$:

$$\begin{aligned}
[s_1] \neq [s_2] &\Longrightarrow s_1 \neq s_2 \\
&\overset{\text{Def. 3.12}}{\Longrightarrow} \min\{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } \delta((\hat{s}, q), q_1) = (\hat{s}', q_1)\} \neq \\
&\qquad \min\{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } \delta((\hat{s}, q), q_2) = (\hat{s}', q_2)\} \\
&\Longrightarrow \{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } \delta((\hat{s}, q), q_1) = (\hat{s}', q_1)\} \neq \\
&\qquad \{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } \delta((\hat{s}, q), q_2) = (\hat{s}', q_2)\} \\
&\overset{\text{Def. 3.8}}{\Longrightarrow} \{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } ((\hat{s}, q), (\hat{s}', q_1)) \in E'\} \neq \\
&\qquad \{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } ((\hat{s}, q), (\hat{s}', q_2)) \in E'\} \\
&\overset{\text{Def. 2.11}}{\Longrightarrow} \lightning
\end{aligned}$$

The last inequality yields a contradiction. This is due to the fact that the uniqueness of the memory update, i.e. item 2.(b).ii of Definition 2.11, holds for G'. Hence, $[s_1] = [s_2]$ and we are done with the proof for item 2. of Definition 2.11.

What remains to be shown is that Player 0 wins a play ρ of Γ if and only if he wins the corresponding play ρ'' of Γ'' (item 3.). For the proof we utilise the fact that automata $\mathcal{A}$ and $\mathcal{A}/_{\approx_S}$ are equivalent. Let ρ be a play of Γ and let ρ' (resp. ρ'') be the corresponding play of Γ' (resp. Γ'') according to item 2. of Definition 2.11:

$$\begin{aligned}
\rho \text{ is winning for Player 0 in } \Gamma \quad &\overset{\text{Def. 2.11}}{\Longleftrightarrow} \quad \rho' \text{ is winning for Player 0 in } \Gamma' \\
&\overset{\text{Def. 3.5,3.8}}{\Longleftrightarrow} \quad q_0\rho' \text{ is an accepting run of } \mathcal{A}_{\Gamma'} \\
&\overset{\text{Rem. 3.7}}{\Longleftrightarrow} \quad \rho \in L(\mathcal{A}_{\Gamma'}) \\
&\overset{\text{La. 3.13}}{\Longleftrightarrow} \quad \rho \in L(\mathcal{A}_{\Gamma'}/_{\approx_S}) \\
&\overset{\text{Rem. 3.7}}{\Longleftrightarrow} \quad q_0\rho'' \text{ is an accepting run of } \mathcal{A}_{\Gamma'}/_{\approx_S} \\
&\overset{\text{Def. 3.8,3.5}}{\Longleftrightarrow} \quad \rho'' \text{ is winning for Player 0 in } \Gamma''
\end{aligned}$$

□

Chapter 4

Reduction of ω-Automata

In this chapter we present two algorithms for the reduction of ω-automata. The first one is for nondeterministic Büchi automata. It is based on the notion of delayed simulation introduced in [EWS05]. The second one is for a special case of Büchi automata, namely deterministic weak Büchi automata (DWA). This algorithm is taken from [Löd01]. The reduction algorithms are the core for our algorithm to reduce the memory for winning strategies in infinite games. They predominantly affect the size of the resulting strategy automata.

4.1 Reduction of Büchi Automata

We have already argued that the problem of reducing a game graph can equivalently be formulated as the problem of reducing the corresponding game automaton. For the reduction of Büchi game automata we use the notion of delayed simulation (cf. [EWS05]). In our setting the situation is slightly simpler than in the paper because we consider only deterministic automata. Delayed simulation on a Büchi game automaton can easily be computed by means of a modification of the set of final states and the computation of the direct bisimulation relation on this new state space. We start out explaining a notion of games which can be used to determine pairs of states that can be merged in a quotient automaton.

4.1.1 Simulation Games

Definition 4.1. Let $\mathcal{A}$ be a Büchi automaton with no dead ends for which we define the *basic simulation game* $G_{\mathcal{A}}(q_0, q'_0)$. It is played by two players, *Spoiler* and *Duplicator*. In each play of $G_{\mathcal{A}}(q_0, q'_0)$ Spoiler and Duplicator move in turn, Spoiler starting at vertex q_0 and Duplicator starting at vertex q'_0. In each round Spoiler moves from vertex q_i to a vertex q_{i+1} by choosing a transition $(q_i, a_i, q_{i+1}) \in \Delta$. Duplicator has to respond by a transition with the same labelling, i.e. he has to choose a transition $(q'_i, a_i, q'_{i+1}) \in \Delta$. Spoiler wins the game $G_{\mathcal{A}}(q_0, q'_0)$ if and only if the game halts, i.e. Duplicator cannot respond by a transition with the same labelling as chosen by Spoiler most recently. If Duplicator wins the game two infinite runs $\rho = q_0 a_0 q_1 a_1 q_2 \ldots$ and $\rho' = q'_0 a_0 q'_1 a_1 q'_2 \ldots$ are built up by the two players. The pair (ρ, ρ') is called the *outcome* of the play.

Both runs produce the same (infinite) sequence of letters $\omega = a_0 a_1 a_2 \ldots$. Regarding the acceptance condition of a Büchi automaton the definition of a basic game is not appropriate. ρ and ρ' being labelled by the same sequence ω does not mean that both runs are accepting with respect to the Büchi acceptance condition. Hence, what we need to take into account is which states are visited, i.e. whether the states visited are accepting or not.

Definition 4.2. The *delayed simulation game* $G^{de}_{\mathcal{A}}(q_0, q'_0)$ is defined as the basic simulation game $G_{\mathcal{A}}(q_0, q'_0)$ from Definition 4.1 extended by the rule that an outcome (ρ, ρ') is winning for Duplicator if the following holds:

$$\forall i (q_i \in F \Longrightarrow \exists j \geq i : q'_j \in F)$$

If $\mathcal{A}$ is a Büchi automaton then we say that state q'_0 *delayed simulates* state q_0 if there is a winning strategy for Duplicator in the game $G^{de}_{\mathcal{A}}(q_0, q'_0)$. We denote this by $q_0 \preceq_{de} q'_0$.

Since we deal with deterministic automata we can simplify Definition 4.2 for our case. For each letter Spoiler can choose exactly one possible transition and Duplicator must answer by the unique transition with the same letter. Hence, the simulation game between the two players simplifies to a game where the moves of Duplicator are uniquely determined by the moves of Spoiler. Spoiler tries to build up a run $\rho = q_0 q_1 q_2 \ldots$ which satisfies the Büchi winning condition such that the run $\rho' = q'_0 q'_1 q'_2 \ldots$ for the same labelling sequence violates the Büchi winning condition. If such a word does not exist then Duplicator wins the simulation game $G^{de}_{\mathcal{A}}(q_0, q'_0)$ and $q_0 \preceq_{de} q'_0$ holds. We can rewrite the definition of the delayed simulation relation $\preceq_{de}$ in the following way:

$$q_0 \preceq_{de} q'_0 :\Longleftrightarrow$$
$$\forall w \in \Sigma^* (\delta(q_0, w) \in F \Longrightarrow \forall \alpha \in \Sigma^\omega \exists i : \delta(q'_0, w\alpha[0 \ldots i]) \in F)$$

If we have the outcome (ρ, ρ') and the corresponding play is won by Duplicator and the run ρ is accepting then the run ρ' is accepting as well. For our later definition of states being equivalent we wish the other direction to hold as well. But this is not guaranteed by the definition of delayed simulation. To overcome this we later introduce the notion of mutual simulation.

We want to prove the result stated in the last paragraph in a more formal way.

Proposition 4.3. *Let $\mathcal{A}$ be a Büchi automaton with state set Q and let $\preceq_{de}$ denote the delayed simulation relation as defined above. Then the following holds:*

1. *The relation $\preceq_{de}$ is a reflexive and transitive relation on Q.*

2. *If $q_0 \preceq_{de} q'_0$ then $L(\mathcal{A}_{q_0}) \subseteq L(\mathcal{A}_{q'_0})$.*

Proof. Reflexitivity of $\preceq_{de}$ is obvious since each state simulates itself. Duplicator wins the corresponding game by reproducing Spoiler's moves. To show transitivity, assume that $q_0 \preceq_{de} q_0'$ and $q_0' \preceq_{de} q_0''$. Then, Duplicator wins both $G^{de}_{\mathcal{A}}(q_0, q_0')$ and $G^{de}_{\mathcal{A}}(q_0', q_0'')$. Assume that $\rho = q_0q_1q_2\dots$ is accepting. Then $\rho' = q_0'q_1'q_2'\dots$ must be accepting and since Duplicator wins $G^{de}_{\mathcal{A}}(q_0', q_0'')$ we can conclude that $\rho'' = q_0''q_1''q_2''\dots$ is accepting as well. Hence, if there exists some $i \in \mathbb{N}$ such that $q_i \in F$ then there must exist some $j \geq i$ such that $q_j' \in F$ which implies that there must exist some $k \geq j$ with $q_k'' \in F$. Hence,

$$q_i \in F \Longrightarrow \exists k \geq i : q_k'' \in F$$

which means that Duplicator wins $G^{de}_{\mathcal{A}}(q_0, q_0'')$ and we are done.

The second part of Proposition 4.3 follows immediately. If $q_0 \preceq_{de} q_0'$ and $w \in L(\mathcal{A}_{q_0})$ then the run of $L(\mathcal{A}_{q_0'})$ on w is accepting by the above details. Thus $w \in L(\mathcal{A}_{q_0'})$. □

4.1.2 Mutual Simulation vs. Bisimulation

Our aim is now to define a relation on the state space of a game automaton such that q_0 is in relation to q_0' if and only if q_0 simulates q_0' and q_0' simulates q_0. To achieve that we introduce the *mutual simulation game* on $\mathcal{A}$. It is defined as the basic simulation game with the difference that at the beginning Spoiler has a free choice whether he wants to build up the run ρ starting in q_0 or ρ' starting in q_0'. Note that a game automaton has a deterministic transition structure and the letter a_0 chosen by Spoiler uniquely determines the transitions (q_0, a_0, q_1) and (q_0', a_0, q_1'). Thus, it makes no difference whether he chooses to build up ρ or ρ'. Nor would it make any difference if Spoiler was allowed to switch between runs ρ and ρ' whenever he wanted to, say after a_4, a_{749} and a_{8139}. The only important thing is the ω-word $\alpha = a_0a_1a_2\dots$ that he chooses. The latter situation where Spoiler is allowed to switch arbitrarily between ρ and ρ' describes another type of game, namely a bisimulation game. This gives rise to the following fundamental observation:

For game automaton $\mathcal{A}$
mutual simulation and bisimulation are the same.

Of course, we can state the same for the delayed versions of the corresponding (bi)simulation games. Thus, we can abstain from mutual delayed simulation and use delayed bisimulation instead. We use results from [EWS05] to show that mutual delayed simulation is appropriate to compute a reduced Büchi automaton. Furthermore, we show that delayed bisimulation can be computed efficiently. These observations are the basis for our algorithm to reduce Büchi game automata.

In a bisimulation game Spoiler chooses one of the runs ρ or ρ' and a transition to move along. Subsequently, Duplicator answers choosing a transition with the same labelling for the other run. If a play comes to a halt then Spoiler wins. In this case the pair of vertices where the play started is not bisimilar. If the play goes on forever the winning condition for the delayed bisimulation game is as follows:

If an accept state is seen at position i of either run
then an accept state must be seen at some position $j \geq i$ in the other run.

Mutual simulation defines an equivalence relation $\simeq$ on the state space. The delayed version of it preserves the recognised language of a given Büchi automaton. Thus, it is appropriate for our aim of state space reduction. The mutual delayed simulation relation $\simeq_{de}$ is defined as follows:

$$q \simeq_{de} q' :\iff q \preceq_{de} q' \text{ and } q' \preceq_{de} q$$

It is important to note that computing $\simeq_{de}$ asymptotically requires no more time than computing $\preceq_{de}$. For computing $\simeq_{de}$ we only have to consider at most twice as many pairs of states as for computing $\preceq_{de}$. As we show now mutual delayed simulation is a suitable choice as equivalence relation for the reduction of a Büchi automaton. For the proof we treat the more general case where we assume that the given Büchi automaton $\mathcal{A}$ is nondeterministic. For this purpose let us define the quotient automaton $\mathcal{A}/_{\approx}$ of a nondeterministic Büchi automaton.

Definition 4.4. Let $\mathcal{A}$ be an automaton with Büchi acceptance condition and let $\approx$ be an equivalence relation on the state space Q of $\mathcal{A}$. We define the quotient automaton $\mathcal{A}/_{\approx} = (Q/_{\approx}, \Sigma, [q_0], \Delta/_{\approx}, F/_{\approx})$ where

$$\Delta/_{\approx} = \{([q], a, [q']) \mid \exists q_0 \in [q], \exists q_0' \in [q'] \text{ with } (q_0, a, q_0') \in \Delta\}$$

Here, $[q]$ denotes the equivalence class of q, i.e. the set of all $q' \in Q$ with $q' \approx q$. $Q/_{\approx}$ denotes the quotient state space, i.e. the set of all equivalence classes of Q with respect to $\approx$.

$$Q/_{\approx} = \{[q] \mid [q] \text{ is an equivalence class of } Q \text{ with respect to } \approx\}$$

$F/_{\approx}$ denotes the set of all equivalence classes of final states of $\mathcal{A}$. This means $[q] \in Q/_{\approx}$ is declared final if and only if there exists $q' \in F$ with $q' \approx q$.

To prove that mutual delayed simulation preserves the recognised language of a Büchi automaton we first show a helping lemma.

Lemma 4.5. *Let $\mathcal{A}$ be a Büchi automaton. Then the following hold:*

1. *If $q_0 \preceq_{de} q_0'$ and $(q_0, a_0, q_1) \in \Delta$ then there exists q_1' with $q_1 \preceq_{de} q_1'$ and $(q_0', a_0, q_1') \in \Delta$.*

2. *If $q_0 \preceq_{de} q_0'$ and $[q_0]a_0[q_1]a_1[q_2]\ldots$ is a finite or infinite run of $\mathcal{A}/_{\simeq_{de}}$ then there exists a run $q_0'a_0q_1'a_1q_2'\ldots$ of $\mathcal{A}$ of the same length such that $q_i \preceq_{de} q_i'$ for every i.*

3. *If $q_0 \preceq_{de} q_0''$ and $[q_0]a_0[q_1]a_1[q_2]\ldots$ is an infinite run of $\mathcal{A}/_{\simeq_{de}}$ with $q_0 \in F$ then there exists a finite run $q_0''a_0\ldots a_{r-1}q_r''$ of $\mathcal{A}$ such that $q_j \preceq_{de} q_j''$ for $j \leq r$ and $q_r'' \in F$.*

Proof. We start proving part 1. If $q_0 \preceq_{de} q_0'$ then, by the definition of $\preceq_{de}$, Duplicator wins $G^{de}_{\mathcal{A}}(q_0, q_0')$. Let f be a winning strategy such that $q_1' = f(q_0 q_0' a_0 q_1)$. Consequently, there must be a transition $(q_0', a_0, q_1') \in \Delta$. From (q_1, q_1') we can construct a winning strategy f' for Duplicator in the corresponding game $G^{de}_{\mathcal{A}}(q_1, q_1')$. For this we take f' with $f'(\rho) = f(q_0 q_0' a_0 \rho)$ where $\rho \in Q(Q\Sigma Q)^*$.

For part 2 we use induction on the first part of the lemma. By definition of $\mathcal{A}/_{\simeq_{de}}$ we know there exist $\hat{q_0}$ and $\hat{q_1}$ such that (i) $\hat{q_0} \simeq_{de} q_0$, (ii) $\hat{q_1} \simeq_{de} q_1$ and (iii) $(\hat{q_0}, a_0, \hat{q_1}) \in \Delta$. From (i) and our assumption we obtain $\hat{q_0} \preceq_{de} q_0 \preceq_{de} q_0'$ and by transitivity of $\preceq_{de}$ we get $\hat{q_0} \preceq_{de} q_0'$. So, from (iii) and part 1 we deduce there exists $(q_0', a_0, q_1') \in \Delta$ such that $\hat{q_1} \preceq_{de} q_1'$. Again by transitivity, from (ii) we get $q_1 \preceq_{de} q_1'$. We have thus constructed the first transition (q_0', a_0, q_1') of the desired run $\rho' = q_0' a_0 q_1' \dots$ which we can continue inductively in an analogous way.

For part 3 assume the run $[q_0]a_0[q_1]a_1[q_2]\dots$ as stated in the lemma. By part 2 we know about the existence of the infinite run $q_0' a_0 q_1' a_1 q_2' \dots$ where we set $q_0' := q_0$. Let f be a winning strategy for Duplicator in $G^{de}_{\mathcal{A}}(q_0', q_0'')$ and consider $q_0'' a_0 q_1'' a_1 q_2'' \dots$ defined by $q_{i+1}'' := f(q_0' q_0'' a_0 \dots q_i')$. As in the proof of part 1 we can argue that $q_j \preceq_{de} q_j' \preceq_{de} q_j''$ holds for every j. Because $q_0' = q_0 \in F$ there must exist r with $q_r'' \in F$. □

With the help of this lemma we can now prove that mutual delayed simulation is a good choice for reducing the state space of a Büchi automaton. It preserves the recognised language.

Theorem 4.6. *Let $\mathcal{A}$ be a Büchi automaton. Then $L(\mathcal{A}) = L(\mathcal{A}/_{\simeq_{de}})$.*

Proof. We show the claim by set inclusion in both directions. To see that $L(\mathcal{A}) \subseteq L(\mathcal{A}/_{\simeq_{de}})$ consider an accepting run $\rho = q_0 a_0 q_1 a_1 q_2 \dots$ of $\mathcal{A}$. From the definition of a quotient automaton (cf. Definition 4.4) it follows directly that $\rho = [q_0]a_0[q_1]a_1[q_2]\dots$ is an accepting run of $\mathcal{A}/_{\simeq_{de}}$. Both runs produce the same ω-sequence $\alpha = a_0 a_1 a_2 \dots$. Thus, we can copy the run from $\mathcal{A}$ into $\mathcal{A}/_{\simeq_{de}}$ by using the q_i of ρ as representatives for the classes $[q_i]$. This makes the direction from left to right easy.

Proving $L(\mathcal{A}) \supseteq L(\mathcal{A}/_{\simeq_{de}})$ is more difficult because an accepting run $\rho = [q_0]a_0[q_1]a_1[q_2]\dots$ of $\mathcal{A}/_{\simeq_{de}}$ cannot be directly transformed into the accepting run $\rho = q_0 a_0 q_1 a_1 q_2 \dots$ of $\mathcal{A}$. This is because we cannot guarantee that ρ is a run of $\mathcal{A}$ at all. Nevertheless, we can construct an accepting run of $\mathcal{A}$ over the same word inductively. We construct a sequence $(\rho_l)_{l \in \mathbb{N}}$ of finite runs of $\mathcal{A}$ on finite prefixes of $\alpha = a_0 a_1 a_2 \dots$ such that ρ_{l+1} strictly extends ρ_l and contains at least $l+1$ final states. Taking the limit of the sequence of the ρ_l we obtain the desired run. W.l.o.g. we may assume that the first state of ρ, i.e. q_0, is the initial state of $\mathcal{A}$. At the induction start we set $\rho_0 = q_0$. In the induction step we construct ρ_{l+1} from ρ_l as follows. If $\rho_l = q_0' a_0 q_1' a_1 q_2' \dots q_i'$ has already been constructed such that $q_i \preceq_{de} q_i'$ then there exists $j > i$ such that $q_j \in F$. By the second part of Lemma 4.5 there exists a run $q_i' a_i q_{i+1}' a_{i+1} q_{i+2}' \dots q_j'$ such that $q_j \preceq_{de} q_j'$. Finally, by the third part of the Lemma there exists $k \geq j$ and a run $q_j' a_j q_{j+1}' a_{j+1} q_{j+2}' \dots q_k'$ such that $q_k \preceq_{de} q_k'$ and $q_k' \in F$. Set $\rho_{l+1} = \rho_l a_i q_{i+1}' \dots q_k'$. □

One possibility for computing certain simulation and bisimulation relations is a reformulation as parity games (cf. [EWS05]). However, this does not help for our choice. We have just proven that mutual delayed simulation can be used to reduce the state space of a Büchi automaton. Now we utilise the remarks from the beginning of this subsection and switch over from mutual delayed simulation to delayed bisimulation. As we will see the delayed bisimulation relation for a given Büchi automaton can be computed very efficiently, namely in time $O(m \cdot \log(n))$ (m is the number of transitions of $\mathcal{A}$ and n the number of states). We reduce its computation to the computation of another bisimulation relation on the state space, called direct bisimulation.

Definition 4.7. The *direct (strong) simulation game* $G^{di}_{\mathcal{A}}(q_0, q_0')$ is defined as the basic simulation game $G_{\mathcal{A}}(q_0, q_0')$ (cf. Definition 4.1) extended by the rule that an outcome (ρ, ρ') is winning for Duplicator if the following holds:

$$\forall i (q_i \in F \Longrightarrow q_i' \in F)$$

If $\mathcal{A}$ is a Büchi automaton then we say that state q_0' *direct simulates* state q_0 if there is a winning strategy for Duplicator in the game $G^{di}_{\mathcal{A}}(q_0, q_0')$. We denote this by $q_0 \preceq_{di} q_0'$.

Extending direct simulation in the notion of bisimulation yields the corresponding *direct bisimulation game* $G^{bi,di}_{\mathcal{A}}(q_0, q_0')$. An outcome (ρ, ρ') is winning for Duplicator if and only if he can exactly mimic the visits to final states independent of which pebble is chosen by Spoiler. That means for each i either both q_i and q_i' are final or none is:

$$\forall i (q_i \in F \iff q_i' \in F)$$

The corresponding equivalence relation is denoted by $\approx_{di}$ and by $\approx_{de}$ we denote the delayed bisimulation relation. As we will shortly see the delayed bisimulation relation is very similar to the direct bisimulation relation. For direct bisimulation to yield the same equivalence relation in a given Büchi automaton as delayed bisimulation we need to do some preprocessing. More precisely, we modify the set of final states by computing some kind of predecessor *closure* of F. At the beginning we set $F' := F$ and iterate the following until a fixed point is reached:

If there is a state q such that all successors of q are in F', put q in F'.

Therewith we obtain a new automaton called $cl(\mathcal{A})$ which is defined as $\mathcal{A}$ with the difference that it has as final states the set F'.

Lemma 4.8. *Let $\mathcal{A}$ be a Büchi automaton and $cl(\mathcal{A})$ be obtained as explained above. Then $L(\mathcal{A}) = L(cl(\mathcal{A}))$.*

Proof. Assume we add state q to F' in the ith iteration. Then all successors of q must have been added before, i.e. in iteration $1, \ldots, i-1$, or they have already been in F. Thus, if state q is visited in a run ρ then a final state (a successor of q) is visited no matter if q was final or not. Hence, declaring q to be a final state we do not get more accepting runs than before. Vice versa, if $q \notin F'$ then there must be at least one state q' which is a non-final successor of q. The acceptance of any run going through q and q' (or any other non-final successor of q) is not touched since both states are non-final in $\mathcal{A}$ and remain non-final in $cl(\mathcal{A})$. If a run goes through q and a final successor of q then both in $\mathcal{A}$ and $cl(\mathcal{A})$ a final state (namely the successor of q) is visited. Again, the acceptance of the run is not influenced by the changed set of accept states. Altogether, the same runs as before are accepting and we obtain the claim from the lemma. □

It is obvious that this iterative computation can be done in linear time in the size of $\mathcal{A}$. Afterwards we can compute the direct bisimulation relation for $cl(\mathcal{A})$ in time $O(m \cdot \log(n))$ (cf. [PT87]) which altogether yields the latter complexity for the whole computation costs. What is left to be shown is that direct bisimulation for $cl(\mathcal{A})$ yields indeed the same relation as delayed bisimulation for $\mathcal{A}$.

Lemma 4.9. *Let $\mathcal{A}$ be a Büchi automaton. For any two states q, q' we have*

$$q \approx_{de} q' \text{ in } \mathcal{A} \iff q \approx_{di} q' \text{ in } cl(\mathcal{A})$$

Proof. We show that any winning strategy for Duplicator in the delayed bisimulation game on $\mathcal{A}$ is a winning strategy for Duplicator in the direct bisimulation game on $cl(\mathcal{A})$, and vice versa. We start with a winning strategy f for Duplicator in the delayed bisimulation game $G^{bi,de}_{\mathcal{A}}(q_0, q'_0)$. Take the two runs ρ and ρ' and the pair of states (q_i, q'_i) reached in the game after i moves of each of the two players. Let Duplicator play according to f. We have to show that $q_i \in F' \iff q'_i \in F'$. For a contradiction, by symmetry it suffices to show that $q_i \notin F'$ and $q'_i \in F'$ is not possible. In the latter situation, since $q_i \notin F'$ there is an infinite path starting in q_i without any final state on it. Take a strategy for Spoiler playing exactly this path. Since $q'_i \in F'$ there is no path beginning in q'_i without final state on it. Altogether, $q'_i \in F'$ and Duplicator has no chance that a state $q_j \in F'$ for a $j \geq i$ is reached (he even does not have the opportunity at all to choose any edge for run ρ after move i). So, Spoiler wins the delayed bisimulation game. A contradiction to f being a winning strategy for Duplicator in this game.

For the other direction, take a winning strategy f for Duplicator in the direct bisimulation game $G^{bi,di}_{cl(\mathcal{A})}(q_0, q'_0)$. We claim that it is winning for Duplicator in $G^{bi,de}_{\mathcal{A}}(q_0, q'_0)$ as well. For any index i, again, it suffices to show that $q_i \in F$ implies the existence of a j such that $q'_j \in F$. By definition of F' we get that $q_i \in F$ implies $q_i \in F'$. Since f is winning in the direct bisimulation game this implies $q'_i \in F'$. We distinguish between two cases, $q'_i \in F$ and $q'_i \in F' \setminus F$. For the first case, set $j := i$ and immediately obtain $q'_j = q'_i$ as the final state of $\mathcal{A}$ we were looking for. For the second case, if $q'_i \in F' \setminus F$ then it must have been added in an iteration, say l, such that all successors of q'_i must have been in F' after iteration $l - 1$. This yields an analogous situation for the successors of q'_i and iteration $l - 1$ as for q'_i and iteration

l. Repeating this inductively we can be sure to eventually leave $F' \setminus F$ and enter F. Thus, we can conclude that any infinite path starting from q'_i reaches a final state of $\mathcal{A}$ after finitely many steps. We take this final state for the q'_j. So, in both cases there is a $j \geq i$ such that $q'_j \in F$ and we get the desired result. □

We have shown several properties of $\mathcal{A}$, $cl(\mathcal{A})$ and particular equivalence relations on them. One property we want to use in our algorithm is rather obvious but still missing to be proven.

Remark 4.10. Let $\mathcal{A}$ be a deterministic Büchi automaton. Then we get the equality $L(cl(\mathcal{A})) = L(cl(\mathcal{A})/_{\approx_{di}})$.

Proof. Lemma 4.9 holds for any Büchi automaton $\mathcal{A}$. If we apply it to $cl(\mathcal{A})$ then we get:

$$q \approx_{de} q' \text{ in } cl(\mathcal{A}) \iff q \approx_{di} q' \text{ in } cl(cl(\mathcal{A}))$$

Since $cl(\cdot)$ is an idempotent operator the latter equivalence simplifies to

$$q \approx_{de} q' \text{ in } cl(\mathcal{A}) \iff q \approx_{di} q' \text{ in } cl(\mathcal{A})$$

This means that for $cl(\mathcal{A})$ the relations $\approx_{de}$ and $\approx_{di}$ are the same. Hence, the respective quotient automata recognise the same language (no matter if $\approx_{de}$ or $\approx_{di}$ preserved the recognised language of $cl(\mathcal{A})$ or not). We get $L(cl(\mathcal{A})/_{\approx_{de}}) = L(cl(\mathcal{A})/_{\approx_{di}})$. Since $\mathcal{A}$ is deterministic $cl(\mathcal{A})$ is deterministic as well which means that

$$L(cl(\mathcal{A})/_{\simeq_{de}}) = L(cl(\mathcal{A})/_{\approx_{di}}).$$

From Theorem 4.6 we know that $\simeq_{de}$ preserves the recognised language of any Büchi automaton. We thereby obtain:

$$L(cl(\mathcal{A})) = L(cl(\mathcal{A})/_{\simeq_{de}}) = L(cl(\mathcal{A})/_{\approx_{di}})$$

□

4.1.3 Reduction of Büchi Game Automata

If we apply the game reduction algorithm from Section 2.2.4 to a given Request-Response game then we obtain a Büchi game. Hence, we need to give a transformation from Büchi games to Büchi game automata and have to show that the resulting automaton $\mathcal{A}$ satisfies Definition 3.5. Moreover, we give an acceptance condition for the quotient automaton $\mathcal{A}/_{\approx}$ (resp. $\mathcal{A}/_{\approx_S}$) and show that it satisfies Definition 3.10 (resp. 3.12).

Let $\Gamma' = (G', \varphi')$ be a reduced Büchi game where $G' = (S \times Q, E')$ is the reduced game graph with the Büchi winning condition

$$\rho' \in \text{Win}_0 :\iff Inf(\rho') \cap F \neq \emptyset$$

and $F \subseteq S \times Q$ is a set of final states. We define the Büchi game automaton $\mathcal{A} = ((S \times Q)\dot{\cup}\{q_0, q_{sink}\}, Q, q_0, \delta, F)$ where $S \times Q$ and F are given by the game reduction. Additionally, we introduce the initial state q_0 and the sink state q_{sink} both of which are declared non-final. δ is defined according to Definition 3.5. Analogous to φ' we define for any run ρ of $\mathcal{A}$:

$$\rho = q_0\rho' \text{ is accepting} :\Longleftrightarrow Inf(\rho) \cap F \neq \emptyset$$

where ρ' is the corresponding play in Γ'. We show that $\mathcal{A}$ satisfies the acceptance condition of a game automaton. Let $\rho = q_0(s_1, q_1)(s_2, q_2)\ldots$ be a regular run of $\mathcal{A}$. Then, by definition

$$\begin{aligned} \rho = q_0\rho' \text{ is accepting in } \mathcal{A} \quad &\Longleftrightarrow \quad Inf(\rho) \cap F \neq \emptyset \\ &\Longleftrightarrow \quad Inf(\rho') \cap F \neq \emptyset \\ &\Longleftrightarrow \quad \rho' = (s_1, q_1)(s_2, q_2)\ldots \in \text{ Win}_0 \end{aligned}$$

If ρ is non-regular, i.e. $\mathcal{A}$ finally assumes q_{sink}, then ρ is rejecting because $\mathcal{A}$ stays in q_{sink} until infinity (cf. definition of δ_4) and q_{sink} is non-final. We conclude that the automaton $\mathcal{A}$ satisfies Definition 3.5. From Remark 3.7 we know that $\mathcal{A}$ accepts the language $L_0(\Gamma)$, the winning plays for Player 0 in Γ.

In Section 4.1.2 we have already seen that for a Büchi game automaton $\mathcal{A}$ the following holds:

$$L(\mathcal{A}/_{\simeq_{de}}) = L(\mathcal{A}) = L(cl(\mathcal{A})) = L(cl(\mathcal{A})/_{\simeq_{di}}) = L(cl(\mathcal{A})/_{\approx_{di}})$$

We utilise this equivalence to proceed from $\mathcal{A}$ to $cl(\mathcal{A})$. We compute the quotient automaton $cl(\mathcal{A})/_{\approx_{di}}$ and use it for our algorithm. To be able to define it we need two more things. Firstly, we have to show that $\approx_{di}$ is compatible with $cl(\mathcal{A})$. Secondly, we have to give an acceptance condition $F/_{\approx_{di}}$ and prove that it satisfies Definition 3.10. As acceptance condition for $cl(\mathcal{A})/_{\approx_{di}}$ we choose:

$$[(s', q')] \in F/_{\approx_{di}} :\Longleftrightarrow \exists (s, q) \in F : (s, q) \approx_{di} (s', q')$$

By the definition of direct bisimulation (cf. page 52) we get that only final states and only non-final states can be equivalent. This yields:

$$[(s', q')] \in F/_{\approx_{di}} \quad \Longleftrightarrow \quad \forall (s, q) \text{ with } (s, q) \approx_{di} (s', q') : (s, q) \in F$$

Lemma 4.11. *Let $\mathcal{A}$ be a Büchi game automaton and $\approx_{di}$ be the direct bisimulation relation on $S \times Q$. Then $\approx_{di}$ is compatible with $cl(\mathcal{A})$.*

Proof. To prove item 1. of Definition 3.9 take $(s_1, q_1), (s'_1, q'_1) \in S \times Q$ and consider the run $\rho = (s_1, q_1)(s_2, q_2)(s_3, q_3)\ldots$ of $cl(\mathcal{A})_{(s_1,q_1)}$[1] on some input word $\alpha = q_2 q_3 \ldots$. Since $cl(\mathcal{A})$ is deterministic ρ uniquely determines the run $\rho' = (s'_1, q'_1)(s'_2, q'_2)(s'_3, q'_3)$ of $cl(\mathcal{A})_{(s'_1,q'_1)}$ on the same α. We can assume w.l.o.g. that $q_i = q'_i$ for all $i \geq 1$. We need to show:

[1]Note that $cl(\mathcal{A}_{(s_1,q_1)}) = cl(\mathcal{A})_{(s_1,q_1)}$

$$(s_2, q_2) \not\approx_{di} (s_2', q_2') \Longrightarrow (s_1, q_1) \not\approx_{di} (s_1', q_1')$$

According to the definition of $\approx_{di}$ any equivalence class of $S \times Q/_{\approx_{di}}$ consists only of final states or only of non-final states. Hence, the premise of the above implication means

$$\exists i : i \geq 2 \text{ and } (\rho(i) \in F \iff \rho'(i) \notin F).$$

But, this implies

$$\exists i : i \geq 1 \text{ and } (\rho(i) \in F \iff \rho'(i) \notin F)$$

which (by definition of $\approx_{di}$) means that $(s_1, q_1) \not\approx_{di} (s_1', q_1')$.

For proving item 2. we consider two runs $\rho = q_0(s_1, q_1)(s_2, q_2)\dots$ and $\rho' = q_0(s_1', q_1')(s_2', q_2')\dots$ of $cl(\mathcal{A})$ such that $(s_i, q_i) \approx_{di} (s_i', q_i')$ for all $i > 0$. By symmetry it suffices to show that if ρ is accepting then ρ' is accepting as well. If ρ is accepting then there are infinitely many i such that $(s_i, q_i) \in F$. By the remarks prior to this Lemma for exactly the same indices as for ρ the states (s_i', q_i') are final. Hence, there are also infinitely many i such that $(s_i', q_i') \in F$ and accordingly ρ' is accepting. This completes the proof of item 2. and we are done. □

Lemma 4.12. *Let $\mathcal{A}$ be a Büchi game automaton. Let further $cl(\mathcal{A})/_{\approx_{di}}$ be denoted as in Section 4.1.2 and $F/_{\approx_{di}}$ be defined as above. Then $cl(\mathcal{A})/_{\approx_{di}}$ satisfies Definition 3.10.*

Proof. Let $cl(\mathcal{A})/_{\approx_{di}}$ have the set of final states as in Lemma 4.12 and let $\rho' = q_0[(s_1', q_1')][(s_2', q_2')]\dots$ be a run of $cl(\mathcal{A})/_{\approx_{di}}$. Let $\mathcal{R}$ denote the set of all runs $\rho = q_0(s_1, q_1)(s_2, q_2)\dots$ of $cl(\mathcal{A})$ with $(s_i, q_i) \approx_{di} (s_i', q_i')$ for all $i > 0$. We show that ρ' is accepting if and only if there exists an accepting run $\rho \in \mathcal{R}$:

$$\begin{array}{lcl}
\rho' \text{ accepting} & \iff & \exists^\omega i : [(s_i', q_i')] \in F/_{\approx_{di}} \\
& \overset{\text{p. } 55}{\iff} & \exists^\omega i : \forall (s_i, q_i) \text{ with } (s_i, q_i) \approx_{di} (s_i', q_i') : (s_i, q_i) \in F \\
& \iff & \exists^\omega i : \forall \rho (\rho \in \mathcal{R} \Longrightarrow \rho(i) \in F) \\
& \iff & \forall \rho (\rho \in \mathcal{R} \Longrightarrow \rho \text{ accepting}) \\
& \overset{\approx_{di} \text{ comp.}}{\iff} & \exists \rho (\rho \in \mathcal{R} \text{ and } \rho \text{ accepting})
\end{array}$$

□

By what we know from Section 4.1.2 the Büchi game automata $\mathcal{A}$ and $cl(\mathcal{A})$ are equivalent. Together with the above results and Remark 3.11 this yields the equivalence of $\mathcal{A}$ and $cl(\mathcal{A})/_{\approx_{di}}$. Since we are searching for the quotient automaton $cl(\mathcal{A})/_{\approx_S}$ we need to define an acceptance condition satisfying Definition 3.12. We choose

$$([s], q) \in F/_{\approx_S} :\iff [(s, q)] \in F/_{\approx_{di}}$$

Note that $F/_{\approx_S}$ is well-defined because for $s_1, s_2 \in S$ with $s_1 \approx_S s_2$ we have $[(s_1, q)] = [(s_2, q)]$. We show that with the above definition for $F/_{\approx_S}$ the automaton $cl(\mathcal{A})/_{\approx_S}$ satisfies Definition 3.12.

Lemma 4.13. *Let $\mathcal{A}$ be a Büchi game automaton. Let further $\approx_S$ denote the equivalence relation from Definition 3.12 where $\approx$ is replaced by $\approx_{di}$. If $F/_{\approx_S}$ is defined as above then $cl(\mathcal{A})/_{\approx_S}$ satisfies Definition 3.12.*

Proof. We only have to verify the acceptance condition of $cl(\mathcal{A})/_{\approx_S}$. Consider the run $\rho' = q_0([s_1], q_1)([s_2], q_2)\dots$ of $cl(\mathcal{A})/_{\approx_S}$ and the corresponding run $\rho = q_0[(s_1, q_1)][(s_2, q_2)]\dots$ of $cl(\mathcal{A})/_{\approx_{di}}$. We have to show that ρ' is accepting if and only if ρ is accepting:

$$\begin{aligned} \rho' \text{ accepting} &\iff \exists^\omega i : ([s_i], q_i) \in F/_{\approx_S} \\ &\overset{\text{Def. } F/_{\approx_S}}{\iff} \exists^\omega i : [(s_i, q_i)] \in F/_{\approx_{di}} \\ &\iff \rho \text{ accepting} \end{aligned}$$

□

4.2 Minimisation of Deterministic Weak Büchi Automata

Deterministic weak Büchi automata (DWA) form a subclass of deterministic Büchi automata (DBA). We introduce DWA separately from general Büchi automata because of a simple reason. DWA are very specific and we use them to reduce game graphs for weak parity games but not for Büchi games. First we introduce DWA and present an algorithmic approach to minimise them. In Section 4.2.1 we explain why and how we can use this algorithm to reduce the size of a game automaton of a reduced weak parity game. This is due to the fact that the "good" states of both DWA and reduced weak parity game automata are uniformly spread among the strongly connected components (SCCs). This means that acceptance only depends on the SCCs assumed in a run but not on the individual states within these components. To get this clearer let us define DWA and colourings for them.

Definition 4.14. Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a (deterministic) Büchi automaton. We call a state $q \in Q$ *recurrent* if there exists a word $w \in \Sigma^* \setminus \epsilon$ such that $\delta(q, w) = q$. Otherwise q is called *transient*. A *strongly connected component* (SCC) is a maximal subset $S \subseteq Q$ such that each state in S is reachable from every other state in S. A SCC is called transient if it has only one state which is then transient itself. Otherwise a SCC is called recurrent. A DBA $\mathcal{A}$ is called *weak* if and only if every SCC of $\mathcal{A}$ contains only final states or only non-final states. According to this we call a SCC final or non-final.

In Lemma 4.19 we show that a given DWA $\mathcal{A}$ can be reduced with a minimisation algorithm for standard DFA. However, to be sure that the obtained DWA is minimal we require $\mathcal{A}$ to be of a special format, i.e. in normal form (cf. Definition 4.17). The first step towards a normalised DWA is to express its acceptance condition by means of a colouring.

Definition 4.15. Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DWA and $k \in \mathbb{N}$. A mapping $c : Q \to \{0, \ldots, k\}$ is called an *$\mathcal{A}$-colouring* if and only if the following hold:

1. $c(q)$ is even for every recurrent state $q \in F$
2. $c(q)$ is odd for every recurrent state $q \notin F$
3. $c(p) \leq c(q)$ for every $p, q \in Q$ with $\delta(p, a) = q$ for some $a \in \Sigma$

A colouring c is called *k-maximal* if and only if the following hold:

4. $c'(q) \leq c(q)$ for every $\mathcal{A}$-colouring $c' : Q \to \{0, \ldots, k\}$, $q \in Q$
5. $c(q) \leq k$ for every $q \in Q$

A colouring c is called *maximal* if it is k-maximal for some $k \in \mathbb{N}$.

A colouring c for a DWA $\mathcal{A}$ normally needs not be maximal. Nevertheless, we can use it to express the acceptance condition of $\mathcal{A}$ in terms of a weak parity acceptance condition.

Remark 4.16. Let $\mathcal{A}$ be a DWA and let c be a colouring for $\mathcal{A}$. Then for any run ρ of $\mathcal{A}$ we have

$$\rho \text{ is accepting if and only if } \max(Occ(c(\rho))) \text{ is even.}$$

Proof. For the proof we consider the sequence $c(\rho)$ for a run ρ of $\mathcal{A}$:

$$\begin{array}{lcl}
\rho \text{ is accepting} & \overset{\text{Def.}}{\iff} & \mathcal{A} \text{ on } \rho \text{ visits infinitely many final states} \\
& \overset{\mathcal{B}\text{ weak}}{\iff} & \mathcal{A} \text{ on } \rho \text{ eventually assumes a final SCC } S \\
& & \text{and stays in it until infinity} \\
& \overset{\text{Def. 4.15}}{\iff} & \mathcal{A} \text{ on } \rho \text{ eventually assumes an SCC } S \\
& & \text{where all } s \in S \text{ have an even colour } c(s) \\
& \overset{c(\rho)\text{ weakly inc.}}{\iff} & \max(Occ(c(\rho))) \text{ is even}
\end{array}$$

□

There are many possible colourings for the same DWA depending on the choice of k. If we choose k higher and higher then we can always find new maximal colourings by just adding an appropriate constant to the values of a given maximal colouring c. Thus, we are not interested in the precise value of k but in the fact which states have even colours and which have odd colours.

Definition 4.17. For an $\mathcal{A}$-colouring c we define F_c to be the set of states which get an even colour, i.e. $F_c = \{q \mid c(q) \text{ is even}\}$. $\mathcal{A}$ is said to be in *normal form* if and only if $F = F_c$ for some k-maximal $\mathcal{A}$-colouring c with k even.

Our aim is to show that from a given DWA one can efficiently compute an equivalent minimal DWA by means of algorithms for DFA minimisation. To start out we want to prove two things. Firstly, we show that DFA minimisation is correct even for the case of DWA, i.e. if we apply it to a given DWA then we get an equivalent DWA. Secondly, minimisation algorithms for DFA do not in general yield a minimal result for the DWA. But if the given DWA is in normal form then this is truly the case. Let us first show that DFA minimisation on a DWA is correct. For that it is helpful to introduce the notion of a limit language.

Definition 4.18. Let Σ be an alphabet and $U \subseteq \Sigma^*$ be a regular language. By $lim(U)$ we denote the set of infinite sequences over Σ which have infinitely many prefixes in U.

$$lim(U) = \{\alpha \in \Sigma^\omega \mid \exists^\omega i : \alpha(0) \dots \alpha(i) \in U\}$$

Lemma 4.19. *Let $\mathcal{A}$ be a DWA and let $\mathcal{A}^{min}$ be the minimal DFA equivalent to the DFA $\mathcal{A}$. Then $L_\omega(\mathcal{A}) = L_\omega(\mathcal{A}^{min})$.*

Proof. We show the claim by set inclusion in both directions. Let $w \in L_\omega(\mathcal{A})$. Since $\mathcal{A}$ assumes F on w again and again it is clear from Definition 4.18 that $w \in lim(L_*(\mathcal{A}))$. Since $\mathcal{A}$ taken as DFA is equivalent to $\mathcal{A}^{min}$ it follws that $w \in lim(L_*(\mathcal{A}^{min}))$. Again by Definition 4.18 we get $w \in L_\omega(\mathcal{A}^{min})$. For the other direction one can use the reverse implications. □

We have just proven the correctness of a DFA minimisation algorithm when being applied to a DWA. The question is now whether the resulting automaton is a minimal DWA as it is for DFA. This is not the case in general but if we transform the given DWA into normal form we get a minimal result. To show this we have to do some preparations. First we show that two given states of a DWA from where exactly the same words are accepted must have the same colour.

Lemma 4.20. *Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DWA and let c be a k-maximal $\mathcal{A}$-colouring for some $k \in \mathbb{N}$. For all $p, q \in Q$ we have:*

$$L_\omega(\mathcal{A}_p) = L_\omega(\mathcal{A}_q) \Longrightarrow c(p) = c(q)$$

Proof. The proof is tedious and takes nearly two pages but we cannot leave it out. For a contradiction, assume there are $p, q \in Q$ such that $L_\omega(\mathcal{A}_p) = L_\omega(\mathcal{A}_q)$ and $c(p) \neq c(q)$. From all such pairs of states choose p, q such that $c(p)+c(q)$ is maximal and $c(p) < c(q)$. We show that $c(q) = c(p) + 1$. Assume $c(p) > k - 2$. Since $c(p) < c(q)$ this implies $c(p) = k - 1$ and $c(q) = k$. Hence, $c(q) = c(p) + 1$ holds. Now, assume that $c(p) \leq k - 2$. From every such p there is a state reachable with colour $c(p) + 1$. This claim follows intuitively from the fact that c is maximal.

Proposition 4.21. *Let $\mathcal{A}$ and c be as above. For every $p \in Q$ with $c(p) \leq k - 2$ there is a state $q \in Q$ reachable from p with $c(q) = c(p) + 1$.*

Proof. For a contradiction, assume that $p \in Q$ does not satisfy this property. We set $P := \{q \mid q \text{ is reachable from p and } c(p) = c(q)\}$. Note that $p \in P$ and hence $P \neq \emptyset$. We define a new $\mathcal{A}$-colouring $c' : (Q \setminus P)\dot{\cup}P \to \{0, \dots, k\}$ as follows:

$$\begin{aligned} c'(q) &:= c(q) &&, \text{ if } q \in Q \setminus P \\ c'(q) &:= c(q) + 2 &&, \text{ if } q \in P \end{aligned}$$

We obtain $c(p) < c'(p)$ contradicting the k-maximality of c. □

With this result we can resume the proof of Lemma 4.20. We now know that from every $p \in Q$ with $c(p) \leq k - 2$ there is a state q reachable with $c(q) = c(p) + 1$. Let this state reachable from p be called r and let $w \in \Sigma^*$ be such that $\delta(p, w) = r$. Analogously, set $s := \delta(q, w)$. From $L_\omega(\mathcal{A}_p) = L_\omega(\mathcal{A}_q)$ we deduce $L_\omega(\mathcal{A}_r) = L_\omega(\mathcal{A}_s)$. Assume w.l.o.g. there was a word α such that $\alpha \in L_\omega(\mathcal{A}_r)$ and $\alpha \notin L_\omega(\mathcal{A}_s)$. Then $w\alpha \in L_\omega(\mathcal{A}_p)$ and $w\alpha \notin L_\omega(\mathcal{A}_q)$ contradicting the assumption that the two latter languages are equal. Since c is a colouring we have $c(p) \leq c(r)$ and $c(q) \leq c(s)$. From that and our initial assumption that $c(p) + c(q)$ is maximal such that $c(p) < c(q)$ it follows that $c(r) = c(s)$. Otherwise r and s must have been chosen in place of p and q. To summarise what we know already, we have $c(r) = c(p) + 1$ and $c(p) < c(q)$. This yields $c(r) \leq c(q)$, altogether $c(r) \leq c(q) \leq c(s)$. Since we already know $c(r) = c(s)$ this implies $c(q) = c(r)$ and since $c(r) = c(p) + 1$ we get $c(q) = c(p) + 1$. So, in both cases we get that $c(q) = c(p) + 1$.

From this we derive a contradiction. Before we can proceed we need the following result saying that from every state q we can find a run through $\mathcal{A}$ that does only visit states of the same colour, namely that of q.

Proposition 4.22. *Let $\mathcal{A}$ and c be as above. For every $q \in Q$ there is an $\alpha \in \Sigma^\omega$ with run ρ of $\mathcal{A}_q$ on α such that* $\max(Occ(c(\rho))) = c(q)$.

Proof. If q is recurrent then we can find a non-empty cycle labelled by, say $w \in \Sigma^* \setminus \epsilon$, from q back to q such that all states on this cycle have the same colour. Hence, $\alpha = w^\omega$ shows the claim. So, let q be transient. If the claim does not hold then there was no recurrent state q' reachable from q that has the same colour as q. Because otherwise there is a word $w' \in \Sigma^* \setminus \epsilon$ to get from q to q' and by the first argument we find a cycle labelled by w'' from q' back to q'. But then the word $\alpha = w'w''^\omega$ would satisfy the condition. So, we know that if q is transient there is no recurrent state reachable from q with the same colour as q. But if there is no recurrent state reachable from q that has the same colour as q then we can define a new colouring c':

$$\begin{aligned} c'(r) &:= c(r) + 1 &&, \text{ if } r \text{ is reachable from } q \text{ and } c(q) = c(r) \\ c'(r) &:= c(r) &&, \text{ otherwise} \end{aligned}$$

Note that at least for state q the c'-value is higher than the c-value and that c' is a k-maximal $\mathcal{A}$-colouring[2]. This contradicts the assumption that c is k-maximal. □

[2]W.l.o.g. any state not satisfying the condition claimed can have a c-value of at most $k - 1$. Thus, $c'(q) \leq k$ for every $q \in Q$.

Using the last result, let $\alpha \in \Sigma^\omega$ be such that $c(p) = \max(Occ(c(\rho)))$ for the run ρ of $\mathcal{A}_p$ on α. Consider the run ρ' of $\mathcal{A}_q$ on α. If in $c(\rho')$ a colour greater than $c(q)$ occurs, let $w \in \Sigma^* \setminus \epsilon$ be a prefix of α such that $c(\delta(q,w)) > c(q)$. Since w is a prefix of α we have $c(\delta(p,w)) = c(p)$. Set $r := \delta(p,w)$ and $s := \delta(q,w)$. By the same argument as above we can deduce that $L_\omega(\mathcal{A}_r) = L_\omega(\mathcal{A}_s)$. Additionally, we have $c(s) > c(q) = c(p) + 1 > c(p) = c(r)$ which yields $c(s) \neq c(r)$. We get that $c(s) + c(r) > c(p) + c(q)$ which together with the fact $L_\omega(\mathcal{A}_r) = L_\omega(\mathcal{A}_s)$ again contradicts our assumption about the maximality of $c(p) + c(q)$. Therefore, $c(p) = \max(Occ(c(\rho)))$ and $c(q) = \max(Occ(c(\rho')))$ and since $c(q) = c(p) + 1$ we get that $\max(Occ(c(\rho)))$ is even if and only if $\max(Occ(c(\rho')))$ is odd. By Remark 4.16 we can conclude that $\alpha \in L_\omega(\mathcal{A}_p)$ if and only if $\alpha \notin L_\omega(\mathcal{A}_q)$. This is a contradiction to our primal assumption about the equality of $L_\omega(\mathcal{A}_p)$ and $L_\omega(\mathcal{A}_q)$ and completes the proof for Lemma 4.20. □

We can now do the actual proof for showing that a DWA in normal form can be minimised by algorithms for DFA minimisation.

Lemma 4.23. *Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DWA in normal form. If $L_*(\mathcal{A}_p) \neq L_*(\mathcal{A}_q)$ for some $p, q \in Q$ then $L_\omega(\mathcal{A}_p) \neq L_\omega(\mathcal{A}_q)$.*

Proof. Since $\mathcal{A}$ is in normal form there is a maximal $\mathcal{A}$-colouring c with $F_c = F$. Let $p, q \in Q$ such that $L_*(\mathcal{A}_p) \neq L_*(\mathcal{A}_q)$. This means that w.l.o.g. there is a word $w \in \Sigma^*$ such that $\delta(p,w) \in F$ but $\delta(q,w) \notin F$. Set $r := \delta(p,w)$ and $s := \delta(q,w)$. Since $F_c = F$ this implies that $c(r)$ is even and $c(s)$ is odd, thus $c(r) \neq c(s)$. By Lemma 4.20 we can conclude that $L_\omega(\mathcal{A}_r) \neq L_\omega(\mathcal{A}_s)$. Choose $\alpha \in \Sigma^\omega$ such that $\alpha \in L_\omega(\mathcal{A}_r)$ if and only if $\alpha \notin L_\omega(\mathcal{A}_s)$. Concatenating w and α we get $w\alpha \in L_\omega(\mathcal{A}_p)$ if and only if $w\alpha \notin L_\omega(\mathcal{A}_q)$ which yields $L_\omega(\mathcal{A}_p) \neq L_\omega(\mathcal{A}_q)$. □

We need to show that what we know for DFA can be analogously assumed for DWA, namely that an automaton is minimal if we have $L_\omega(\mathcal{A}_p) \neq L_\omega(\mathcal{A}_q)$ for all pairs of distinct states $p, q \in Q$. For that we introduce a congruence just like the Nerode congruence over finite words.

Definition 4.24. Let $L \subseteq \Sigma^\omega$ be a language. We define the right congruence $\sim_L \subseteq \Sigma^* \times \Sigma^*$ as follows:

$$u \sim_L v :\Longleftrightarrow \forall \alpha \in \Sigma^\omega (u\alpha \in L \Longleftrightarrow v\alpha \in L)$$

As usual we denote by $[u]_L$ the $\sim_L$-class of $u \in \Sigma^*$ and by *index*$(\sim_L)$ the index of $\sim_L$, i.e. the number of $\sim_L$-classes. Minimality of a DWA refers to the fact that there is no DWA with less states recognising the same language. We now show that each DWA recognising a language L needs at least *index*$(\sim_L)$ states and that a given DWA is minimal if all pairs of its states are non-equivalent. Two states p and q are said to be non-equivalent if $L_\omega(\mathcal{A}_p) \neq L_\omega(\mathcal{A}_q)$.

Lemma 4.25. *Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DWA and let $L = L_\omega(\mathcal{A})$. For every reachable state $q \in Q$ there is a word $u_q \in \Sigma^*$ such that $\delta(q_0, u) = q$ implies $u \in [u_q]_L$*

for all $u \in \Sigma^*$. We have $|Q| \geq index(\sim_L)$ and if $L_\omega(\mathcal{A}_p) \neq L_\omega(\mathcal{A}_q)$ for all $p, q \in Q$ then $\mathcal{A}$ is a minimal DWA.

Proof. Let $q \in Q$ be reachable and let $\delta(q_0, u) = \delta(q_0, v) = q$. If we set $u_q := u$ then to prove our first claim it suffices to show that $u \sim_L v$. Let $\alpha \in \Sigma^\omega$. The runs on $u\alpha$ and $v\alpha$ differ only in their first part up to state q. However, since we have a DWA acceptance only depends on the infinity sets of both runs which are equal in our case. Hence, $\mathcal{A}$ accepts $u\alpha$ if and only if $\mathcal{A}$ accepts $v\alpha$. Furthermore, assume $|Q| < index(\sim_L)$. Then there exist two words $w_1, w_2 \in \Sigma^*$ such that $q' := \delta(q_0, w_1) = \delta(q_0, w_2)$ and $w_1 \nsim_L w_2$. Then we know that there exists a word $\alpha \in \Sigma^\omega$ such that w.l.o.g. $w_1\alpha \in L$ and $w_2\alpha \notin L$. By the same argument as above this is a contradiction since both runs have the same infinity sets and, accordingly, both $w_1\alpha$ and $w_2\alpha$ are accepted or not accepted. From this we can also conclude that a DWA where all pairs of states are non-equivalent is minimal. □

From Lemmas 4.23 and 4.25 we can deduce that a DFA minimisation algorithm applied to a DWA in normal form we get a minimal DWA. Thus, what is left to be done is to give an algorithm that computes for a given DWA $\mathcal{A}$ its normal form $\mathcal{A}'$.

Lemma 4.26. *Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DWA. Then there exists a set $F' \subseteq Q$ of final states such that $\mathcal{A}' = (Q, \Sigma, q_0, \delta, F')$ is in normal form and equivalent to $\mathcal{A}$.*

Proof. The main part of the proof is to find a k-maximal colouring for $\mathcal{A}$. Let n be the number of states of $\mathcal{A}$, i.e. $n := |Q|$. Then, we can choose n as a suitable value for k to be sure k is large enough. From part three of Definition 4.15 we can conclude that all states within the same SCC have the same colour. Thus, we can reduce the problem of finding an $\mathcal{A}$-colouring to finding a colouring of the SCC-graph of $\mathcal{A}$. For computing the SCC-graph of $\mathcal{A}$ we take a common SCC detection algorithm from [Sed91]. We get the SCC-graph $G = (V, E)$ where $V = \{1, \ldots, m\}$ if $Q_1, \ldots, Q_m$ are the SCCs of $\mathcal{A}$[3]. Moreover, we have $(i, j) \in E$ if and only if $i \neq j$ and there are states $q_i \in Q_i$ and $q_j \in Q_j$ such that there exists some $a \in \Sigma$ with $\delta(q_i, a) = q_j$. Moreover, we can assume that the transient SCCs are marked. To find a maximal colouring for $\mathcal{A}$ we need to compute a mapping $d : V \to \{0, \ldots, k\}$. The demands on the k-maximal colouring c can be equivalently formulated for d:

1. $d(i)$ even if Q_i is recurrent and final
2. $d(i)$ odd if Q_i is recurrent and non-final
3. $d(i) \leq d(j)$ if $(i, j) \in E$

Additionally, all d-values need to be maximal with the above properties. From that we obtain a k-maximal colouring for $\mathcal{A}$ by setting

$$c(q) = j :\Longleftrightarrow d(Q_i) = j \text{ and } q \in Q_i.$$

[3] i is identified with the SCC Q_i.

Note that the SCC-graph for $\mathcal{A}$ is acyclic. Hence, V is partially ordered by E and we can topologically sort it (cf. [Sed91]). This means we can compute a permutation $i_1, \ldots, i_m$ of $1, \ldots, m$ such that $(i_k, i_l) \in E$ implies $k < l$. The following algorithm computes the mapping d:

Algorithm 4.27. (Maximal Colouring)
Input: SCC-graph G of a DWA $\mathcal{A}$, permutation $i_1, \ldots, i_m$
Output: Maximal colouring for G

```
01  for $j := m$ downto 1 do
02    if $succ(i_j) = \emptyset$ [Initialise $d(i_j)$ with maximal possible value]
03      then if $i_j$ is final
04              then $d(v_j) := k$
05              else $d(v_j) := k - 1$
06      else [Compute $d(i_j)$ with help of successor values]
07        $l := \min\{d(k) \mid k \in succ(v_j)\}$
08        if $v_j$ is transient
09          then $d(v_j) := l$
10          else if $l$ is even and $v_j$ is final
11                  then $d(v_j) := l$
12                  else if $l$ is odd and $v_j$ is not final
13                          then $d(v_j) := l$
14                          else $d(v_j) := l - 1$
```

Since the permutation $i_1, \ldots, i_m$ is topologically sorted all d-values of the successors of i_j are defined before $d(i_j)$ is computed. The values for the SCCs without successors are maximal, namely k or $k-1$. Thus, by lines 06 to 14 the values for all SCCs are maximal. From d we compute the colouring c as described above. F_c is then defined to be the set of all states with an even colour. For the automaton $\mathcal{A}'$ we choose $F' = F_c$. By definition of colourings $\mathcal{A}$ and $\mathcal{A}'$ are equivalent. □

We have seen that DFA minimisation algorithms can be used to minimise DWA. This implies similar complexities for both problems. As a last result in this chapter we see that in DWA minimisation we obtain a unique result just as in DFA minimisation.

Theorem 4.28. *Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DWA with n states. Then one can compute an equivalent minimal DWA $\mathcal{A}'^{min}$ in time $O(n \cdot \log(n))$. $\mathcal{A}'^{min}$ is uniquely determined up to isomorphism.*

Proof. The first step is to transform $\mathcal{A}$ into a DWA in normal form, say $\mathcal{A}'$. For this we use Algorithm 4.27. The SCC-graph of $\mathcal{A}$ and the permutation $i_1, \ldots, i_m$ can be computed in linear time (cf. [Sed91]). Moreover, the size of G is bounded by the size of $\mathcal{A}$. Since $\mathcal{A}$ is deterministic it has $n \cdot |\Sigma|$ transitions. This yields $|V| + |E| \leq n + n \cdot |\Sigma|$ for the size of G. For at most m computations of l in line 7 each edge of G is considered once which yields the complexity $O(|E|)$. Since $|E|$

is bounded by $n \cdot |\Sigma|$ and the **if**-statements take constant time we get $O(n)$ as time complexity[4] for Algorithm 4.27. The computation of c and the set F_c can be derived from d in time $O(n)$. We obtain the DWA $\mathcal{A}' = (Q, \Sigma, q_0, \delta, F_c)$ in normal form and by Lemma 4.26 $\mathcal{A}$ and $\mathcal{A}'$ are equivalent. Then we apply a DFA minimisation algorithm to $\mathcal{A}'$ and obtain $\mathcal{A}'^{min}$ which is equivalent to $\mathcal{A}'$ by Lemma 4.19. This can be done in time $O(n \cdot \log(n))$ (cf. e.g. [Hop71]). Note that there is a homomorphism from $\mathcal{A}$ to $\mathcal{A}'^{min}$ preserving final and non-final states. Final (non-final) SCCs from $\mathcal{A}$ remain final (non-final) in $\mathcal{A}'^{min}$. Hence, $\mathcal{A}'^{min}$ is a DWA as well. By Lemmas 4.23 and 4.25 and our other results $\mathcal{A}'^{min}$ is an equivalent minimal DWA for $\mathcal{A}$. As time complexity we obtain $O(n) + O(n) + O(n \cdot \log(n))$ which is in $O(n \cdot \log(n))$.

To show that $\mathcal{A}'^{min}$ is uniquely determined by $L_\omega(\mathcal{A})$ (up to isomorphism) we show that the ω-language recognised by it only depends on the $*$-language recognised by the DFA $\mathcal{A}'^{min}$.

Lemma 4.29. *Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ and $\mathcal{A}' = (Q', \Sigma, q_0', \delta', F')$ be two DWA in normal form. If $L_\omega(\mathcal{A}) = L_\omega(\mathcal{A}')$ then $L_*(\mathcal{A}) = L_*(\mathcal{A}')$.*

Proof. Let c be a l-maximal $\mathcal{A}$-colouring and c' be a l'-maximal $\mathcal{A}'$-colouring such that $F = F_c$ and $F' = F_{c'}$. Since $\mathcal{A}$ and $\mathcal{A}'$ are in normal form l and l' are even. W.l.o.g. we can assume that both c and c' are k-maximal for $k := \max\{l, l'\}$. Otherwise we can add an appropriate constant to either the c-values (if $k = l'$) or to the c'-values (if $k = l$) and thereby obtain two k-maximal colourings. Our aim is to show that c and c' are actually the same functions, i.e. $c = c'$.

Take an arbitrary $u \in \Sigma^*$ and consider the states q_u and q_u' reached in $\mathcal{A}$ from q_0 and in $\mathcal{A}'$ from q_0' by reading u, i.e. $q_u = \delta(q_0, u)$ and $q_u' = \delta'(q_0', u)$. For a contradiction, assume there exists a word $u \in \Sigma^*$ with $c(q_u) \neq c'(q_u')$. We consider the case $c(q_u) < c'(q_u')$ and from this deduce a contradiction which yields the equality of c and c'. We define a new colouring $d : Q \to \{0, \ldots, k\}$ by

$$d(q_u) := \max\{c(q_u), c'(q_u')\} \text{ for each } u \in \Sigma^*.$$

Due to Lemmas 4.20 and 4.25 this definition is independent of the representative $v \in \Sigma^*$. That means, if we have $q_v = q_w$ then v and w are $L_\omega(\mathcal{A})$-equivalent and since $L_\omega(\mathcal{A}) = L_\omega(\mathcal{A}')$ also $L_\omega(\mathcal{A}')$-equivalent. Hence, $L_\omega(\mathcal{A}'_{q_v'}) = L_\omega(\mathcal{A}'_{q_w'})$ which according to Lemma 4.20 yields the same colours for q_v' and q_w'. Since we have assumed $c(q_u) < c'(q_u')$ we get $c(q_u) < d(q_u)$. Thus, if we can show that d is indeed an $\mathcal{A}$-colouring then we have a contradiction to the k-maximality of c. From the definition of d and the fact that c and c' are colourings it is clear that for any $q \in Q$ and $a \in \Sigma$ the inequality $d(q) \leq d(\delta(q, a))$ holds. Thus, we have item 3. of Definition 4.15 satisfied. What remains to be checked is the d-values of recurrent states. Let $q \in F$ be such a state. Then there are words $v, w \in \Sigma^*$ such that $\delta(q_0, v) = q$ and $\delta(q, w) = q$. Since q is a recurrent final state we have $c(q)$ even. Hence, if $d(q)$ is odd then $d(q) \neq c(q)$ and, consequently, $d(q) = c'(q_v')$. Consider the run of $\mathcal{A}$ on $\alpha := vw^\omega$. Due to Lemma 4.25 any of the infinitely

[4] $|\Sigma|$ is assumed to be a constant.

many prefixes of α where $\mathcal{A}$ reaches state q is $\sim_{L_\omega(\mathcal{A})}$-equivalent to v. If we consider the run of $\mathcal{A}'$ on α then by the same argument as above $\mathcal{A}'$ visits infinitely often a state q' with $L_\omega(\mathcal{A}'_{q'}) = L_\omega(\mathcal{A}'_{q'_v})$. Lemma 4.20 yields $c'(q') = c'(q'_v)$. Since $\mathcal{A}'$ visits q' infinitely often its colour is the highest colour seen and since $d(q) = c'(q'_v)$ is odd by assumption $\mathcal{A}'$ rejects α. Since $\mathcal{A}$ accepts α this is a contradiction to $L_\omega(\mathcal{A}) = L_\omega(\mathcal{A}')$. Thus, $d(q)$ must be even and in an analogous way one can show that recurrent states from $Q \setminus F$ have odd d-values. Hence, d is an $\mathcal{A}$-colouring and we get the contradiction we aimed at. Thus, for all $u \in \Sigma^*$ we have $c(q_u) = c'(q'_u)$. Since both $\mathcal{A}$ and $\mathcal{A}'$ are in normal form this yields $q_u \in F \iff q'_u \in F'$. From that we conclude $L_*(\mathcal{A}) = L_*(\mathcal{A}')$. □

Since we already know that DFA minimisation yields an automaton which is uniquely determined (up to isomorphism) we can conclude that $\mathcal{A}'^{min}$ is uniquely determined by $L_*(\mathcal{A})$. From Lemma 4.29 we know that $L_*(\mathcal{A})$ only depends on $L_\omega(\mathcal{A})$. So, $\mathcal{A}'^{min}$ is uniquely determined (up to isomorphism) by $L_\omega(\mathcal{A})$. This completes our proof of Theorem 4.28. □

4.2.1 Reduction of Weak Parity Game Automata

In this Section we apply the results from Section 3.2 to weak parity game automata. For a given weak parity game we define the corresponding weak parity game automaton, show that Definition 3.5 is met and explain how to transform it into an equivalent DWA. After that we apply the minimisation algorithm from Section 4.2 to the obtained DWA. We give acceptance conditions $F/_\approx$ and $F/_{\approx_S}$ and show that they satisfy Definitions 3.10 and 3.12. The last step is a transformation back to a weak parity automaton.

Assume we are given the reduced weak parity game $\Gamma' = (G', \varphi')$ according to the algorithm from page 27. We have the reduced game graph $G' = (S \times Q, E')$ and the weak parity winning condition defined for a play ρ' of Γ':

$$\rho' \in \text{Win}_0 :\iff \max(Occ(c(\rho'))) \text{ even}$$

The colouring $c : S \times Q \to \{0, \dots, 2 \cdot |Q|\}$ is defined as given on page 27. We define the weak parity game automaton $\mathcal{A} = ((S \times Q) \dot\cup \{q_0, q_{sink}\}, Q, q_0, \delta, c)$ where $S \times Q$ and c are given by the game reduction. We introduce the initial state q_0 and the sink state q_{sink} and extend the domain of c:

$$c(q_0) := 0 \text{ and } c(q_{sink}) := 2 \cdot |Q| + 1.$$

δ is defined according to Definition 3.5 and for a run ρ of $\mathcal{A}$ we define:

$$\rho = q_0\rho' \text{ is accepting} :\iff \max(Occ(c(\rho))) \text{ is even}$$

where ρ' is the corresponding play in Γ'. Let $\rho = q_0(s_1, q_1)(s_2, q_2)\dots$ be a regular run of $\mathcal{A}$. We get:

$$\begin{aligned}\rho = q_0\rho' \text{ is accepting in } \mathcal{A} &\iff \max(Occ(c(\rho))) \text{ is even}\\ &\iff \max(Occ(c(\rho'))) \text{ is even}\\ &\iff \rho' = (s_1, q_1)(s_2, q_2)\ldots \in \text{ Win}_0\end{aligned}$$

If ρ is non-regular then $\mathcal{A}$ assumes q_{sink} after a finite number of steps and stays in it until infinity (cf. definition of δ_4). Since $c(q_{sink}) = 2 \cdot |Q| + 1$ is odd and the maximal possible value of c we get that $\max(Occ(c(\rho)))$ is odd. Accordingly, ρ is rejecting and $\mathcal{A}$ satisfies Definition 3.5. From Remark 3.7 we know that $\mathcal{A}$ accepts the language $L_0(\Gamma)$, the winning plays for Player 0 in Γ.

In Section 4.2 we have seen that DWA can be minimised efficiently, namely in time $O(n \cdot \log(n))$. In this Section we show that the minimisation algorithm for DWA can be utilised to reduce the state space of a weak parity game automaton. Moreover, we show that DWA and weak parity game automata are equivalent. This is due to two reasons. Firstly, both in a DWA[5] and in a weak parity game automaton all states in one and the same strongly connected component have the same colour. Secondly, for any run ρ the sequence of seen colours in ρ is weakly increasing. We start out explaining how a weak parity game automaton can be transformed into a DWA.

Lemma 4.30. *Let $\mathcal{A} = ((S \times Q)\dot\cup\{q_0, q_{sink}\}, Q, q_0, \delta, c)$ be a weak parity game automaton. Then $\mathcal{A}$ can be transformed in linear time into an equivalent DWA $\mathcal{B} = ((S \times Q)\dot\cup\{q_0, q_{sink}\}, Q, q_0, \delta, F)$ where F is defined to be the set of all[6] states with even colour:*

$$F := \{(s, q) \mid (s, q) \in S \times Q, c(s, q) \text{ even}\}\dot\cup\{q_0\}$$

Proof. Consider any strongly connected component C of $\mathcal{A}$. All states within C have the same colour. Accordingly, all of them are declared final or non-final in $\mathcal{B}$ and we have either $C \subseteq F$ or $C \cap F = \emptyset$. Accordingly, $\mathcal{B}$ is a DWA. Assume we are given the weak parity game automaton $\mathcal{A}$ with colouring c. Let $\rho = q_0(s_1, q_1)(s_2, q_2)\ldots$ be the accepting run of $\mathcal{A}$ on $\alpha = q_1q_2\ldots$ where $\mathcal{A}$ finally assumes the strongly connected component C which it never leaves. Since $\max(Occ(c(\rho)))$ is even all states of C have an even colour and, accordingly, all of them are declared final in $\mathcal{B}$. Since C is never left $\mathcal{B}$ from a certain point onwards visits only final states until infinity and accepts α. If we assume that ρ is not accepting then all states of C have an odd colour and are declared non-final in $\mathcal{B}$. Again, since C is never left from a certain point onwards only non-final states are visited until infinity and α is rejected. Hence, $\mathcal{A}$ and $\mathcal{B}$ accept the same language. Moreover, F can be computed from c in linear time. $\square$

The first step of the minimisation algorithm for DWA is to compute a maximal colouring. Obtaining a new set of final states we thereby transform the given automaton into normal form. We have shown that a DWA in normal form can be

[5]with a colouring defined for it

[6]We assume that $c(q_0) = 0$ and $c(q_{sink}) = 2 \cdot |Q| + 1$.

minimised by computing the equivalence relation $\approx_{\mathcal{A}}$ as known from minimisation of DFA:

$$(s,q) \approx_{\mathcal{A}} (s',q') :\Longleftrightarrow$$
$$\forall w \in \Sigma^* : \delta((s,q),w) \in F \Longleftrightarrow \delta((s',q'),w) \in F$$

We give a simple acceptance condition $F/_{\approx_{\mathcal{A}}}$ that satisfies Definition 3.10:

$$[(s',q')] \in F/_{\approx_{\mathcal{A}}} : \Longleftrightarrow \ \exists (s,q) : (s,q) \approx_{\mathcal{A}} (s',q') \text{ and } (s,q) \in F$$

To prove that the corresponding weak Büchi quotient game automaton is equivalent to the given automaton we first have to show that $\approx_{\mathcal{A}}$ is a compatible equivalence relation.

Lemma 4.31. *Let $\mathcal{A}$ be a weak Büchi game automaton and $\approx_{\mathcal{A}}$ the equivalence relation on $S \times Q$ as defined above. Then $\approx_{\mathcal{A}}$ is compatible with $\mathcal{A}$.*

Proof. To prove compatibility for $\approx_{\mathcal{A}}$ we have to go through items one and two from Definition 3.9. For the proof of the first item we utilise the definition of $\approx_{\mathcal{A}}$. Let $(s_1,q_1) \in S \times Q$ and $\rho = (s_1,q_1)(s_2,q_2)(s_3,q_3)\dots$ be the run of $\mathcal{A}_{(s_1,q_1)}$ on the input word $\alpha = q_2 q_3 \dots$. Since $\mathcal{A}$ is deterministic it uniquely determines for given (s'_1,q'_1) the run $\rho' = (s'_1,q'_1)(s'_2,q'_2)(s'_3,q'_3)\dots$ of $\mathcal{A}_{(s'_1,q'_1)}$ on the same α. We can assume w.l.o.g. that $q_i = q'_i$ for all $i \geq 1$ and show an equivalent of item 1.:

$$(s_2,q_2) \not\approx_{\mathcal{A}} (s'_2,q'_2) \Longrightarrow (s_1,q_1) \not\approx_{\mathcal{A}} (s'_1,q'_1)$$

If $(s_2,q_2) \not\approx_{\mathcal{A}} (s'_2,q'_2)$ then there exists $w \in \Sigma^*$ such that w.l.o.g. $\delta((s_2,q_2),w) \in F$ and $\delta((s'_2,q'_2),w) \notin F$. This implies $\delta((s_1,q_1),q_2 w) \in F$ and $\delta((s'_1,q'_1),q'_2 w) \notin F$. Since $q_2 = q'_2$ we get that there exists $w' \in \Sigma^*$, namely $w' := q_2 w = q'_2 w$, such that

$$\delta((s_1,q_1),w') \in F \text{ and } \delta((s'_1,q'_1),w') \notin F.$$

Hence, by definition of $\approx_{\mathcal{A}}$ this yields $(s_1,q_1) \not\approx_{\mathcal{A}} (s'_1,q'_1)$.

For proving item 2. we refer to the equivalence relation used. According to the definition of $\approx_{\mathcal{A}}$ two states can only be merged if both are final or both are non-final. Therefore, all states in an equivalence class of $\mathcal{A}/_{\approx_{\mathcal{A}}}$ are final in $\mathcal{A}$ or none is. Now, assume we are given two runs $\rho = q_0(s_1,q_1)(s_2,q_2)\dots$ and $\rho' = q_0(s'_1,q'_1)(s'_2,q'_2)\dots$ of $\mathcal{A}$ such that $(s_i,q_i) \approx_{\mathcal{A}} (s'_i,q'_i)$ for all $i > 0$. By the above remarks we get:

$$(s_i,q_i) \in F \iff (s'_i,q'_i) \in F$$

By symmetry, it suffices to show that if ρ is accepting then ρ' is accepting as well. If ρ is accepting then there are infinitely many i such that $(s_i,q_i) \in F$ and the above equivalence implies that for the same infinite set of indices the states (s'_i,q'_i) are final as well. Hence, also ρ' is accepting. □

Lemma 4.32. *Let $\mathcal{A}$ be a weak Büchi game automaton. Let further $F/_{\approx_{\mathcal{A}}}$ be defined as above. Then $\mathcal{A}/_{\approx_{\mathcal{A}}}$ satisfies Definition 3.10.*

Proof. The proof is analogous to the one for Lemma 4.12. This is due to the fact that for deterministic automata the relations $\approx_{di}$ and $\approx_{\mathcal{A}}$ are the same. To adopt the proof to the present case we only need to replace the Büchi game automaton $cl(\mathcal{A})$ from Lemma 4.12 by the weak Büchi game automaton $\mathcal{A}$ and the equivalence relation $\approx_{di}$ by $\approx_{\mathcal{A}}$. □

Since $\mathcal{A}/_{\approx_{\mathcal{A}}}$ satisfies Definition 3.10 the automata $\mathcal{A}$ and $\mathcal{A}/_{\approx_{\mathcal{A}}}$ are equivalent by Remark 3.11. Now we are looking for an acceptance condition for $\mathcal{A}/_{\approx_S}$. It is defined analogously to the case of general Büchi automata (cf. page 56):

$$([s], q) \in F/_{\approx_S} :\Longleftrightarrow \; [(s, q)] \in F/_{\approx_{\mathcal{A}}}$$

We have to show that $\mathcal{A}/_{\approx_S}$ satisfies Definition 3.12. Again, the proof is analogous to the case of Büchi game automata (cf. Lemma 4.13) and we do not give it here. But we conclude that for a weak Büchi game automaton $\mathcal{A}$ with the above definitions of $F/_{\approx_{\mathcal{A}}}$ and $F/_{\approx_S}$ the quotient automaton $\mathcal{A}/_{\approx_S}$ is equivalent to $\mathcal{A}$.

Now we want to deal with the problem of transforming a minimised weak Büchi game automaton into a weak parity game automaton. We show that such a DWA has special properties which make this task very easy. For that we go back to the case of general DWA and not just of game automata. Assume we have a DWA $\mathcal{A}$. To apply the minimisation algorithm from Section 4.2 to it we need a maximal $\mathcal{A}$-colouring d. Our aim is to show that after the minimisation d can be used to define a colouring for an equivalent weak parity automaton obtained from the minimal DWA $\mathcal{A}/_{\approx_{\mathcal{A}}}$.

Lemma 4.33. *Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DWA and let d be a k-maximal $\mathcal{A}$-colouring for some $k \in \mathbb{N}$. Any two states p, q of $\mathcal{A}$ that have different colours are not equivalent:*

$$d(p) \neq d(q) \Longrightarrow p \not\approx_{\mathcal{A}} q$$

Proof. Let $p, q \in Q$ such that $d(p) \neq d(q)$. From Lemma 4.20 it follows that $L_\omega(\mathcal{A}_p) \neq L_\omega(\mathcal{A}_q)$. Then w.l.o.g. there exists $\alpha \in \Sigma^\omega$ such that $\alpha \in L_\omega(\mathcal{A}_p)$ and $\alpha \notin L_\omega(\mathcal{A}_q)$. Since the run ρ of $\mathcal{A}_p$ on α infinitely often visits a final state and the run ρ' of $\mathcal{A}_q$ on α only finitely often visits a final state there exists $i \in \mathbb{N}$ such that $\rho(i) \in F$ and $\rho'(i) \notin F$. Accordingly, $\alpha[0, \ldots, i-1] \in L_*(\mathcal{A}_p)$ and $\alpha[0, \ldots, i-1] \notin L_*(\mathcal{A}_q)$. Hence, $L_*(\mathcal{A}_p) \neq L_*(\mathcal{A}_q)$ which implies $p \not\approx_{\mathcal{A}} q$. □

From this Lemma we can immediately conclude that states of $\mathcal{A}$ which are $\approx_{\mathcal{A}}$-equivalent have the same colour. Thus, the following colouring d^{min} for $\mathcal{A}^{\min}$ is well-defined:

$$\begin{gathered} d^{\min} : Q/_{\approx_{\mathcal{A}}} \to \{0, \ldots, k\} \\ d^{\min}([q]) := d(q) \end{gathered}$$

We utilise this result to explain how a reduced weak Büchi game automaton $\mathcal{B}$ can be transformed into an equivalent weak parity game automaton $\mathcal{C}$.

Lemma 4.34. *Let $\mathcal{A} = ((S \times Q)\dot{\cup}\{q_0, q_{sink}\}, Q, q_0, \delta, F)$ be a weak Büchi game automaton and let $\mathcal{B} := \mathcal{A}/_{\approx_S}$ be defined according to Definition 3.12. Let d be a $(k+2)$-maximal $\mathcal{A}$-colouring for some $k \in \mathbb{N}$. Then $\mathcal{B}$ can be transformed into an equivalent weak parity game automaton $\mathcal{C}$.*

Proof. From Theorem 4.28 we know that $\mathcal{A}/_{\approx_\mathcal{A}}$ is still a DWA. If we have $s_1 \approx_S s_2$ then we get for all $q \in Q$ that $(s_1, q) \approx_\mathcal{A} (s_2, q)$ and $d(s_1, q) = d(s_2, q)$. Final (resp. non-final) SCCs in $\mathcal{A}/_{\approx_\mathcal{A}}$ remain final (resp. non-final) also in $\mathcal{B}$. This means that $\mathcal{B}$ is a DWA as well. The important thing is that the colouring d for $\mathcal{A}$ is translated into a colouring $d/_{\approx_S}$ for $\mathcal{B}$ in a natural way. By this we mean that the step from $d^{\min}$ to $d/_{\approx_S}$ is analogous to the step from d to $d^{\min}$. Since both in $\mathcal{A}^{\min}$ and $\mathcal{B}$ only states with the same colour are merged we have colourings for $\mathcal{A}, \mathcal{A}^{\min}$ and $\mathcal{B}$ which can be directly transformed into each other, i.e. for $(s, q) \in S \times Q$:

$$d(s,q) = d^{\min}([(s,q)]) = d/_{\approx_S}([s], q)$$

The colouring $d/_{\approx_S}$ can be taken as colouring for the weak parity game automaton $\mathcal{C}$. Let $\alpha \in \Sigma^\omega$ and let $\rho_\mathcal{A}$ (resp. $\rho_\mathcal{B}, \rho_\mathcal{C}$) be the run of $\mathcal{A}$ (resp. $\mathcal{B}, \mathcal{C}$) on α. By the above remarks, note that $d(\rho_\mathcal{A}) = d/_{\approx_S}(\rho_\mathcal{B}) = d/_{\approx_S}(\rho_\mathcal{C})$. This yields:

$$\begin{aligned} \rho_\mathcal{A} \text{ is accepting} &\overset{\text{Rem. 4.16}}{\Longleftrightarrow} \max(Occ(d(\rho_\mathcal{A}))) \text{ is even} \\ &\Longleftrightarrow \max(Occ(d/_{\approx_S}(\rho_\mathcal{C}))) \text{ is even} \\ &\overset{\text{Def.}}{\Longleftrightarrow} \rho_\mathcal{C} \text{ is accepting} \end{aligned}$$

Hence, $\mathcal{A}$ and $\mathcal{C}$ are equivalent. Note that $\mathcal{A}$ and $\mathcal{B}$ are equivalent by Lemma 3.13. This implies that $\mathcal{B}$ and $\mathcal{C}$ are equivalent as well. ☐

With Lemmas 4.30 and 4.34 we have established a transformation from weak parity game automata to weak Büchi game automata, and vice versa.

Chapter 5

Memory Reduction

In this chapter we summarise our results from the first four chapters and describe algorithms to reduce the memory necessary for implementing winning strategies in Request-Response and Staiger-Wagner games. For this we give detailed descriptions of the steps to be performed in both cases.

5.1 Request-Response Games

We are given a Request-Response game $\Gamma = (G, \varphi)$ with game graph $G = (Q, E)$. Furthermore, we are given the set $\mathcal{F} = \{(P_1, R_1), \ldots, (P_k, R_k)\}$ of pairs of sets $P_i, R_i \subseteq Q$. We use the following winning condition φ:

$$\text{Player 0 wins } \rho :\Longleftrightarrow \forall r(\rho(r) \in P_i \Longrightarrow \exists s \geq r : \rho(s) \in R_i)$$

As we have already seen in Section 2.2.4 Request-Response games are reducible to Büchi games. Consequently, since Büchi games are determined Request-Response games are as well. From this we conclude that $Q = W_0 \dot\cup W_1$. Our aim is to find for a given vertex $q \in W_0$ a strategy automaton $\mathcal{A}_f$ with as few states as possible which implements a winning strategy f for Player 0 from q. Before we describe an algorithm for finding such a strategy automaton we give an example which has excellent properties to illustrate our algorithm.

Example 5.1. Consider the Request-Response game from Figure 5.1 below where we have the set $\mathcal{F} = \{(\{0\}, \{1\}), (\{0\}, \{2\}), (\{0\}, \{3\}), (\{0\}, \{4\}), (\{0\}, \{5\})\}$.

The winning region of Player 0 in this example is the complete set of vertices, i.e. $W_0 = Q = \{0, 1, 2, 3, 4, 5\}$. Accordingly, we have $W_1 = \emptyset$. Player 0 has five positional strategies $f_1, \ldots, f_5$:

$$f_i(w0) = i \text{ for } i = 1, \ldots, 5$$

Obviously, none of them is winning. If Player 0 plays according to f_i then only P_i is responded to. Any strategy automaton implementing a winning strategy for

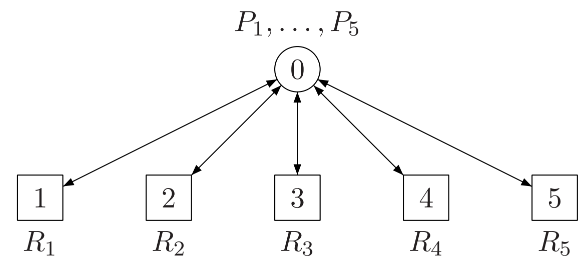


Figure 5.1: *Request-Response game graph G*

Player 0 needs at least five states. Indeed, we can give an automaton $\mathcal{A}_f$ of that size which implements a uniform winning strategy f from W_0 for him. The idea is to move cyclically from vertex 0 to vertices $1, \dots, 5$:

$$\begin{aligned} S &= \{pair_i \mid i = 1, \dots, 5\} \\ Q &= \{0, 1, 2, 3, 4, 5\} \\ s_0 &= pair_1 \end{aligned}$$

The initial memory content is set to $pair_1$. It signalises that the pair $(P_1, R_1) = (\{0\}, \{1\})$ should be responded to first. Note that f would also be a uniform winning strategy for Player 0 from W_0 if we chose $s_0 = pair_i$ for any $i \in \{2, \dots, 5\}$. The memory update function $\sigma : S \times Q \to S$ is defined as follows:

$$\begin{aligned} \sigma(pair_i, 0) &= pair_i && \text{for } i \in \{1, \dots, 5\} \\ \sigma(pair_i, j) &= pair_{(i \bmod 5)+1} && \text{for } i, j \in \{1, \dots, 5\} \end{aligned}$$

The output function $\tau : S \times Q_0 \to Q$ specifies the winning strategy f for Player 0 from W_0. It is defined as follows:

$$\tau(pair_i, 0) = i \text{ for } i \in \{1, \dots, 5\}$$

If we apply the reduction algorithm from Section 2.2.4 to compute a strategy automaton which implements a uniform winning strategy from W_0 for Player 0 we get a game graph with 1920 states:

$$|Q'| = 2^{|\mathcal{F}|} \cdot |\mathcal{F}| \cdot 2 \cdot |Q| = 2^5 \cdot 5 \cdot 2 \cdot 6 = 32 \cdot 5 \cdot 2 \cdot 6 = 1920$$

The memory of this construction has a size of $2^5 \cdot 5 \cdot 2 = 320$. But only very few of them can be reached in G'. If a play starts in vertex 0 then after the first move (to one of the vertices $1, \dots, 5$) all request-response pairs $P_1, \dots, P_5$ are activated. So, we visit the two memory contents $(\emptyset, 1, 0)$ and $(\{1, \dots, 5\}, 1, 0)$. If the play starts in any other vertex then it takes two moves until all pairs are activated and only three memory contents are visited, namely $(\emptyset, 1, 0)$, $(\emptyset, 2, 1)$ and $(\{1, \dots, 5\}, 2, 0)$. This yields a total of four reachable memory contents after the first activation of all five pairs from $\mathcal{F}$. From then on there is exactly one set P_i responded to at any visit to a vertex from $\{1, \ldots, 5\}$. One move later we reach vertex 0 and all five pairs are again activated. Hence, the first memory component, i.e. the set of active pairs, always has at least four elements. This means that any memory content with less than four elements in the first component is non-reachable (except $(\emptyset, 1, 0)$ and $(\emptyset, 2, 1)$). A rough estimation yields a total of at most $6 \cdot 5 \cdot 2 + 2 = 62$ reachable memory contents. In fact only 32 memory contents are reachable. Our aim is to further reduce this set by our algorithm.

We now give a step-by-step procedure for finding the strategy automaton $\mathcal{A}_f$ we are looking for:

Algorithm 5.2. (Memory Reduction for Request-Response Games)
Input: Request-Response game $\Gamma = (G, \varphi)$
Output: Strategy automaton $\mathcal{A}_f$ for Player 0 from W_0

1. From the given Request-Response game $\Gamma = (G, \varphi)$ we establish a game reduction to the Büchi game $\Gamma' = (G', \varphi')$. For that we use the reduction algorithm as described in the proof of Theorem 2.14. By Definition 2.11 we know that Player 0 wins a play ρ in Γ if and only if Player 0 wins the corresponding play ρ' in Γ'.

2. From Γ' we construct the Büchi game automaton $\mathcal{A}_{\Gamma'}$ according to Definition 3.5. By Remark 3.7 we know that $\mathcal{A}_{\Gamma'}$ accepts exactly those sequences of states of G that are winning plays for Player 0 in Γ.

3. From the automaton $\mathcal{A}_{\Gamma'}$ we compute the automaton $cl(\mathcal{A}_{\Gamma'})$ applying the closure operator to the set F of final states as described on page 52. From Lemma 4.8 we know that $\mathcal{A}_{\Gamma'}$ and $cl(\mathcal{A}_{\Gamma'})$ are equivalent. For this reason item 3. of Definition 2.11 still holds. This means that Γ is reducible to the automaton game of $cl(\mathcal{A}_{\Gamma'})$ (cf. Definition 3.8).

4. On $cl(\mathcal{A}_{\Gamma'})$ we compute the direct bisimulation $\approx_{di}$. As we know from Remark 4.10 direct bisimulation on $cl(\mathcal{A}_{\Gamma'})$ preserves the recognised language. To assure that the reduced automaton obtained from $cl(\mathcal{A}_{\Gamma'})$ has as state set $(S' \times Q)\dot{\cup}\{q_0, q_{sink}\}$ for a finite memory S' we do not merge all states of $cl(\mathcal{A}_{\Gamma'})$ that could be merged according to the direct bisimulation relation. Instead we choose the pairs of states $(s_1, s_2) \in S \times S$ to be merged according to Definition 3.12 where we replace $\approx$ by $\approx_{di}$ for the present case. To apply Lemma 3.13 we have to show compatibility for $\approx_{di}$ with $cl(\mathcal{A}_{\Gamma'})$ which has been done in Lemma 4.11. With Lemma 4.8 and this result we can conclude that the automaton $\mathcal{B} := cl(\mathcal{A}_{\Gamma'})/_{\approx_S}$ is equivalent to $\mathcal{A}_{\Gamma'}$.

5. From $\mathcal{B}$ we obtain the corresponding automaton game $\Gamma''_{\mathcal{B}}$ according to Definition 3.8. $\Gamma''_{\mathcal{B}}$ is a Büchi game and can be solved like was described in Section 2.2.1. Moreover, from Theorem 3.14 we know that Γ is reducible to $\Gamma''_{\mathcal{B}}$. Thus, we can compute a strategy automaton $\mathcal{A}_f$ for Player 0 from $\Gamma''_{\mathcal{B}}$ in the same way as we could do from Γ'. It can be constructed as described in the proof of Theorem 2.12. The automaton $\mathcal{A}_f$ is the one we were looking for.

The algorithm above takes exponential time. This is due to step one. There we have an exponential blow-up of the game graph because of the memorisation of the requested pairs (P_i, R_i). Steps two and three take linear time in the size of G', step four takes time $O(m \cdot \log(n))$. Here, we assume that m is the number of edges and n is the number of vertices of G'. Step five takes polynomial time due to the computation of $Attr_0(Recur_0(F'))$ when solving $\Gamma''_{\mathcal{B}}$.

If we apply the above algorithm to the game from example 5.1 then we only consider the reachable memory contents in step one. We get a Büchi game graph with 57 reachable vertices and 165 transitions between them. Six vertices are final. To be able to compute the memory equivalence relation $\approx_S$ we need to add all vertices $(s, q) \in S \times Q$ where there exists a vertex $q' \in Q$ such that (s, q') is reachable from an initial game position in G'. Additionally we have to add all vertices that are reachable from vertices (s, q) with this property. The reason for this is explained in Section 6.1. Hence, we get 186 additional vertices (among them 51 final vertices) and 410 new transitions. We get a game graph G' with 243 states and 575 transitions which is then viewed as the corresponding game automaton $\mathcal{A}_{\Gamma'}$ (cf. step two). On $\mathcal{A}_{\Gamma'}$ we compute the closure of final states. We get 30 additional final states and, thus, obtain a Büchi automaton with 87 final states. In step four we compute the direct bisimulation relation $\approx_{di}$ on this automaton and obtain 25 equivalence[1] classes. According to Definition 3.12 we obtain an equivalence relation $\approx_S$ on the set S with the following[2] twelve equivalence classes:

Class 1: $\{(11111, 1, 0), (10111, 1, 0), (11011, 1, 0), (11101, 1, 0), (11110, 1, 0)\}$
Class 2: $\{(11111, 2, 0), (01111, 2, 0), (11011, 2, 0), (11101, 2, 0), (11110, 2, 0)\}$
Class 3: $\{(11111, 3, 0), (01111, 3, 0), (10111, 3, 0), (11101, 3, 0), (11110, 3, 0)\}$
Class 4: $\{(11111, 4, 0), (01111, 4, 0), (10111, 4, 0), (11011, 4, 0), (11110, 4, 0)\}$
Class 5: $\{(11111, 5, 0), (01111, 5, 0), (10111, 5, 0), (11011, 5, 0), (11101, 5, 0)\}$
Class 6: $\{(00000, 1, 0)\}$
Class 7: $\{(11110, 1, 1)\}$
Class 8: $\{(00000, 2, 1)\}$
Class 9: $\{(01111, 2, 1)\}$
Class 10: $\{(10111, 3, 1)\}$
Class 11: $\{(11011, 4, 1)\}$
Class 12: $\{(11101, 5, 1)\}$

We obtain the quotient game automaton $\mathcal{B} := cl(\mathcal{A}_{\Gamma'})/_{\approx_S}$ with 42 reachable states (16 final states) and 90 transitions. Since $\approx_{di}$ is compatible with $cl(\mathcal{A}_{\Gamma'})$ (cf. Lemma 4.11) we conclude that $\mathcal{B}$ is equivalent to $cl(\mathcal{A}_{\Gamma'})$ and to $\mathcal{A}_{\Gamma'}$. In the last step we transform $\mathcal{B}$ into the corresponding Büchi automaton game $\Gamma''_{\mathcal{B}}$. From Theorem 3.14 it follows that Γ is reducible to $\Gamma''_{\mathcal{B}}$. If we solve $\Gamma''_{\mathcal{B}}$ we obtain the winning region $W''_0 \supseteq \{(00000, 1, 0)\} \times Q$ for Player 0. From that we derive the winning region $W_0 = Q$ for Player 0 in Γ. From the positional winning strategy for Player 0 in $\Gamma''_{\mathcal{B}}$

[1]We do not need to consider equivalent pairs of states with distinct Q-components.

[2]The sets of active requests are coded by bit vectors $(b_1, \ldots, b_5)$ where $b_i = 1$ means that request i is active.

we construct the strategy automaton $\mathcal{A}_f$ implementing the winning strategy f for Player 0 in Γ. $\mathcal{A}_f$ has as states the set of 12 memory equivalence classes from above.

We have already seen that a memory of size five suffices to implement a winning strategy for the Request-Response game from Example 5.1. In Chapter 7 we will examine in more detail for which kind of examples our algorithm yields good results and for which kind the minimisation algorithm for strategy automata should be preferred.

5.2 Staiger-Wagner Games

Before we describe the algorithm for memory reduction for strategies in Staiger-Wagner games we want to give a motivation for it. We believe that in a substantial number of games it is possible to use far less memory than proposed by the reduction algorithm from Section 2.2.5. This gap may even have an exponential dimension.

Example 5.3. Consider the following game $\Gamma = (G, \varphi)$ from Figure 5.2 where we have the usual Staiger-Wagner winning condition φ used with the set family $\mathcal{F} = \{\{1,2,3\}, \{1,2,3,4,5,6,10\}, \{5,6,9,10\}\}$.

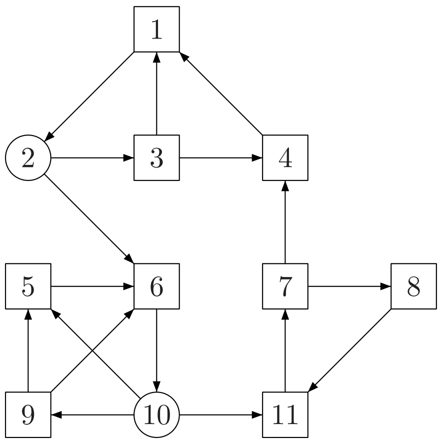


Figure 5.2: *Staiger-Wagner game graph G*

The winning region W_0 of Player 0 is $W_0 = \{1,2,3,4,5,6,9,10\}$ and Player 1 wins from $W_1 = \{7,8,11\}$. Player 0 has positional winning strategies only from vertices 5 and 9 whereas from all other vertices in W_0 he needs to memorise a certain history of the play to win. To achieve that from vertices 6 or 10 he needs a memory of size two. A memory of this size is also necessary if the play starts in vertices $1, 2$ or 3. However, from vertex 4 Player 0 needs three different memory contents to be able to force a win. Hence, any strategy automaton implementing a winning strategy for Player 0 from W_0 needs at least three states. The following strategy automaton $\mathcal{A}_f = (S, Q, s_0, \sigma, \tau)$ implements a winning strategy f for Player 0:

$$\begin{aligned}
S &= \{local, switch, global\}\\
Q &= \{1,2,3,4,5,6,7,8,9,10,11\}\\
s_0 &= local
\end{aligned}$$

The initial memory content is called *local*. It signalises that Player 0 either has to stay in cycle $(1,2,3)$ or that he has to move to vertex 9 next time the play reaches vertex 10. In the latter situation - after the visit to vertex 9 - memory state *global* is assumed to notify that next time vertex 10 is reached Player 0 should move to vertex 5. If the play starts in the upper part of the game graph, i.e. in one of the vertices $1,2,3$ or 4, then the situation is more subtle. There exists a winning strategy $f_{1,2,3}$ from vertices $1,2$ and 3 and a winning strategy f_4 from vertex 4 that give different semantics to the memory state *switch*. If we combine both to get a winning strategy from $\{1,2,3,4\}$ we have to take the semantics of f_4. The memory *switch* is assumed if vertex 4 is visited and at the next visit to vertex 2 we move to vertex 3. This visit to vertex 3 lets $\mathcal{A}_f$ assume state *global*. This strategy assures that all vertices in $\{1,2,3,4\}$ are visited before the play proceeds to vertex 6. The reader may verify that it is not possible to force a visit to vertices $1,2,3$ and 4 from $\{1,2,3,4\}$ with a winning strategy of size two. We now give the definition of the memory update function $\sigma : S \times Q \to S$ and the output function $\tau : S \times Q_0 \to Q$ of $\mathcal{A}_f$. There exist transitions which are not relevant. For example, $\mathcal{A}_f$ can never read 4 when in state *switch*:

$$\begin{aligned}
\sigma(local, i) &= local \text{ for } i \in \{1,2,3,5,6,7,8,10,11\}\\
\sigma(local, 4) &= switch\\
\sigma(local, 9) &= global\\[1ex]
\sigma(switch, i) &= switch \text{ for } i \in \{1,2,4,\ldots,11\}\\
\sigma(switch, 3) &= global\\[1ex]
\sigma(global, i) &= global \text{ for } i \in \{1,\ldots,11\}
\end{aligned}$$

The output function $\tau : S \times Q_0 \to Q$ specifies the winning strategy f for Player 0 from W_0. It is defined as follows:

$$\begin{aligned}
\tau(local, 2) &= 3\\
\tau(local, 10) &= 9\\
\tau(switch, 2) &= 3\\
\tau(switch, 10) &= 9\\
\tau(global, 2) &= 6\\
\tau(global, 10) &= 5
\end{aligned}$$

If we apply the reduction algorithm from Section 2.2.5 then we get more than twenty thousand states for G':

$$|Q'| = 2^{|Q|} \cdot |Q| = 2^{11} \cdot 11 = 2048 \cdot 11 = 22528$$

But if we have a closer look at G then we see that only some of them are relevant. Firstly, not all 22528 states are reachable in G'. For example, any state (s,q) with the memory component $s = \{2, 4\}$ is not reachable because the memory component s stores which vertices have been visited in a play up to a certain time point. But any play that has visited vertices 2 and 4 must also have visited vertex $1, 3$ or 6. Furthermore, any vertex of G' where the memory component is a superset of $\{2, 4\}$ not containing vertices $1, 3$ or 6 is not reachable in G'. This yields a total of $2^6 \cdot 11 = 704$ non-reachable states. Similar considerations can be done with sets like e.g. $\{1, 9\}$, $\{5, 11\}$ or $\{6, 7\}$. To enumerate all sets with this property is left to the reader. All non-reachable states can be eliminated in G' without changing the winning regions or winning strategies of Γ.

A second observation can be made which is related to the winning condition. Suppose a play has started in vertex 10 and has finally left the set $\{5, 6, 9, 10\}$ to proceed to vertex 11. Let us abstain from the fact that Player 0 can win from vertex 10 and would never move to vertex 11 but remain in the set $\{5, 6, 9, 10\}$ until infinity. In the situation mentioned the memory component s becomes equal to $\{5, 6, 9, 10, 11\}$ which means that Player 0 has no chance to win because there is no set $F_i \in \mathcal{F}$ such that $\{5, 6, 9, 10, 11\} \subseteq F_i$. In Γ' this means that no longer an SCC with even colour is reachable. Now, assume the situation where the play starts in vertex 7. No play starting from there can be won by Player 0 for the same reason as for $\{5, 6, 9, 10, 11\}$. Hence, a strategy automaton would not have to distinguish whether vertices $5, 6, 9, 10$ and 11 or whether vertices, say $7, 4$ and 1, have been visited. The memory contents $\{5, 6, 9, 10, 11\}$ and $\{1, 4, 7\}$ could be merged. Note that the same argument can be brought forward if the play started at any vertex which is not contained in any of the F_i, e.g. vertex 8.

If we summarise our observations from above we come to the following conclusions:

1. For a Staiger-Wagner game numerous subsets of Q need not be considered as memory contents. This is due to the fact that there exist state sequences $\rho \in Q^\omega$ which are not possible plays on G. Which sequences are possible and which are not depends on the transition structure of G.

2. The memory component s of any vertex (s, q) such that there exists no $F_i \in \mathcal{F}$ with $s \subseteq F_i$ can be set equivalent to a dummy memory content.

The above example justifies that in the reduction algorithm from Section 2.2.5 there is tremendous potentialities for a reduction of the memory component $S = 2^Q$. We now describe a memory reduction algorithm for Staiger-Wagner games. This algorithm involves more technical details than the one for Request-Response games. If we apply the game reduction algorithm for Staiger-Wagner games then we obtain a weak parity game. Moreover, we have introduced deterministic weak Büchi automata (DWA). In Lemmas 4.30 and 4.34 we have seen that both types of automata are equivalent. Any weak parity game automaton which is obtained from a Staiger-Wagner can be transformed into an equivalent DWA. We do so by computing a maximal colouring (cf. Definition 4.15) and set the final states to be the set of states with even colour. In a similar way any DWA can be considered as a weak parity automaton. A major advantage of DWA over general (deterministic) Büchi automata is that there exists an efficient minimisation algorithm for DWA (cf. Theorem 4.28). We have to do some intermediate steps to transform a given weak parity automaton into a DWA. Again, we are looking for a strategy automaton $\mathcal{A}_f$ implementing a winning strategy for Player 0 with as few states as possible. The algorithm for memory reduction for strategies in Staiger-Wagner games works as follows:

Algorithm 5.4. (Memory Reduction for Staiger-Wagner Games)
Input: Staiger-Wagner game $\Gamma = (G, \varphi)$
Output: Strategy automaton $\mathcal{A}_f$ for Player 0 from W_0

1. From the given Staiger-Wagner game $\Gamma = (G, \varphi)$ we establish a game reduction to the weak parity game $\Gamma' = (G', \varphi')$. For that we use the reduction algorithm as described in Section 2.2.5 (cf. page 27). By Definition 2.11 we know that Player 0 wins a play ρ in Γ if and only if Player 0 wins the corresponding play ρ' in Γ'.

2. From Γ' we construct the weak parity game automaton $\mathcal{A}_{\Gamma'}$ according to Definition 3.5. To have a well-defined colouring for $\mathcal{A}_{\Gamma'}$ we additionally set the colour of q_0 to zero and the colour of q_{sink} to $2 \cdot |Q| + 1$. By Remark 3.7 we know that $\mathcal{A}_{\Gamma'}$ accepts exactly those sequences of states of G that are winning plays for Player 0 in Γ.

3. We transform $\mathcal{A}_{\Gamma'}$ into an equivalent DWA $\mathcal{A}$ (cf. Lemma 4.30). From Theorem 4.28 we know that any DWA in normal form can be minimised by a DFA minimisation algorithm. To transform the DWA $\mathcal{A}$ into normal form we need a maximal $\mathcal{A}$-colouring according to Definition 4.15. We cannot be sure that the colouring[3] given by the game reduction is a maximal $\mathcal{A}$-colouring. So, we compute a new $(k+2)$-maximal[4] colouring c for some even $k \in \mathbb{N}$. All vertices within the same SCC get the same colour and so we need at most as many colours as G' can have SCCs. Since an SCC can be a singleton we set k to be the smallest even number greater than the number of vertices of $S \times Q$, i.e. $k := \min\{n \in \mathbb{N} \mid n > 2^{|Q|} \cdot |Q|,\ n \text{ even}\}$. With this value for k and Algorithm 4.27 we obtain a unique $(k+2)$-maximal $\mathcal{A}$-colouring c[5]. To obtain a DWA in normal form we define the set of final states as the set of all states with even colour:

$$F := F_c = \{q' \in Q' \mid c(q') \text{ is even}\}$$

So, we obtain the DWA $\mathcal{A} = ((S \times Q)\dot\cup\{q_0, q_{sink}\}, Q, q_0, \delta, F)$ in normal form where all except F is given by $\mathcal{A}_{\Gamma'}$. Due to Lemmas 4.30 and 4.26 $\mathcal{A}_{\Gamma'}$ and $\mathcal{A}$ are equivalent.

[3]Note that this colouring was defined for a weak parity automaton and has nothing to do with Definition 4.15.

[4]We choose $k+2$ because we have to set $c(q_{sink}) := k+1$.

[5]We actually obtain a colouring d for the SCC-graph of $\mathcal{A}$ and can derive c from it.

4. If we apply a DFA minimisation algorithm to the DWA $\mathcal{A}$ from step three then by Theorem 4.28 this yielded an equivalent minimal DWA $\mathcal{B}$. To assure that $\mathcal{B}$ has as state set $(S' \times Q)\dot\cup\{q_0, q_{sink}\}$ for a finite memory S' we do not merge all states of $\mathcal{A}$ that could be merged. Instead we choose the pairs of states $(s_1, s_2) \in S \times S$ to be merged according to Definition 3.12. As a drawback we get that $\mathcal{B}$ is not minimal. For the present case $\approx$ is replaced by $\approx_\mathcal{A}$ where

$$\begin{gathered}(s,q) \approx_\mathcal{A} (s',q') :\Longleftrightarrow \\ \forall w \in \Sigma^* : \delta((s,q),w) \in F \Longleftrightarrow \delta((s',q'),w) \in F\end{gathered}$$

is the usual definition of state equivalence known from DFA automata theory. If we want to apply Lemma 3.13 we have to prove compatibility for $\approx_\mathcal{A}$ with $\mathcal{A}$ which has been done in Lemma 4.31. With this result we can conclude that the automaton $\mathcal{B} := \mathcal{A}/_{\approx_S}$ is equivalent to $\mathcal{A}$. Note that $\mathcal{B}$ is still a DWA (cf. Lemma 4.34).

5. From Lemma 4.33 we know that in $\mathcal{A}$ only states with the same colour can be $\approx_\mathcal{A}$-equivalent and from Lemma 4.34 we know that $\mathcal{B}$ together with the colouring $c/_{\approx_S}$ (cf. page 69) yields a weak parity game automaton $\mathcal{C}$ which is equivalent to $\mathcal{B}$. $c/_{\approx_S}$ is defined as follows:

$$c/_{\approx_S} : (S' \times Q)\dot\cup\{q_0, q_{sink}\} \to \{0, \dots, k+2\}$$

$$c/_{\approx_S}(q') := \begin{cases} c(s,q) & \text{, if } q' = ([s],q) \\ c(q') & \text{, if } q' = q_0 \text{ or } q' = q_{sink} \end{cases}$$

6. From $\mathcal{C}$ we obtain the corresponding automaton game $\Gamma''_\mathcal{B}$ according to Definition 3.8. $\Gamma''_\mathcal{B}$ is a weak parity game and can be solved like was described in Section 2.2.2. Moreover, from Theorem 3.14 we know that Γ is reducible to $\Gamma''_\mathcal{B}$. Thus, we can compute a strategy automaton $\mathcal{A}_f$ for Player 0 from $\Gamma''_\mathcal{B}$ in the same way as we could do from Γ'. It can be constructed as described in the proof of Theorem 2.12. The automaton $\mathcal{A}_f$ is the one we were looking for.

If we apply this algorithm to the game from Example 5.3 we get a weak parity game graph G' with 592 reachable states and 889 transitions. If we add all states (s,q) with reachable s and all states reachable from those states we obtain an extended game graph with 4793 states and 7226 transitions. In step two we transform this game graph into the corresponding weak parity game automaton $\mathcal{A}_{\Gamma'}$. We compute a maximal colouring and define all states with even colour as final. To the resulting DWA $\mathcal{A}$ we apply the minimisation algorithm for DFA and obtain the compatible equivalence relation $\approx_\mathcal{A}$ with 39 equivalence classes. From that we compute the equivalence relation $\approx_S$ on S which has 19 classes. Lemma 3.13 yields that $\mathcal{A}$ and $\mathcal{B} := \mathcal{A}/_{\approx_S}$ are equivalent. $\mathcal{B}$ has 97 reachable states and 154 transitions and we transform it into the corresponding automaton game $\Gamma''_\mathcal{B}$. From Theorem 3.14 it follows that Γ is reducible to $\Gamma''_\mathcal{B}$. If we solve the latter game by the iterative computation of attractor sets (cf. Section 2.2.2) then we obtain the winning region $W''_0 \supseteq \{\emptyset\} \times \{1, 2, 3, 4, 5, 6, 9, 10\}$ for Player 0. From W''_0 we obtain the winning region $W_0 = \{1, 2, 3, 4, 5, 6, 9, 10\}$ for Player 0 in Γ. With the positional winning strategy for Player 0 in $\Gamma''_{\mathcal{B}}$ we can construct the strategy automaton $\mathcal{A}_f$ implementing a winning strategy for Player 0 from W_0. $\mathcal{A}_f$ has 19 states, namely the equivalence classes of $\approx_S$.

Example 5.3 has a special property. We have 195 reachable memory contents but 150 of them are "losing" for Player 0. By this we mean that they are not a subset of any of the three sets from $\mathcal{F}$. From all states in G' that have a losing memory component only states with odd colour are reachable. Since all these states retain odd colours in the maximal colouring the transformation to DWA (cf. Lemma 4.30) declares them as non-final. Consequently, from each of this vertices the empty language is accepted and the DFA minimisation algorithm detects them to be all $\approx_{\mathcal{A}}$-equivalent. The corresponding equivalence class has 4437 elements where all other classes have between 3 and 32 equivalent states. Similarly, 18 classes of $\approx_S$ contain 1 to 5 equivalent memory contents but one class contains all 150 memory contents losing for Player 0.

Chapter 6

The Implementation

In this chapter we describe how the memory reduction algorithm (Algorithm 3.4) has been implemented. Minimisation (even reduction) of ω-automata is a difficult problem for most types of acceptance conditions. Our algorithm can only be applied to those types of games for which we have a reduction algorithm in our hands. In particular, if we establish a game reduction from a given game Γ to a new game Γ' then the type of winning condition of Γ' determines whether our algorithm can be applied or not (cf. step three of Algorithm 3.4).

As we have seen in Chapter 4 there exists an efficient minimisation algorithm for DWA. We apply it to the equivalent class of weak parity game automata (Algorithm 5.4). We do not know about an efficient minimisation algorithm for the more general class of deterministic Büchi automata (DBA). Hence, we use a heuristic for a class which is again more general[1] than DBA, namely nondeterministic Büchi automata (NBA). This is the notion of delayed simulation introduced in [EWS05] and we use it to reduce the state space of a Büchi game automaton (Algorithm 5.2).

Before we present the tool *GASt* that is used to realise the implementation of Algorithms 5.2 and 5.4 we address a problem that emerges with the computation of $\approx_S$, the equivalence relation on the set of memory contents. It has a major impact on the running time of our algorithm.

6.1 Equivalence of Memory Contents

Let us consider the game graph from Figure 3.5. If we apply the game reduction algorithm from page 27 to this game then we obatin a weak parity game graph with $|S| \cdot |Q| = 16 \cdot 4 = 64$ vertices. One part of the resulting game graph, i.e. the copies for the memory contents $\emptyset$, $\{0\}$ and $\{0, 2\}$, can be seen in Figure 6.1. We have left out all other copies and most of the transitions as well.

In this game graph only 20 vertices and 29 transitions are reachable from the set of initial game positions, i.e. from $\{\emptyset\} \times Q$. For example, in the copy of memory content $\{0\}$ only the vertices 1 and 2 are reachbale. The question is how to treat

[1]DBA are strictly weaker than NBA (cf. [TL03]).

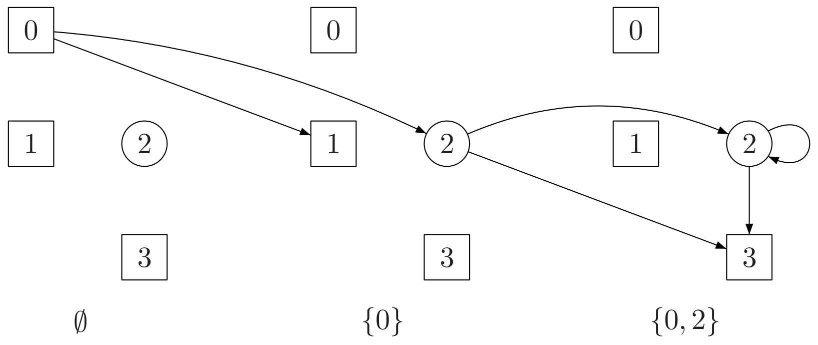


Figure 6.1: *Game reduction applied to the game graph from Figure 3.5*

non-reachable vertices when computing the equivalence relation $\approx_S$. According to Definition 3.12 we have to consider all vertices of $S \times Q$. Obviously, if for memory content $s \in S$ there exists no $q \in Q$ such that (s, q) is reachable then the copy of s does not need to be considered. But if there exists $s \in S$ and $q_1, q_2 \in Q$ such that (s, q_1) is reachable but (s, q_2) is not then it is not immediately clear whether (s, q_2) has to be considered or not. This problem does not arise in the minimisation of normal automata. There, we delete non-reachable states before we start the actual minimisation. In the present case we want to leave out as many states as possible to improve the running time. We do present two approaches here how to deal with this problem. In the first approach we do not need to consider the non-reachable states. This version was implemented first. Later we found out that we get better results with another approach but, of course, only with the drawback of higher computation costs.

In the first approach to solve the above problem we restrict the definition of $\approx_S$ further. We additionally claim that for two memory contents s_1, s_2 to be $\approx_S$-equivalent exactly for the same vertices $q \in Q$ in the original game graph the vertices $(s_1, q), (s_2, q)$ in the new game graph are reachable. We call this the *normal* version of our algorithm. This means to compute the equivalence relation $S/_{\approx_S}$ in step three of Algorithm 3.4 we use the following definition:

$$\begin{aligned} s_1 \approx_S s_2 :\Longleftrightarrow \forall q \in Q : (s_1, q) \text{ is reachable} &\Longleftrightarrow (s_2, q) \text{ is reachable and} \\ &\text{if } (s_1, q) \text{ is reachable then } (s_1, q) \approx (s_2, q) \end{aligned}$$

If we compute $\approx_S$ with this definition of equivalence then we get a relation which is neither a subset nor a superset of the relation that we get from Definition 3.12. Nevertheless, we get a correct result, i.e. the quotient automaton $\mathcal{A}_{\Gamma'}/_{\approx_S}$ is equivalent to $\mathcal{A}_{\Gamma'}$. This is due to the fact that two non-equivalent states can only be merged if neither of them is reachable in $\mathcal{A}_{\Gamma'}$. Furthermore, if (s, q) is non-reachable in $\mathcal{A}_{\Gamma'}$ then $([s], q)$ is non-reachable in $\mathcal{A}_{\Gamma'}/_{\approx_S}$. In the quotient automaton only such states are merged which are either all reachable or all non-reachable:

$$(s, q) \text{ is reachable in } \mathcal{A}_{\Gamma'} \Longleftrightarrow ([s], q) \text{ is reachable in } \mathcal{A}_{\Gamma'}/_{\approx_S}$$

With this definition we get for the Staiger-Wagner game from Example 5.3 a memory of size 37. Of course, this definition of $\approx_S$ is too restricting. Normally, for two different memory contents s_1 and s_2 not exactly the states with the same Q-component are reachable. We wish for a definition that makes it possible to compare memory contents independently of this property. The idea is to lessen the above restriction in such a way that only one of the states (s_1, q) or (s_2, q) has to be reachable. That means we do not consider pairs where both (s_1, q) and (s_2, q) are non-reachable.

$$s_1 \approx_S s_2 :\Longleftrightarrow \forall q \in Q : \text{if } (s_1, q) \text{ is reachable or } (s_2, q) \text{ is reachable then } (s_1, q) \approx (s_2, q)$$

To see that this definition does not yield an equivalence relation consider the situation in the following Figure 6.2.

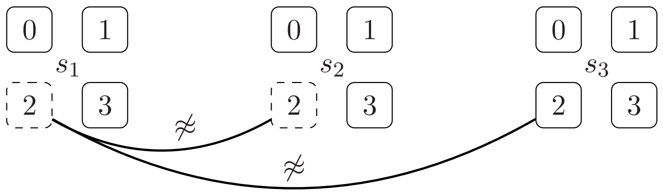


Figure 6.2: *Non-transitive relation*

We assume that all states with the same Q-component and no edge between them are $\approx$-equivalent. That means $((s_1, 2), (s_2, 2))$ and $((s_1, 2), (s_3, 2))$ are the only non-equivalent pairs of states that are relevant. Note that $\approx$ is an equivalence relation. Additionally, we suppose that $(s_1, 2)$ and $(s_2, 2)$ are non-reachable, indicated by the dashed borders of these states. We get $s_1 \approx_S s_2$ because only the pair $((s_1, 2), (s_2, 2))$ is non-equivalent but neither of its states is reachable. We also get $s_2 \approx_S s_3$ because all pairs of states with the same Q-component are $\approx$-equivalent. But we get $s_1 \not\approx_S s_3$ because $(s_1, 2) \not\approx (s_3, 2)$ and $(s_3, 2)$ is reachable. Altogether:

$$s_1 \approx_S s_2, s_2 \approx_S s_3 \not\Longrightarrow s_1 \not\approx_S s_3$$

Hence, the relation $\approx_S$ is non-transitive. That means it does not suffice to consider only those pairs of states where at least one of the associated states is reachable. We always need to consider all pairs. More precisely, if there exists a reachable memory content, say s, and within its copy a reachable state, say (s, q), then for each reachable memory content s' we need to consider the state (s', q). Since in the copy of the initial memory content s_0 all states (s_0, q) are reachable this implies that all reachbale copies have to be fully considered. This is the same as Definition 3.12 restricted to reachable memory contents. If we have e.g. $|Q| = 10$ and two reachable memory contents s_1, s_2 where for $q_1 \in Q$ only the state (s_1, q_1) and for $q_2 \in Q$ only the state (s_2, q_2) is reachable then nonetheless we need to take into consideration all ten pairs of states to determine whether s_1 and s_2 are equivalent. This makes the computation of $\approx_S$ very time-consuming.

The above considerations imply that for each pair $((s_1, q), (s_2, q))$ of states with reachable s_1 and reachable s_2 we need to know whether the pair is $\approx$-equivalent or not. In Algorithms 5.2 and 5.4 the actual computation of $\approx$ is done with the minimisation algorithm for DFA. There, to determine whether (s_1, q) is $\approx$-equivalent to (s_2, q) we need all states of $\mathcal{A}_{\Gamma'}$ reachable from (s_1, q) or (s_2, q). In our implementation we take the following three steps to get all of $\mathcal{A}_{\Gamma'}$ necessary for computing $\approx$:

1. In the game reduction algorithm we use a depth-first search to detect the reachable fragment of $\mathcal{A}_{\Gamma'}$.
2. From step one we get all reachable memory contents. For each reachable memory content s we add to the result of the first step all non-reachable states (s, q).
3. To be able to compute $\approx$ we need to add all states reachable from a state added in step two.

The following figure clarifies which states need to be considered when using this approach. $\mathit{Reach}(\{s_0\} \times Q)$ denotes the set of all states reachable from the set of initial game positions. $NR(s)$ denotes the set of all states with memory component s that are non-reachable from the set of initial game positions. In the figure all states with memory content s_1 are reachable. In the copy of s_2 only few states are reachable whereas in the copy of s_4 only few states are non-reachable. All states which have as memory content s_1, s_2 or s_4 need to be considered. Additionally, we add all states in $\mathit{Reach}(NR(s_2))$ and $\mathit{Reach}(NR(s_4))$ to compute $\approx$-equivalence for states in $NR(s_2)$ and $NR(s_4)$. The copy of s_3 is completely located outside $\mathit{Reach}(\{s_0\} \times Q)$ and, thus, needs not be considered.

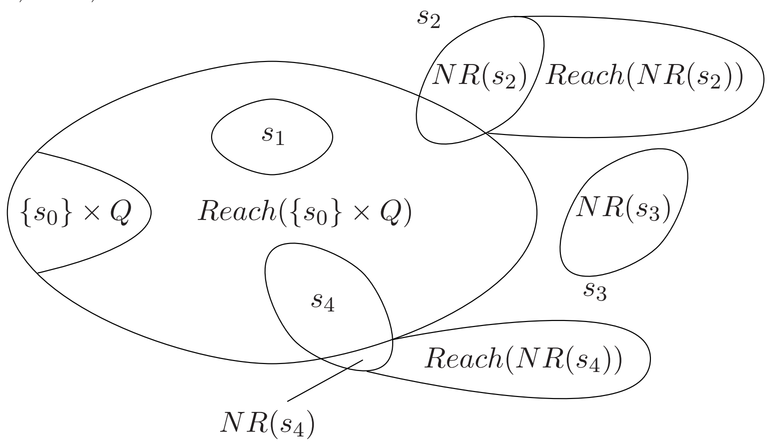

s_2
$NR(s_2)$
$\mathit{Reach}(NR(s_2))$
s_1
$\{s_0\} \times Q$
$\mathit{Reach}(\{s_0\} \times Q)$
$NR(s_3)$
s_3
s_4
$\mathit{Reach}(NR(s_4))$
$NR(s_4)$

Figure 6.3: *Computation of* $\approx_S$

If we add non-reachable states as described above then we speak of the *extended* version of Algorithm 3.4. Since in the first approach (the "normal" version) presented at the beginning of this section only reachable states need to be considered the computation time is much faster than for the second approach (the “extended” version). Nevertheless, we have decided to use both. A major reason for this is that the second approach often yields better results. Whereas for Example 5.3 the first approach only needs to consider 592 states (and 889 transitions) and yields a memory of size 37 the second approach considers 4793 states (and 7226 transitions) but at the same time improves the memory reduction to the size of 19. But it is important to note that neither of the two approaches dominates the other one. There are also test cases where the first approach yields better results. An example for that is the Staiger-Wagner game used in the proof of Remark 7.1. Though, the difference to the result of the extended version of our algorithm is only a small constant there.

If we use the extended version of our Algorithm then there is one problem with the definition of $s_{\min}$ (cf. page 43). Consider an edge $((s,q),(s',q'))$ and let $s_{\min}$ be the minimal memory content which is $\approx_S$-equivalent to s'. If $([s],q)$ is reachable then $([s_{\min}],q')$ and all vertices reachable from it are reachable in the quotient game automaton. Since $(s_{\min},q')$ needs not be reachable in the given game automaton the number of reachable memory contents in the quotient game automaton compared to the given game automaton may increase, consequently. To overcome this we modify the definition of $s_{\min}$ in the sense that we choose only from the reachable target states.

$$s_{\min} := \min\{\hat{s}' \mid \exists \hat{s} : \hat{s} \approx_S s \text{ and } \delta((\hat{s},q),q') = (\hat{s}',q'), (\hat{s}',q') \text{ is reachable}\}.$$

If none of the target states is reachable then neither is one of the source states. In this case we can redirect the transition to any state without changing the language recognised by the quotient game automaton. Then we set $s_{\min} := s_0$. In the implementation we use the above definition of $s_{\min}$ but if none of the target states is reachable then we simply leave out the transition. Note that the refined definition of $s_{\min}$ has no influence on the quotient game automaton if we consider the normal version of our algorithm. This is due to the fact that in this case all considered states are reachable.

6.2 The Tool GASt

The memory reduction algorithms worked out in this thesis have been integrated in an existing tool, called “GAS*t*”, which is an abbreviation for “Games, Automata and Strategies”. This tool is predominantly implemented in JAVA and has been developed by Nico Wallmeier since 2002. The first version was the result of his diploma thesis finished in January 2003 (cf. [Wal03]). Many features have been added since then. The tool was renamed in October 2006. Prior it was called “SymProg”. GAS*t* provides various algorithms for synthesis problems over infinite games and automata. It can solve all common types of games, e.g.:

- Reachability games
- Safety games

- Büchi games
- Parity games (strong and weak version)
- Staiger-Wagner games
- Request-Response games
- Muller games

For all included games the solution algorithms are implemented on the enumerative level. This means that e.g. the set of vertices of a game graph is explicitly enumerated. Most algorithms are also provided on the symbolic level, i.e. the underlying data structures are represented by binary decision diagrams (BDDs). The input must also be encoded by a binary representation. As the solution to an infinite game GAS*t* prints out the winning regions and corresponding winning strategies for the two players. For solving those types of games where positional winning strategies do not suffice the common game reduction algorithms are implemented. These algorithms can be found e.g. in [TL03] or [WHT03].

Another set of features provided by GAS*t* is the determinisation of ω-automata, e.g. the constructions of Safra or Muller/Schupp, and the tool ReMORDS. Since these parts of GAS*t* are irrelevant for us we do not go into details about them here. For further information about the fundamentals on the different types of games implemented in GAS*t*, the symbolic algorithms and the implementation itself we refer to the diploma thesis of Nico Wallmeier (cf. [Wal03]).

Clearly, the implementation of Algorithm 3.4 can be realised directly on the given data structures. We do not need to introduce the auxiliary states q_0 and q_{sink}. Nor do we have to use labels for the transitions because they can be read off the target states of the transitions (cf. Definition 3.5). These things were only necessary to prove the formal correctness of our algorithm.

Our algorithm is integrated as follows. Our following remarks refer to both Request-Response and Staiger-Wagner games where we have implemented our algorithm only on the enumerative level. At first the adequate game reduction algorithm is applied to Γ which means the creation of the extended game graph G' and the winning condition φ'. This is initiated by the method *solve* implemented in the class *AbstractGame*. Our algorithm is encapsulated in an own class and gets G' and φ' as input. It returns the game graph G'' and the winning condition φ''. Then an algorithm for solving the game $\Gamma'' = (G'', \varphi'')$ is applied and the result is printed out. The call sequences that compute the memory reduction in both our algorithm and the minimisation algorithm for strategy automata are explained in more details in the following section.

6.3 The Memory Reduction Algorithms

6.3.1 Request-Response Games

The game reduction algorithm from Request-Response to Büchi games (cf. Section 2.2.4) as well as the algorithms to compute the sets $Attr_0^+(\cdot)$ and $Recur_0(\cdot)$ (cf. Section 2.2.1) have already been implemented in GAS*t*. Hence, steps one, two and five from Algorithm 5.2 are skipped here and we focus only on steps three and four. In the class *RequestResponseGameEnum* the boolean flag *memoryReduction* is queried to check whether the memory reduction algorithm has been activated on the user interface. If it has been activated then the first thing to do is to add all non-reachable states which have a reachable memory content and all states reachable from these states. This is necessary to be able to compute the equivalence relation $\approx_S$ on S properly (cf. Section 6.1). Since the computation of the quotient game graph depends on the fact which states are reachable and which are not we store this information in two sets (*reachableStates* and *nonReachableStates*) and pass it to the class *RequestResponseMemoryReductionAlgorithm* when we initiate the actual memory reduction procedure. It is carried out by the method *reduceMemory* which is inherited from the super class *MemoryReductionAlgorithm*. The subclass has a member variable *gg* of type *GameGraphEnum*. It encapsulates the enumerative representation of the game graph of the reduced game. It is initialised with the game graph G' of the reduced game Γ' and manipulated by the memory reduction algorithm. At the end of the computation it contains the quotient game graph[2] $G'' = \mathcal{A}_{\Gamma'}/_{\approx_S}$. The method *reduceMemory* is rather simple. It initiates the following subroutines:

1.) computeClosure : This method implements the closure operator from page 52 and terminates with the fix point $cl(\mathcal{A}_{\Gamma'})$. This is step three from Algorithm 5.2.

2.) computeEquivalentStates : This method computes the direct bisimulation relation $\approx_{di}$ on $cl(\mathcal{A}_{\Gamma'})$. Since we have a deterministic automaton this can be done with the block refinement algorithm by Hopcroft (cf. [Hop71]). From the result we obtain the relation $\approx_S$ on the set S of memory contents. The object *equivalenceClasses* stores a listing of the computed $\approx_S$-classes. This is the first part of step four from Algorithm 5.2.

3.) computeQuotientGameGraph : This method computes the quotient game automaton $cl(\mathcal{A}_{\Gamma'})/_{\approx_S}$. We choose a fixed total order $\prec_S$ on S and each class $[s] \in S/_{\approx_S}$ is represented by its $\prec_S$-minimal element. For that transitions are redirected by replacing source or target state (if necessary). After that all states with no incoming transitions are deleted. We perform an additional depth-first search on the remaining graph to update the set of reachable states. All other states are deleted afterwards as well. The set of final states is stored

[2]For simplicity we abstain from distinguishing between game automata and automaton games.

in the object *setOfFinalStates* and is updated accordingly. This completes step four of Algorithm 5.2.

The resulting game graph is returned to the class *RequestResponseGameEnum*. As a last step the set of final states stored in the member variable *finalset* in the class *RequestResponseGameEnum* is updated to remove all non-reachable states in it. After that the resulting Büchi game is solved with the algorithm from Section 2.2.1.

We finish this section with some remarks on the data structures used for implementing a reduced game graph. A big problem while implementing the memory reduction algorithms was the data structure used for the vertices of a reduced game graph. Since GAS*t* implements several game reduction algorithms it is useful to implement a super class *ReducedState* that encapsulates a vertex of a reduced game graph. This class should have two member variables. One variable *oldstate* which stores the Q-component of the vertex and one abstract variable which represents the memory component. There are several classes, e.g. *ReducedStateStaigerWagner*, *ReducedStateRR* or *ReducedStateLAR*, which are subclasses of this class. Unfortunately, the super class does not have a member variable for the abstract memory. The reason for this is that the memory is not encapsulated in a separate class. Each subclass introduces a new member variable and methods to work with it. This has caused numerous problems.

In our algorithms a lot of operations are carried out directly on the memory but not on reduced states. Hence, we use helping methods that manipulate memory as objects. Especially for the memory reduction algorithm for Request-Response games this is problematical. We have introduced a set containing the three memory components from the game reduction algorithm on page 26. When working with this set we have discovered a lot of side effects. Sometimes two copies of one and the same memory are detected to be distinct when appearing in different vertices. To overcome this we had to make lots of effort to implement methods which test memory contents on equality. This problem could be solved by encapsulating the memory of a reduced state in an own class.

6.3.2 Staiger-Wagner Games

Most of the data structures is the same as for the memory reduction algorithm for Request-Response games, e.g. the objects *reachableStates* and *nonReachableStates*. The major difference is the method *reduceMemory*. There, we have to compute the graph of strongly connected components of G' and a maximal colouring for it. The rest of the algorithm, i.e. determine the equivalent pairs of states and memory contents and the computation of the quotient game graph is the same as above. The method *reduceMemory* realises steps three to five from Algorithm 5.4:

1.) csccgg = new ColoredSCCGameGraphEnum(gg) : A new object called *csccgg* of type *ColoredSCCGameGraphEnum* is created. This class encapsulates an SCC-graph of a coloured game graph. The construction of the SCC-graph is initiated directly in the constructor of this class by a call of the method *detectSCCs*. It has two subroutines. The first one, called *mapNodesToSCCs*, detects which nodes of the graph belong to the same SCC. For this we have implemented the SCC detection algorithm from [Sed91]. To store the SCCs we use the class *SCCEnum*. The second method, called *detectConnectivity*, enumerates the successors and predecessors for each SCC and stores them in the corresponding *SCCEnum*-object. Unfortunately, there seems to be no possibility to compute the set of vertices and the transition structure of the SCC-graph simultaneously.

2.) gg = csccgg.computeMaximalColouring : This method implements Algorithm 4.27 on the SCC-graph *csccgg*. We do not need to topologically sort the set of SCCs with respect to the edge relation. Instead we perform a depth-first search on this set starting from those SCCs that have no predecessors. If we reach a leaf SCC in the search tree we assign a colour to it and then backtrack to the parent we came from. This procedure guarantees that an SCC is treated after all of its successors. If we have assigned a value to an SCC then we change the colours of all of its nodes in the original game graph. For step four of Algorithm 5.4 we can consider states with even number as final and states with odd number as non-final. Hence, we do not need an explicit representation of a set of final states. At the end of this step we have a normalised DWA $\mathcal{A}$.

3.) computeEquivalentStates : From this point on the algorithm works analogously to the one for Request-Response games from the last section. We compute the equivalence relation $\approx_{\mathcal{A}}$ known from the theory on minimisation of DFA. This is done with the block refinement algorithm by Hopcroft (cf. [Hop71]). From the result we obtain the relation $\approx_S$ on the set S of memory contents. The object *equivalenceClasses* stores a listing of the computed $\approx_S$-classes. This is the first part of step four from Algorithm 5.4.

4.) computeQuotientGameGraph : The method computes the quotient game automaton $\mathcal{A}/_{\approx_S}$. Again, we take a fixed total order $\prec_S$ on S and represent each class $[s] \in S/_{\approx_S}$ by its $\prec_S$-minimal element. All transitions are redirected accordingly by replacing source or target state (if necessary). After that we perform an additional depth-first search on the remaining graph to update the set of reachable states. All non-reachable states are deleted afterwards. This completes step four of Algorithm 5.4. The resulting quotient automaton has the colouring $c/_{\approx_S}$ as defined on page 69. Thus, step five is immediately accomplished.

The resulting quotient game graph is returned to the method *createResultGG* in the class *StaigerWagnerGameEnum*. There it is stored in the local variable *cgg* of type *ColoredGameGraphEnum*. The new colours from the maximal colouring are maintained for the remaining vertices. The corresponding weak parity game is solved with the algorithm from Section 2.2.2.

6.3.3 The Marking Algorithm

One aim of this thesis is to compare different approaches for memory reduction. In particular, we consider the minimisation algorithm for strategy automata and want to find out for which examples it yields better results than our new approach, and vice versa. Hence, we have also implemented Algorithm 3.3. Algorithm 3.4 reduces the memory component S of a reduced game graph and then computes a winning strategy on the resulting game graph. In contrast to that the minimisation algorithm for strategy automata has as input a fixed strategy f and returns the minimal automaton implementing it. Hence, it is also possible to compose both algorithms.

In GASt each class that implements an enumerative infinite game inherits the method *solve* from the abstract super class *AbstractGameEnum*. It computes a solution to a given game and returns an object of type *GameResultEnum*. This object contains the winning regions of the game, i.e. a partition of the set of vertices of the given game graph, and positional winning strategies, represented as a mapping from the set of vertices into itself. For reduced games the winning regions contain vertices which consist of a memory content $s \in S$ and a vertex $q \in Q$. From that the winning regions of the original game can be read off. The positional winning strategies each are a subset of the set of transitions in the reduced game graph.

We have implemented the marking algorithm for minimising strategy automata directly on the game result object. We take each memory content of some vertex in the strategy object and from that create the set of states of the strategy automaton under consideration. We add a transition from state s_1 to state s_2 if and only if there exist $q_1, q_2 \in Q$ such that $((s_1, q_1), (s_2, q_2))$ is a transition in one of the two strategies. If such a transition exists then q_1 is the label of the transition and q_2 the corresponding output of the automaton. Though, this does not yield a complete transition table. Some transitions are missing, e.g. transitions that are irrelevant for the winning strategies. If we consider the game graph from Figure 3.5 then the vertex $(\{0, 1, 2\}, 1)$ is not reachable in G'. Hence, there is no outgoing transition from state $\{0, 1, 2\}$ labelled by 1.

The first thing we have to do before we can apply the minimisation algorithm to the strategy automaton is to extend the transition table to get a deterministic transition structure. Otherwise, if we apply the algorithm with the transitions as given then we can compare pairs of states only for those transition labels for which there exists an outgoing transition from both states under consideration. But this can yield a non-transitive relation. The minimised automaton would not be correct.

We use a heuristic to add the missing transitions. We perform a sequential search on the list of states. If we discover a pair of states where all shared transition labels yield the same output then we match the missing transitions of the two states as follows. If a state has an outgoing transition for a label missing for the other one then we copy it. If a transition label is missing for both states then we add one each. For that, we copy twice the transition from s_0 with the respective label.

For the actual computation of the equivalence relation we use a simple marking algorithm as presented in pseudo-code in Algorithm 3.3. In the initialisation we consider all pairs $(s_1, s_2) \in S \times S$. For each such pair and each $q \in Q$ we compare the values $\tau(s_1, q)$ and $\tau(s_2, q)$. The pair (s_1, s_2) is marked if and only if there exists $q \in Q$ such that $\tau(s_1, q) \neq \tau(s_2, q)$. The value $q'_1 := \tau(s_1, q)$ (resp. $q'_2 := \tau(s_2, q)$) is the Q-component of the target state reached from (s_1, q) (resp. (s_2, q)) in the positional strategy of the respective player. This means there exists $s'_1 \in S$ (resp. $s'_2 \in S$) such that $((s_1, q), (s'_1, q'_1))$ (resp. $((s_2, q), (s'_2, q'_2))$) is a transition in the positional winning strategy implemented by the automaton under consideration. After the initialisation we loop over all pairs of states and all input letters $q \in Q$ and search for pairs to be newly marked. The algorithm terminates if there is a loop in which no pair is marked. After we have computed the equivalence classes we manipulate the strategy object returned by the method *solve* such that we delete some of its transitions. We choose a fixed order on the state set and represent each equivalence class by its minimal element. Hence, we can delete all states which are not minimal in their equivalence class and all outgoing transitions. The strategies implemented by the reduced strategy automaton are then printed out on the user interface.

Chapter 7

Comparison of the two Approaches

In this chapter we want to compare the minimisation algorithm for strategy automata with our new approach of memory reduction. Before we do so we have to make an important remark about the starting points of both algorithms. A major difference is that our new algorithm reduces the memory prior to solving the game whereas the other algorithm takes a computed solution, i.e. a strategy, and tries to implement it with as little memory as possible. But it could be that this strategy is far too complicated for the game under consideration. Hence, if the latter algorithm fails to reduce the memory to whatever size then it is actually the fault of the algorithm which was used to solve the game. The minimisation algorithm for strategy automata is only as good as the complexity of the given strategy allows for.

In the sequel we present three examples which illustrate the pros and cons of both the considered approaches. In all three cases we consider parameterised games where the game graphs are of linear and the winning conditions are of exponential size in the parameter. We start with a simple example where we focus only on our new algorithm. There we reduce the memory provided by a game reduction algorithm from exponential to constant size. This shows that our new method of memory reduction is capable of detecting arbitrarily many redundant memory contents. This result is improved in Section 7.2.1. Again, we consider a family of example games for which our algorithm yields a strategy automaton of constant size. But the minimisation algorithm will fail to do so and return an automaton of exponential size. This is due to the fact that the positional winning strategy on the game graph resulting from the game reduction algorithm is very complicated. Our new algorithm first simplifies this game graph and then computes the strategy on the simplified graph. Hence, it is of great importance whether to reduce the memory before or after solving a game. In Section 7.2.2 we present an example where the situation is converse. The new approach does not break the exponential size of the game graph under consideration but nevertheless we obtain a positional winning strategy for Player 0. The shortcoming of our algorithm in this example is that it has to consider all possible plays from a given vertex. The problem there is that the algorithm uses the more general model of an automaton and, hence, cannot take into account the behaviour of the two players.

7.1 A simple Upper Bound

Remark 7.1. There is a family $\Gamma_n = (G_n, \varphi_n)$ of Staiger-Wagner games such that the following hold:

1. Let $\Gamma'_n = (G'_n, \varphi'_n)$ be the reduced game of Γ_n. Then the number of memory contents in the reachable subgraph of G'_n grows exponentially in n.
2. The strategy automaton returned by Algorithm 5.4 is of constant size.

Proof. The Staiger-Wagner game graph G_n can be seen in Figure 7.1. Let R_n denote the set of all non-empty subsets of $\{1, \dots, n\}$, i.e. $R_n = \{R \mid R \subseteq 2^{\{1,\dots,n\}}, R \neq \emptyset\}$. We use the usual Staiger-Wagner winning condition φ_n with $\mathcal{F}$ as follows:

$$\mathcal{F} := \{R \cup \{0\} \mid R \in R_n\} \cup \{R \cup \{0, n+1, v\} \mid R \in R_n, v := \varrho(R)\}$$

where

$$\varrho : R_n \to \{n+2, n+3\}$$

$$\varrho(R) := \begin{cases} n+2 & \text{, if } 1 \in R \\ n+3 & \text{, otherwise} \end{cases}$$

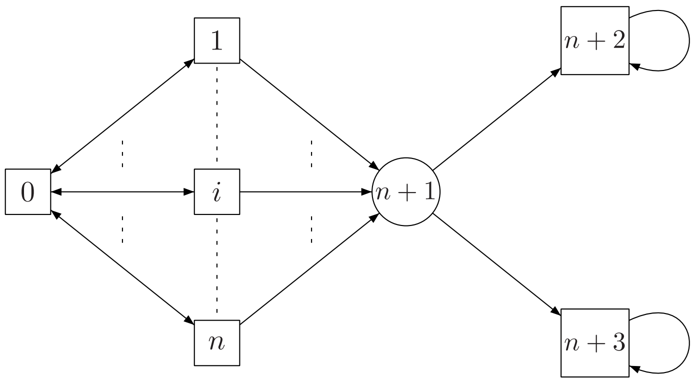


Figure 7.1: *Staiger-Wagner game graph G_n*

That means Player 0 wins any play that stays either in the subset $\{0, \dots, n\}$ or if it reaches vertex $n + 1$ proceeds to vertex $n + 2$ if and only if vertex 1 has been visited before. Note that Player 0 can win this game with a memory of size two. G_n has $n + 4$ vertices and in G'_n there exist $4 \cdot 2^n + 4 \cdot n + 3$ reachable memory contents.

This verifies item 1. of Remark 7.1. To prove item 2. we give an example for $n = 3$ of the set $S/_{\approx_S}$ returned by the normal version of Algorithm 5.4. This version returns thirteen $\approx_S$-equivalence classes whereas the extended version computes seventeen classes. For two memory contents $s_1, s_2 \in S$ we have:

$$\begin{aligned} s_1 \approx_S s_2 \iff \forall q \in Q : &(s_1, q) \text{ is reachable} \iff (s_2, q) \text{ is reachable and} \\ &\text{if } (s_1, q) \text{ is reachable then } (s_1, q) \approx (s_2, q) \end{aligned}$$

This means two memory contents s_1, s_2 are $\approx_S$-equivalent if and only if in the reduced game graph from each pair of vertices with the same Q-component within the copies of s_1 and s_2 exactly the same plays are winning for Player 0. If this is the case s_1 and s_2 are considered equivalent and, accordingly, can be merged in G'_n. For our example the equivalence classes look as follows:

$$\begin{aligned}
&\text{Class } 1 : \{\{0,1\},\{0,1,2\},\{0,1,3\},\{0,1,2,3\}\} \\
&\text{Class } 2 : \{\{0,1,4\},\{0,1,2,4\},\{0,1,3,4\},\{0,1,2,3,4\}\} \\
&\text{Class } 3 : \{\{0,1,4,5\},\{0,1,2,4,5\},\{0,1,3,4,5\},\{0,1,2,3,4,5\}\} \\
&\text{Class } 4 : \{\{0,2\},\{0,3\},\{0,2,3\}\} \\
&\text{Class } 5 : \{\{0,2,4\},\{0,3,4\},\{0,2,3,4\}\} \\
&\text{Class } 6 : \{\{0,2,4,6\},\{0,3,4,6\},\{0,2,3,4,6\}\} \\
&\text{Class } 7 : \{\{0,2,4,5\},\{0,3,4,5\},\{0,2,3,4,5\}, \\
&\qquad\qquad \{1,4,5\},\{2,4,5\},\{3,4,5\},\{4,5\},\{5\}\} \\
&\text{Class } 8 : \{\{0,1,4,6\},\{0,1,2,4,6\},\{0,1,3,4,6\},\{0,1,2,3,4,6\}, \\
&\qquad\qquad \{1,4,6\},\{2,4,6\},\{3,4,6\},\{4,6\},\{6\}\} \\
&\text{Class } 9 : \{\{4\},\{1,4\},\{2,4\},\{3,4\}\} \\
&\text{Class } 10 : \{\{1\}\} \\
&\text{Class } 11 : \{\{2\},\{3\}\} \\
&\text{Class } 12 : \{\emptyset\} \\
&\text{Class } 13 : \{\{0\}\}
\end{aligned}$$

Note that the normal version of our algorithm requires for two memory contents s_1, s_2 to be equivalent that exactly the same Q-vertices are reachable within the copies of s_1 and s_2. This is a transitive property which transfers to all memory contents of a class. For example, all memory contents in classes one and four are reachable exactly at the Q-components in $\{0, \dots, 4\}$. No other class contains a memory content which is reachable at this set of Q-vertices. Of course, if two memory contents are reachable at the same Q-components then this does not mean that they are $\approx_S$-equivalent. For classes one and four we will shortly see this. We are mainly interested in memory contents which contain 0 and a set from R_n because these memory contents are reachable in plays starting at vertex 0 and we want to analyse which of them are detected equivalent by our algorithm. We have sorted the classes above conveniently. If we consider Γ_n for arbitrary n then the number of memory contents which are not of the aforementioned form is linear in n. They can be found in classes seven and eight and do constitute all of classes nine to thirteen. We focus on classes one to eight, especially classes one to six.

Classes one and two contain successor memory contents in the sense that if we are at a vertex which has a memory content from class one and as Q-component vertex 4 then in the next move we reach a vertex with memory content from class two. E.g. the vertex $(\{0,1\},4)$ has the successors $(\{0,1,4\},5)$ and $(\{0,1,4\},6)$. This holds for classes two and three, four and five and for the classes five and six analogously. The memory contents from class one are not equivalent to the memory contents from class four. Starting from one or the other different sets of plays are winning for Player 0. From all memory contents in class one exactly the same set of plays are winning for Player 0. This holds also for class four but if we consider memory contents from both classes one and four then this is no longer true. The play suffix 5^ω is winning for Player 0 from $(\{0,1\},4)$ because this play eventually reaches vertex $(\{0,1,4,5\},5)$ and loops in it until infinity. We get $Occ(\rho) = \{0,1,4,5\}$ and since $\{0,1,4,5\} \in \mathcal{F}$ Player 0 wins this play. But 5^ω is losing for Player 0 from $(\{0,2\},4)$ because $\{0,2,4,5\} \notin \mathcal{F}$. This means from $(\{0,1\},4)$ the corresponding game automaton accepts the suffix 5^ω and from $(\{0,2\},4)$ rejects[1] it. We get $(\{0,1\},4) \not\approx (\{0,2\},4)$ and, accordingly, $\{0,1\} \not\approx_S \{0,2\}$. Of course, Player 0 has a winning strategy from both memory contents but our algorithm has to distinguish between them. Analogous considerations can be made with classes two and five. Classes three and six contain all memory contents where Player 0 has already won.

Classes seven and eight contain all memory contents from which Player 0 only loses where we have written each class in two lines. The memory contents from the second lines do not contain 0 and, thus, are not interesting for us. The first lines are those memory contents where Player 0 has taken the wrong decision at vertex 4. This means he has chosen to move to vertex 6 if and only if vertex 1 was visited before.

The reader may verify that for values of n larger than 1 the set $S/_{\approx_S}$ has 13 elements. Hence, the strategy automaton returned by Algorithm 5.4 is of constant size. This verifies the second claim of Remark 7.1 above. Classes twelve and thirteen remain as they are where the latter class is reachable at Q-vertices $\{1,\dots,n\}$. Class eleven generalises to $\{\{2\},\dots,\{n\}\}$ and class nine to $\{\{n+1\},\{1,n+1\},\dots,\{n,n+1\}\}$. Class eight contains all memory contents reachable only at Q-vertex $n+3$ and from where Player 0 always loses. Class three contains all memory contents reachable only at Q-vertex $n+2$ and from where Player 0 always wins. Similar considerations yield analogous descriptions for the other classes. Classes ten, twelve and thirteen have constant size where classes nine and eleven grow linearly in n. The sizes of the remaining eight classes are between $2^{n-1}-1$ and $2^{n-1}+n+2$. □

Tables A.1 to A.4 (cf. Appendix A) break down the computation of our implementation[2] of Algorithm 5.4 on the above example. In the first two tables we consider the normal version and in tables three and four the extended version. For a description of the difference between both versions we refer to Section 6.1. For

[1] For 6^ω the situation is converse.

[2] For further details about the used hardware and software we refer to Appendix B.

each version we give the size of the game graphs G_n, G'_n and G''_n and the computation times for the different steps of Algorithm 5.4. In steps one and two the reduced game Γ'_n is created. We do not need to represent the game automaton $\mathcal{A}_{\Gamma'_n}$ explicitly because its transition structure and final states are uniquely determined by the game reduction algorithm from Section 2.2.5 and Lemma 4.30. Step three transforms $\mathcal{A}_{\Gamma'_n}$ into normal form where we give the computation times for the detection of the SCC-graph and the maximal colouring. Steps four and five involve the computation of the equivalence relation $\approx_S$ and the quotient automaton. In the final step six we solve the resulting automaton game Γ''_n.

Note that since S' is of constant size the resulting reduced game graph G''_n is of linear size. We are at a loss why for both the normal and the extended version the time to solve the game Γ''_n grows so rapidly. This must be due to some bug in the method for computing the quotient game graph or in the algorithm for solving weak parity games. For the normal version and $n = 10$ the number of reachable memory contents can be computed but afterwards we get an out of memory error on the *JAVA* heap space. In Table A.1 the set S denotes the set of reachable memory contents in G'_n whereas in Table A.3 it denotes the set of reachable memory contents extended by all memory contents that are reachable from a vertex with a reachable memory content. This is because we consider the extended version of our algorithm (cf. page 84). In Table A.1 we also give the results of the minimisation algorithm for strategy automata. Compared to our new approach it yields a memory which is approximately $O(n)$ larger.

7.2 Memory Reduction vs. Strategy Minimisation

Remark 7.1 shows that with our new algorithm it is indeed possible to abolish the enormous memory that is required by a game reduction algorithm. This is satisfying because we have not only broken through the exponential bound but even obtained a result that is of size $O(1)$, i.e. independent of n. In the above example the minimisation algorithm for strategy automata yields a strategy of size $O(n)$ (cf. Table A.1). By the size of a strategy we mean the minimal number of states that is necessary to implement the strategy by an automaton with output. Even though our algorithm yields the better result the minimisation algorithm for strategy automata is only little worse. Our aim is to strengthen this by studying a further example of a Staiger-Wagner game. There, we show that our algorithm also yields a memory of constant size but the minimisation algorithm for strategy automata returns a strategy of exponential size. This is due to the fact that the winning strategy computed by the algorithm from Section 2.2.2 is much too complicated.

7.2.1 A Shortcoming of the Attractor Construction

For the proof we use the game graph $G_n = (Q_n, E_n)$ from Figure 7.2 together with the following Staiger-Wagner set $\mathcal{F}$:

$$\mathcal{F} = \{U \mid U \subseteq \{v, u_1, \dots, u_n\}, v \in U\} \cup \{R \mid R \supseteq \{x, y\}\}$$

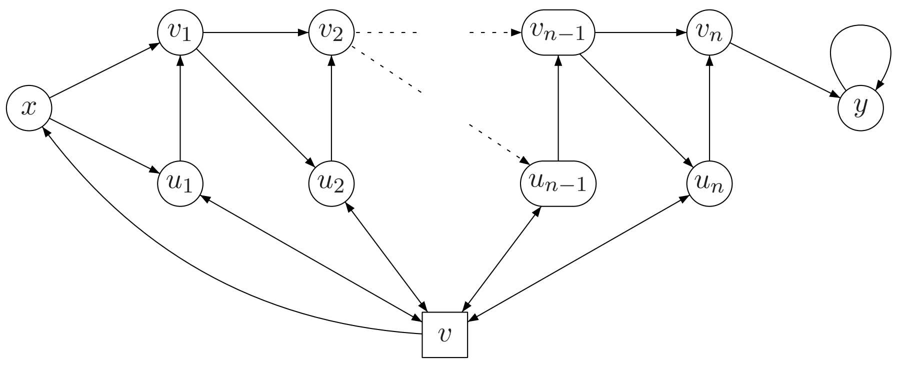


Figure 7.2: *Staiger-Wagner game graph G_n*

If we apply the game reduction algorithm from Section 2.2.5 to the above example then we obtain a weak parity game $\Gamma'_n = (G'_n, \varphi'_n)$. Solving Γ'_n with the algorithm from page 19 we obtain a positional winning strategy f'_n for Player 0 in Γ'_n. For this we assume that the attractor sets are computed by a backward breadth-first search as e.g. in Algorithm 2.6. This is sufficient to obtain attractor strategies as defined on page 15. From f'_n we obtain a finite-state winning strategy f_n for Player 0 in Γ_n. We show that this strategy f_n is of exponential size. If, however, we use our new algorithm to reduce G'_n before we compute f'_n then the corresponding strategy f_n is of constant size. This shows that there exist examples for which a reduction of the used memory prior to solving the game yields substantially better results than doing afterwards.

Lemma 7.2. *Let $\Gamma_n = (G_n, \varphi_n)$ be the Staiger-Wagner game from Figure 7.2 and let $\Gamma'_n = (G'_n, \varphi'_n)$ be the reduced weak parity game of Γ_n according to the game reduction algorithm from Section 2.2.5. Then Player 0 wins from vertex v such that the following hold:*

1. *The positional winning strategy f'_n for Player 0 in Γ'_n from $(\emptyset, v)$ returned by the algorithm from Section 2.2.2 yields an exponential winning strategy f_n for Player 0 in Γ_n from v.*
2. *The reduced game graph G''_n computed by the extended version of Algorithm 5.4 consists of constantly many copies of Q_n.*

Proof. The winning regions of this game are $W_0 = \{x, v, u_1, \dots, u_n\}$ and $W_1 = \{v_1, \dots, v_n, y\}$. It is easy to see that both players have positional winning strategies from their winning regions. Player 0 wins as follows:

"From $u_1, \dots, u_n$ move to v and from $x, v_1, \dots, v_n$ move straight towards y."

For the proof we only consider plays that start in vertex v where we assume that Player 1 moves to vertex x after finitely many steps. Otherwise the play stays in $\{v, u_1, \ldots, u_n\}$ which means that Player 0 wins. We set $U_v := \{U \mid U \subseteq \{v, u_1, \ldots, u_n\}, v \in U\}$. We divide each play ρ that starts at vertex v and eventually proceeds to vertex x into two parts, the initial part up to the visit to vertex x and a second part as everything visited thereafter. Accordingly, ρ has the form

$$\rho = \underbrace{v u_{i_1} v \ldots u_{i_j} v}_{\rho_1} x \ldots$$

where $1 \leq i_l \leq n$ $(1 \leq l \leq j)$ and $j \in \mathbb{N}$. Now, to verify item 1. of the lemma let us analyse the algorithm for solving weak parity games from Section 2.2.2 (cf. page 19). G_n has $2 \cdot n + 3$ vertices. Accordingly, the highest possible (even) colour in G'_n is $k := 2 \cdot (2 \cdot n + 3)$. First the algorithm computes $A_k = Attr_{G'_n,0}(C_k)$. Obviously, each vertex (U, x) for $U \in U_v$ in G'_n is contained in A_k. For any play ρ that has an initial part of the form like ρ_1 Player 0 reaches the set C_k in G'_n by visiting the following states in G_n from x:

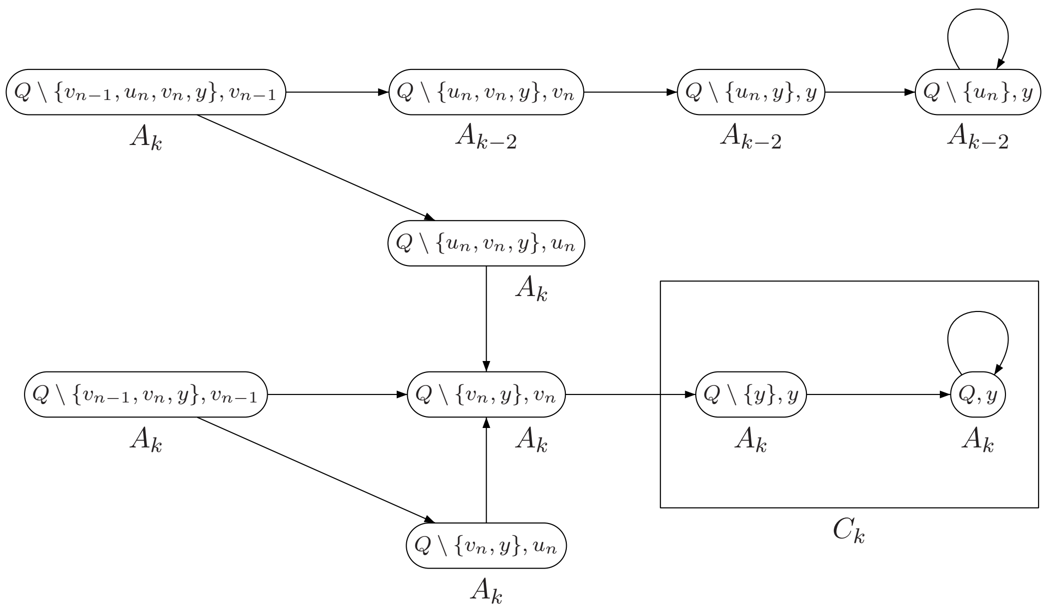
$$u_1, v_1, u_2, v_2, \ldots, u_{n-1}, v_{n-1}, u_n, v_n, y$$

Then all vertices of G_n have been visited which means $Occ(\rho) = Q$ and in G'_n we reach a vertex in C_k. Even if this simple strategy leads Player 0 into C_k this is not the strategy computed by the attractor algorithm from Section 2.2.2. To see this consider the situation in Figure 7.3 below where we give a small part of G'_n. There, we assume that in ρ_1 we have visited the vertices $u_1, \ldots, u_{n-1}$ (and v) but not u_n and on the way from x to v_{n-1} the vertices $v_1, \ldots, v_{n-2}$ have been visited as well. This means that all vertices except for v_{n-1}, u_n, v_n and y have been visited and we are now in the leftmost upper vertex. Analogously, the leftmost lower vertex is reached if u_n has been visited in ρ_1 as well.

It is easy to determine which of the displayed vertices belong to A_k and which belong to A_{k-2}. Now, let us follow the operation of the algorithm on this subgraph of G'_n. It first computes the set A_k by performing a backward breadth-first search from C_k and after that it computes the sets A_j for j decreasing from k. This approach to solving weak parity games is one reason why we will obtain an exponential strategy in this example. Generally speaking, assume Player 0 from a given vertex has two winning strategies f_1, f_2 where in one case we get $Occ(\rho_1) = R_1$ and in the other $Occ(\rho_2) = R_2$. If $|R_1| > |R_2|$ then the attractor algorithm always prefers f_1 to f_2 even if f_1 is much more complicated than f_2. In our example, from the leftmost upper vertex to get into C_k the algorithm returns the attractor strategy with the following transitions:

$$\begin{aligned} (Q \setminus \{v_{n-1}, u_n, v_n, y\}, v_{n-1}) &\longrightarrow (Q \setminus \{u_n, v_n, y\}, u_n) \\ (Q \setminus \{u_n, v_n, y\}, u_n) &\longrightarrow (Q \setminus \{v_n, y\}, v_n) \\ (Q \setminus \{v_n, y\}, v_n) &\longrightarrow (Q \setminus \{y\}, y) \end{aligned}$$

Considering only the Q-components of the visited vertices we get the finite-state strategy for Player 0 in Γ_n. This strategy tells Player 0 to move from v_{n-1} to v_n by



$Q \setminus \{v_{n-1}, u_n, v_n, y\}, v_{n-1}$
A_k
$Q \setminus \{u_n, v_n, y\}, v_n$
A_{k-2}
$Q \setminus \{u_n, y\}, y$
A_{k-2}
$Q \setminus \{u_n\}, y$
A_{k-2}
$Q \setminus \{u_n, v_n, y\}, u_n$
A_k
$Q \setminus \{v_{n-1}, v_n, y\}, v_{n-1}$
A_k
$Q \setminus \{v_n, y\}, v_n$
A_k
$Q \setminus \{y\}, y$
A_k
Q, y
A_k
C_k
$Q \setminus \{v_n, y\}, u_n$
A_k

Figure 7.3: *Staiger-Wagner game graph* G_n

an indirection via u_n.

Unlike in the first case, from the leftmost lower vertex the attractor strategy tells Player 0 to move from $(Q \setminus \{v_{n-1}, v_n, y\}, v_{n-1})$ directly to $(Q \setminus \{v_n, y\}, v_n)$ without an indirection via $(Q \setminus \{v_n, y\}, u_n)$ and then from $(Q \setminus \{v_n, y\}, v_n)$ to $(Q \setminus \{y\}, y)$. This is because the attractor algorithm decreases the distance to C_k in each step and, thus, chooses the straightest way into C_k. For a vertex $q \in A_k$ the distance to C_k is the smallest i such that $q \in Attr_0^i(C_k)$ (cf. Section 2.1.3). Hence, in this case the finite-state strategy for Player 0 in Γ_n is to move from v_{n-1} directly to v_n. Altogether, the strategy f_n constructed from the result of the attractor algorithm tells Player 0 to move from v_{n-1} to v_n if and only if u_n has been visited in ρ_1.

An analogous argument can be given for any vertex u_i which means that on the way from x to y exactly those states u_i are visited that have not been visited in ρ_1. Each set $U \in U_v$ determines a unique strategy f_U and, accordingly, we get 2^n different strategies. For each f_U any strategy automaton which implements f_n needs one state. Since for two subsets $U_1 \neq U_2$ we get distinct strategies each pair of states is non-equivalent. Thus, f_n is of exponential size. This completes the proof of the first item of Lemma 7.2.

To prove item 2. we have to determine which memory contents are $\approx_S$-equivalent and which are not. We are especially interested in sets from U_v because we have seen above that they are all detected non-equivalent by the minimisation algorithm for strategy automata. Given the sets $U_1 \neq U_2$ from U_v and w.l.o.g. $u_i \in U_1 \setminus U_2$ the copy of U_1 in G'_n is reachable at v_i but U_2 is not reachable at v_i. Hence, each subset is reachable at a different set of Q-vertices and we have to use the extended version of Algorithm 5.4 (cf. Section 6.1). Then all sets in U_v are detected equivalent. More precisely, the set $S/_{\approx_S}$ has five elements. For the quotient game automaton we have to use the refined definition of $s_{\min}$ from page 85 to guarantee that the number of reachable copies in G''_n is not larger than the number of equivalence classes of $\approx_S$.

We do not want to give a formal proof here but enumerate the different classes. The following table gives a review of the five equivalence classes where the sets from U_v are all in class three.

Class	Description		Representative
	Winning Plays	Losing Plays	
1	y^ω	None	Q
2	Reach y	otherwise	$\{x\}$
3	Stay in U_v or reach both x and y	otherwise	$\{v, u_1\}$
4	Reach both x and y	otherwise	$\{v, u_1, v_1, u_2\}$
5	None	y^ω	$\{y\}$

Table 7.1: *Equivalence Classes of* $\approx_S$

For example, the memory contents from classes three and four are non-equiavlent because in the game automaton $\mathcal{A}_{\Gamma'_n}$ the input $(vu_1)^\omega$ is accepted from $(\{v, u_1\}, u_1)$ where it is rejected from $(\{v, u_1, v_1, u_2\}, u_1)$. This is because in the second case the set $\{v, u_1, \ldots, u_n\}$ has been left. In memory content $\{x\}$ from class two the set $\{v, u_1, \ldots, u_n\}$ has been left as well but nevertheless $\{x\}$ is non-equivalent to $\{v, u_1, v_1, u_2\}$. This is because from $\{x\}$ Player 0 can move directly towards vertex y to win where from $\{v, u_1, v_1, u_2\}$ the play still has to get by vertex x. This is also the reason why Player 0 cannot win from v_1 because Player 1 decides whether vertex x is visited or not.

We leave it up to the reader to find distinctions for the other pairs of classes and to understand which memory contents belong to which class. This might be a bit sophisticated because normally from two memory contents different sets of plays are possible at all. For $n \geq 4$ consider the memory contents $\{u_1, v_1\}$ and $\{u_2, v_2\}$. They are $\approx_S$-equivalent but reachable at disjoint sets of Q-vertices. In the explanation above for classes three and four we avoided this by choosing appropriate representatives where for classes two and four we argued on a more abstract level. Since in this example we also consider non-reachable vertices in G'_n the analysis gets easier by taking ω-words as inputs for the game automaton $\mathcal{A}_{\Gamma'_n}$ but not as plays for the game Γ'_n. □

7.2.2 Where Game Automata are too weak

With the result from Lemma 7.2 we have shown that there exist example games where our new algorithm yields substantially better results than the minimisation algorithm for strategy automata. For the sake of completeness in this section we present an example where the situation is converse. To relate the results correctly it is important to note that both algorithms have different starting points. In the minimisation algorithm for strategy automata we are given a strategy and try to find an automaton implementing it with as little memory as possible whereas in our new algorithm we reduce the memory component of a game graph and after that compute the strategy. Even if both algorithms have the same objective this difference has to be considered.

The example for the following lemma is much easier than the one from Section 7.2.1 and for this reason we can also give a shorter proof. The game graph $G_n = (Q_n, E_n)$ can be seen in Figure 7.4 and is used with the Staiger-Wagner winning condition

$$\mathcal{F} = \{R \mid w_1 \in R, u_i \in R \iff u'_i \in R \text{ and } v_i \in R \iff v'_i \in R, i = 1, \dots, n\} \cup \{R \mid w_1 \in R, w'_{i1}, w'_{i2} \in R, i = 1, \dots, n\}.$$

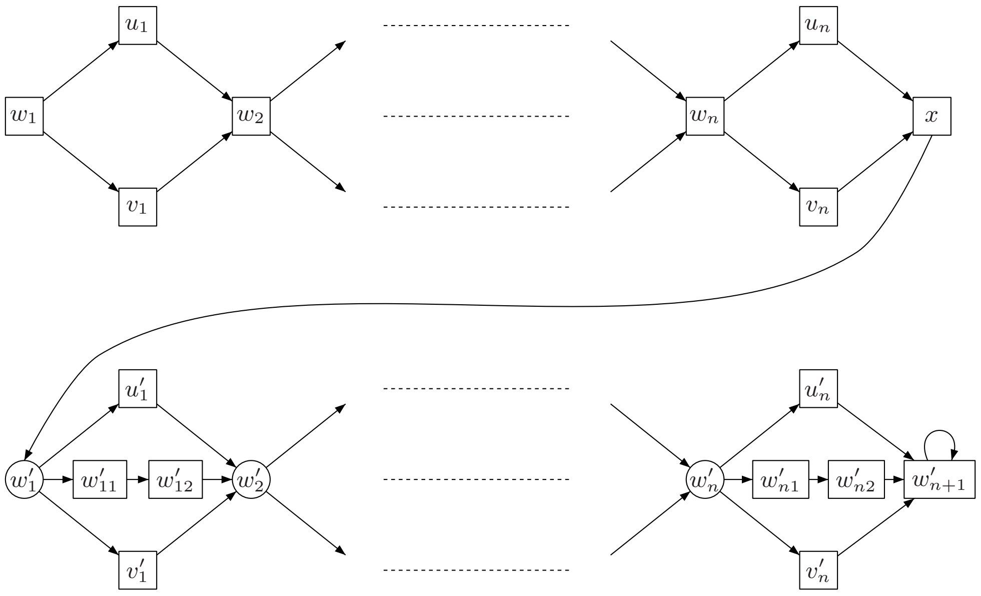


Figure 7.4: *Staiger-Wagner game graph G_n*

Lemma 7.3. *Let $\Gamma_n = (G_n, \varphi_n)$ be the Staiger-Wagner game from Figure 7.4 used with the above winning condition. Let Γ'_n be the corresponding weak parity game returned by the game reduction algorithm from Section 2.2.5. Then Player 0 wins from vertex w_1 and the following hold:*

1. *The attractor algorithm from Section 2.2.2 yields a positional winning strategy f' for Player 0 in Γ'_n from $(\emptyset, w_1)$ and this strategy f' yields a positional winning strategy f for Player 0 in Γ_n from w_1.*

2. *The reduced game graph G''_n computed by Algorithm 5.4 consists of exponentially many copies of Q_n.*

Proof. Let us first analyse the winning condition. We have two kinds of sets where w_1 is contained in each set. Hence, Player 0 can only have a winning strategy from w_1 but from no other vertex. The first kind of sets covers those plays where the developing between vertices w_1 and x is mimicked between vertices w'_1 and w'_{n+1}. At each w'_i he has to choose from two edges and has to move upwards if and only if Player 1 moved upwards at w_i. We call this the "mimic" winning strategy. The second kind of set can be reached by simply moving from each w'_i to w'_{i1} independently of what happened up to vertex x. This is a positional winning strategy and we call it the "autonomous" strategy. Note that Player 0's decision at vertex w'_1 cannot be revised later. Switching from the mimic strategy to the autonomous strategy (or vice versa) means that Player 0 loses.

To prove item 1. of the lemma it is important to understand which of the two described winning strategies is returned by the algorithm from page 19. It is the autonomous strategy. In this game we have $8 \cdot n + 2$ vertices and accordingly the highest possible colour is $2 \cdot (8 \cdot n + 2)$. Though, a vertex of this colour cannot be reached. Starting from w_1 any play visits $2 \cdot n + 1$ vertices up to (and including) vertex x and from there at most $3 \cdot n + 1$ further vertices can be visited, namely by playing the autonomous strategy. We thereby build up a play ρ with $Occ(\rho) \in \mathcal{F}$ and reach the even colour $k' := 2 \cdot (2 \cdot n + 1 + 3 \cdot n + 1) = 10 \cdot n + 4$. Let W_x denote the set of all sets of vertices that can be visited up to (and excluding) vertex w'_1 when starting at vertex w_1. Then any vertex (W, w'_1) with $W \in W_x$ in G'_n is contained in $A_{k'}$ (cf. page 19). To reach $A_{k'}$ in Γ'_n Player 0 has to use the autonomous strategy in Γ_n. The actual reason for this is that playing the autonomous strategy between vertices w'_i and w'_{i+1} $(i = 1, \dots, n)$ two vertices are visited but playing the mimic strategy only one vertex is visited.

For item 2. consider the vertices $(W_1, w'_1), (W_2, w'_1)$ in G'_n with $W_1 \neq W_2 \in W_x$. W.l.o.g. there exists $i \in \{1, \dots, n\}$ such that $u_i \in W_1 \setminus W_2$. Then there exists an input ρ' with u'_i appearing in it such that in the game automaton ρ' is accepted from (W_1, w'_1) but rejected from (W_2, w'_1). Hence, (W_1, w'_1) and (W_2, w'_1) are not $\approx$-equivalent which by Definition 3.12 yields $W_1 \not\approx_S W_2$. By this argument, any of the 2^n sets from W_x is non-equivalent to all others. Thus, $S/_{\approx_S}$ has at least 2^n elements and the reduced game graph G''_n computed by Algorithm 5.4 consists of at least 2^n copies of Q_n. □

Chapter 8

Conclusion

8.1 Summary

In this thesis we have dealt with the problem of reducing the memory that is necessary for implementing strategies in infinite games. To approach this problem we have presented two algorithms. The first one is a minimisation algorithm for deterministic finite automata with output (cf. Section 3.1). It gets as input an automaton $\mathcal{A}_f$ implementing an output function f where in our case parts of f specify a strategy for an infinite game. It returns the automaton $\mathcal{A}_{f,red}$ which is the unique[1] minimal automaton equivalent to $\mathcal{A}_f$. For automata with output equivalence means to compute the same output function. $\mathcal{A}_{f,red}$ can be computed very efficiently, namely in time $O(n \cdot \log(n))$ (cf. [Hop71]). The problem with this approach is that it does not take into account the game Γ for which it implements a strategy. In other words, no matter how complicated f or the implemented strategy is the automaton $\mathcal{A}_{f,red}$ will still implement the same function f and the same strategy.

To overcome this drawback we have presented a new algorithm based on the concept of game reduction (cf. Section 2.2.3). A game reduction is used to solve a game for which positional winning strategies do not suffice. It is an algorithm that transforms a given game $\Gamma = (G, \varphi)$ into a new game $\Gamma' = (G', \varphi')$ such that from a solution to Γ' one can compute a solution to Γ. If this is the case then we say that Γ is reducible to Γ'. The idea is to introduce a memory component S which contains all relevant information about the history of a play. This information is updated appropriately as the play proceeds and is used by the players for making the right moves. For each memory content in S one copy of the set of vertices Q of the given game graph is created and the cartesian product $S \times Q$ is then taken as set of vertices for the new game graph. The size of S often is exponentially large in the size of Q. In return for this blow-up the winning condition for the new game is much easier than the given one in the sense that the new game can be solved by positional winning strategies. From a solution to Γ' one can easily compute finite automata with output that implement winning strategies for both players in Γ. Since these

[1]up to isomorphism

automata have as many states as the set S has elements we aim for reducing this set before we compute them. Accordingly, the idea of our algorithm is to compute from Γ' a new game $\Gamma'' = (G'', \varphi'')$ such that Γ'' has the same structural properties as Γ'. This means Γ shall be reducible to Γ'' and at the same time G'' is to consist of less copies of Q than G' consists of. This means, if we have $G'' = (S' \times Q, E'')$ then we want $|S'| < |S|$. To achieve this, as a first step we transform the game Γ' into a deterministic ω-automaton $\mathcal{A}_{\Gamma'}$, called game automaton. This automaton accepts exactly the winning plays of Player 0 in Γ. On the state set of this game automaton we compute an equivalence relation $\approx$ which is then used to compute an equivalence relation $\approx_S$ on S. For two memory contents s_1, s_2 to be $\approx_S$-equivalent we require for all pairs of states $((s_1, q), (s_2, q))$ to be $\approx$-equivalent (cf. Definition 3.12). For the relation $\approx_S$ we compute the corresponding quotient game automaton $\mathcal{A}_{\Gamma'}/_{\approx_S}$. In the last step we apply the inverse construction of a game automaton to obtain from $\mathcal{A}_{\Gamma'}/_{\approx_S}$ the automaton game Γ''. For the complete algorithm see page 37.

In Theorem 3.14 we have shown that the above procedure of reducing a game graph G' via a transformation to ω-automata preserves the properties of a game reduction. Hence, we have achieved what we wanted. The only requirement for our algorithm to work correctly is that the relation $\approx$ satisfies some simple technical properties which we call compatibility. We have presented two examples of acceptance conditions for which such an equivalence relation is computable efficiently. The first one is called delayed simulation (cf. [EWS05]) and was used here for reducing Büchi game automata. Thereby, we were able to reduce the memory for Request-Response games because they are reducible to Büchi games. A second example of ω-automata that can be reduced efficiently are DWA, namely with the algorithm from [Löd01]. Since Staiger-Wagner games are reducible to weak parity games and the class of weak parity game automata is equal to the class of DWA we are also able to reduce the memory for Staiger-Wagner games. Detailed descriptions of our algorithm applied to both examples can be looked up in Chapter 5.

We have implemented both our new algorithm and the minimisation algorithm for strategy automata (cf. Chapter 6). These algorithms have been integrated into the tool GASt which among other things provides game reductions and solution algorithms for those types of games dealt with in this thesis (cf. [Wal03]). We have done a lot of testing with a variety of examples to figure out the running times of our algorithms. The problem there is that a game reduction often yields a game graph of exponential size. Now and then this has caused out of memory errors. For the extended version of our algorithm (cf. Section 6.1) we have often obtained game graphs where more than 90 % of the created vertices were non-reachable. Up to a size of 15 vertices in the given game graph (or 15,000 to 20,000 vertices for the game reduction) the algorithms from Chapter 5 have tolerable running times. Nevertheless, there are still several open issues that are to be dealt with to improve the implementation of our algorithms. This is especially true for the data structures used in the minimisation algorithm for strategy automata.

In Chapter 7 we have proven that our new algorithm indeed has a right to exist. To do so we have given a family of Staiger-Wagner games for which Algorithm 5.4

reduces the set of reachable memory contents from exponential to constant size (cf. Lemma 7.2). However, the minimisation algorithm for strategy automata yields a strategy automaton of exponential size. Of course, this is mainly because the algorithm for solving weak parity games computes a very complicated winning strategy, namely one of exponential size. We have also seen an example where reduction algorithms for ω-automata are too weak to detect the simplicity of a given game. There we obtain a positional winning strategy for Player 0 but our algorithm yields a memory of exponential size. This is due to the fact that game automata do not take into consideration the behaviour of the two players. In the following Section we give some ideas how our algorithm could be improved.

8.2 Future Prospects

To bring this thesis to a close we want to discuss some suggestions how our algorithm can be improved and which other considerations might play a role for memory reduction in the future.

Manipulating the game reduction. The motivation for our algorithm was based on the observation that the size of the memory provided by a game reduction algorithm often conspicuously exceeds the size of an optimal winning strategy, i.e. of the automaton implementing it. To see this we used the Staiger-Wagner game from Figure 3.5. We argued that if two memory contents both contain vertex 1 then they can be considered equivalent. From that we drew the conclusion that there exists a strategy automaton of size two which implements a winning strategy for Player 0 from his winning region (cf. Figure 3.6). But, if we apply our algorithm to this example then we do not get any equivalent memory contents with vertex 1. If we only consider reachable memory contents then we get for the normal version of Algorithm 5.4 the equivalences $\{0,2,3\} \approx_S \{2,3\} \approx_S \{3\}$ and for the extended version we only get $\{0,2,3\} \approx_S \{2,3\}$. We want to present an idea that allows it to improve this result in the sense that all memory contents with vertex 1 are indeed detected equivalent. It is based on the observation that in the game reduction algorithm from Staiger-Wagner games to weak parity games the colour of any non-reachable vertex can be chosen arbitrarily without violating the definition of game reduction. By a non-reachable vertex we mean a node (s,q) of the game graph G' of the reduced game Γ' such that there is no play ρ through (s,q). Any play ρ is won or lost by Player 0 regardless of the colour of (s,q). This means we get the same set of winning plays for Player 0 in Γ' and, accordingly, the equivalence from item 3. in Definition 2.11 still holds.

Note that non-reachable states are considered only in the extended version of our algorithm and, hence, the following is relevant only for this version. If we apply the game reduction algorithm for Staiger-Wagner games (cf. Section 2.2.5) to the example from Figure 3.5 then of the eight memory contents with vertex 1 four are reachable and four are non-reachable. The reachable ones can be seen in the upper part of Figure 8.1 and the non-reachable ones in the lower part.

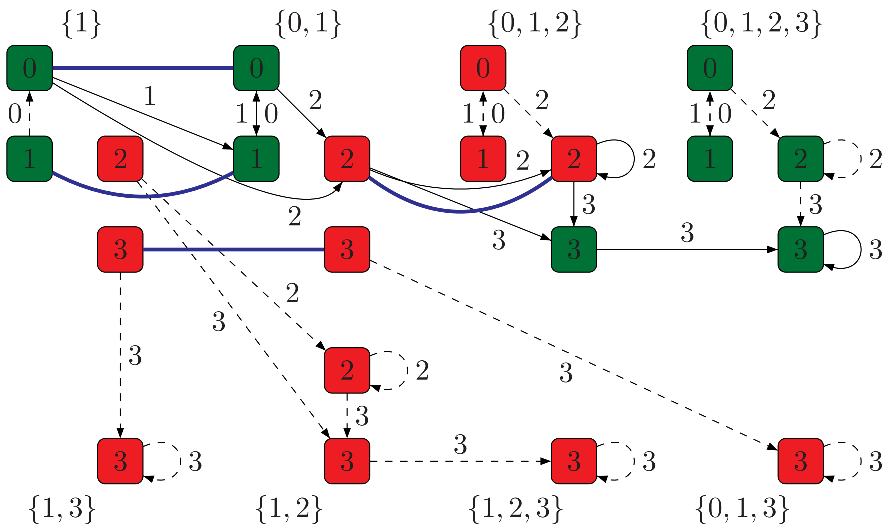


Figure 8.1: *Memory contents with vertex* 1

In this figure we see the DWA which is equivalent to the resulting weak parity game automaton where we have removed most of the states that are irrelevant for the following remarks. We have final (green) and non-final (red) states (cf. Lemma 4.30) and the usual Büchi acceptance condition. In the above figure a state is non-reachable if and only if its outgoing transitions are dashed. The DWA is already given in normal form. Hence, we can apply a DFA minimisation algorithm to obtain an equivalent minimal DWA. The states connected by blue lines are $\approx$-equivalent. Since $(\{1\}, 2)$ and $(\{0, 1\}, 2)$ are not $\approx$-equivalent $\{1\}$ and $\{0, 1\}$ are not $\approx_S$-equivalent. Both $(\{1\}, 2)$ and $(\{0, 1\}, 2)$ are non-accepting but reading input $w = 3$ we reach the non-equivalent pair $((\{1, 2\}, 3), (\{0, 1, 2\}, 3))$. Since $(\{1, 2\}, 3)$ is non-reachable we can change its colour to whatever value we want in order to make it a final state. Since $\{(\{1, 2\}, 3)\}$ is a transient SCC its colour in the normalised DWA only depends on the colour of its successor SCC, i.e. $\{(\{1, 2, 3\}, 3)\}$ which is also non-reachable. Altogether, to make $(\{1\}, 2)$ and $(\{0, 1\}, 2)$ $\approx$-equivalent we change the colour of $(\{1, 2, 3\}, 3)$ to that of $(\{0, 1, 2, 3\}, 3)$. Then after the normalisation we get $(\{1\}, 2) \approx (\{0, 1\}, 2)$ which means $\{1\} \approx_S \{0, 1\}$. The corresponding quotient DWA can be seen in Figure 8.2 where we choose $\{1, 3\} \prec_S \{0, 1, 3\}$ and $\{1, 2\} \prec_S \{0, 1, 2\}$ (cf. Definition 3.12).

This idea can be applied analogously to various other states but we do not want to give all details here. The reader may verify that it is possible to change the colours of the other non-reachable SCCs without successors in the figure above such that after the normalisation all memory contents with vertex 1 become $\approx_S$-equivalent. Then, all vertices with Q-component equal to 2 become non-accepting and all other vertices become accepting. This is consistent with the observation that staying in

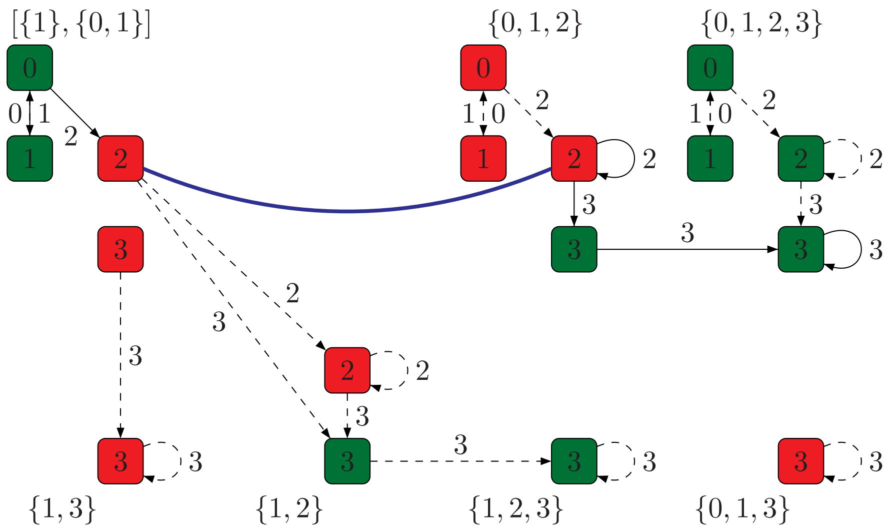


Figure 8.2: *Quotient DWA for modified game reduction*

vertex 2 is not a winning strategy for Player 0 if vertex 1 was seen before.

Note that in general it takes exponential time to compute a colouring which allows for an optimal utilisation of this technique and, thus, is not very useful in practice. The major reason why we give it here is to show that the idea of using state equivalence in an automaton for memory reduction can be even more advanced. The above idea is an extension of a game reduction algorithm and independent of our algorithm.

Distinction between Player 0 and Player 1 vertices. A further improvement of the approach to memory reduction by means of state equivalence in an ω-automaton is to use the partition of the set Q into Player 0 and Player 1 vertices. For both Büchi game automata and DWA we use the minimisation algorithm for standard DFA to compute the equivalence relations $\approx$. There, we use the following condition for equivalence:

$$(s_1, q) \approx (s_2, q) :\iff \forall w \in \Sigma^*(\delta((s_1, q), w) \in F \iff \delta((s_2, q), w) \in F)$$

This means each input word w is either accepted from both states or rejected from both states. The question is whether this condition can be lessened. If in the reduced game a vertex belongs to Player 0 and he has a positional winning strategy from this vertex then he chooses the corresponding transition disregarding the other possibilities. For the game automaton this means that two vertices $(s_1, q), (s_2, q)$ can be considered equivalent if in the game there exist positional winning strategies from these vertices such that the sequences of visited states projected onto the Q-component coincide. To get $s_1 \approx_S s_2$ this has to hold for all vertices in the two considered copies that have the same Q-component. The important thing here is that for some $q \in Q$ the universal quantifier in the condition above can be changed into an existential quantifier. The question is whether (and how) the pairs of states for which the condition for equivalence can be lessened are computable efficiently.

Avoiding the state space explosion. An important consideration for the theory on verification is the state space explosion problem (cf. [TPC02]). In most of the synthesis problems for infinite games the algorithms to solve these problems use exponential memory or have exponential computation costs or both. In this thesis we have dealt with the first of these two problems. An idea to further reduce the memory could be not to consider the whole game reduction but only some parts of it. We do not only mean to leave out the non-reachable parts but also e.g. those vertices where it is already clear that Player 0 loses. It would be interesting to see if there are other observations that allow for simplifications of a game reduction and how they can be realised efficiently in an algorithm.

Another important aspect is the use of symbolic algorithms. There, we do not represent the state space explicitly but on the level of binary decision diagrams (BDDs, cf. [Bry86, Wal03]). Since there exist efficient algorithms for manipulating BDDs we believe that their use could scale down the computation costs of a variety of algorithms.

Appendix A

Computation Times

n	G_n		G'_n			G''_n			Minimisation
	Q	E	S	Q'	E'	S'	Q''	E''	of strategy aut.
1	5	7	15	25	31	9	18	24	2
2	6	10	27	50	72	13	30	48	6
3	7	13	47	97	157	13	34	61	8
4	8	16	83	192	340	13	38	74	10
5	9	19	151	391	741	13	42	87	11
6	10	22	283	814	1624	13	46	100	14
7	11	25	543	1717	3565	13	50	113	16
8	12	28	1059	3644	7812	13	54	126	
9	13	31	2087	7747	17053	13	58	139	
10	14	34	4139	16458	37048				

Table A.1: *Example from Remark 7.1, normal version and minimisation algorithm for strategy automata*

n	Computation times of Algorithm 5.4						
	1,2	3		4,5		6	Total
	G'_n	SCC	Max. Col.	$\approx_S$	$\mathcal{A}/{\approx_S}$	Solve Γ''_n	
1	0 ms	94 ms	0 ms	16 ms	16 ms	15 ms	141 ms
2	0 ms	0 ms	0 ms	47 ms	0 ms	15 ms	62 ms
3	0 ms	0 ms	0 ms	63 ms	16 ms	15 ms	94 ms
4	31 ms	16 ms	0 ms	109 ms	78 ms	62 ms	296 ms
5	31 ms	156 ms	16 ms	375 ms	125 ms	500 ms	1.2 s
6	94 ms	281 ms	94 ms	1.6 s	391 ms	3.8 s	6.2 s
7	407 ms	1.3 s	187 ms	7.3 s	1.5 s	21 s	32 s
8	1.8 s	5.5 s	812 ms	38 s	7 s	72 s	126 s
9	8.6 s	33 s	3.9 s	294 s	42 s	266 s	646 s
10	54 s	Out of memory					

Table A.2: *Example from Remark 7.1, normal version*

n	G_n		G'_n			G''_n		
	Q	E	S	Q'	E'	S'	Q''	E''
1	5	7	15	108	149	7	13	16
2	6	10	27	250	408	17	37	53
3	7	13	47	524	959	17	35	55
4	8	16	83	1062	2114	17	38	65
5	9	19	151	2152	4557	17	42	77
6	10	22	283	4418	9788	17	59	117
7	11	25	543	9204	21083	17	55	111
8	12	28	1059	19390	45558	17	59	123
9	13	31		41120	98585			
10	14	34						

Table A.3: *Example from Remark 7.1, extended version*

n	Computation times of Algorithm 5.4						
	1,2	3		4,5		6	Total
	G'_n	SCC	Max. Col.	$\approx_S$	$\mathcal{A}/\approx_S$	Solve Γ''_n	
1	31 ms	16 ms	0 ms	63 ms	16 ms	47 ms	173 ms
2	16 ms	47 ms	0 ms	296 ms	63 ms	109 ms	531 ms
3	266 ms	172 ms	16 ms	1.2 s	203 ms	984 ms	2.9 s
4	203 ms	657 ms	93 ms	4.5 s	579 ms	7.3 s	13.3 s
5	813 ms	2.3 s	281 ms	17 s	2.3 s	36 s	60 s
6	3.1 s	11.1 s	1.3 s	82 s	11.5 s	116 s	225 s
7	17.7 s	64 s	6 s	535 s	71 s	558 s	1,253 s
8	119.8 s	393 s	35 s	3,644 s	395 s	2,384 s	6,970 s
9	571 s						
10							

Table A.4: *Example from Remark 7.1, extended version*

Appendix B

Hardware and Software

Hardware

- Mainboard ASUS A7-V266-E
- AMD Athlon XP 1800+ 1.53 GHz
- 256 MB RAM

Software

- Operating System
  Microsoft Windows 2000 (with Service Pack 4)
  http://www.microsoft.com

- TEX Implementation
  MiKTEX 2.4 for Windows
  http://www.miktex.org

- Bibliography Administration
  BIBTEX 0.99c
  http://www.bibtex.org

- Text Editor
  TEXnicCenter 1 Beta 7.01
  http://www.texniccenter.org

- Diagram Editor
  jfig 3.06 - The Java Diagram Editor
  http://tams-www.informatik.uni-hamburg.de/applets/jfig/

- Integrated Development Environment
  Eclipse SDK 3.2.0
  http://www.eclipse.org

- Compiler
  Java 2 Platform Standard Edition Development Kit 5.0
  http://java.sun.com
- Unit Testing Framework
  JUnit
  http://www.junit.org
- Version Control System
  Concurrent Versions System (CVS)
  http://www.nongnu.org/cvs

Appendix C

Contents of the CD

Composition

- This document in .ps and .pdf file format
- This document with coloured links in .pdf file format
  - Red links: Sectioning
  - Green links: Bibliography
  - Blue links: World Wide Web

Implementation

- Source code of GASt (current version)
- Test cases for the following games:
  - Figure 3.5
  - Example 5.1
  - Example 5.3
  - Remark 7.1
  - Lemma 7.2
  - Lemma 7.3

Appendix D

Used Symbols

Γ	infinite game, page 11
G	game graph, page 11
Q	set of vertices, page 11
Q_0	Player 0 vertices, page 11
E	edge relation, page 11
ρ	play in an infinite game (or run of an ω-automaton), page 11
$\rho(i)$	the ith vertex (or state) in ρ, page 12
$\Psi(\Gamma)$	set of possible plays in Γ, page 11
$Attr_0$	attractor for Player 0, page 14
$Attr_0^+$	proper attractor for Player 0, page 16
$Recur_0$	recurrence region for Player 0, page 16
φ	winning condition, page 11
F	set of final vertices (or another acceptance component), page 12
$\mathcal{F}$	family of sets of vertices, page 12
$Occ(\rho)$	set of vertices seen at least once in ρ, page 11
$Inf(\rho)$	set of vertices seen infinitely often in ρ, page 11
Win_0	winning plays of Player 0, page 11
W_1	winning region of Player 1, page 13
$L_0(\Gamma)$	language of winning plays of Player 0 in Γ, page 38
S	set of memory contents, page 14

s_0	initial memory content, page 14
$Reach(R)$	the set of states reachable from R, page 84
$NR(s)$	non-reachable states with memory s, page 84
$\mathcal{A}_{\Gamma'}$	game automaton for Γ', page 39
q_0	initial state, page 38
q_{sink}	sink state, page 38
σ	memory update function, page 14
$\approx_S$	equivalence relation on S, page 38
$[s]$	equivalence class of s, page 31
$\mathcal{A}/_{\approx_S}$	quotient automaton of $\mathcal{A}$ with respect to $\approx_S$, page 38
$\prec_S$	total order on S, page 43
Σ	an alphabet, page 48
Σ^*	the set of finite words over Σ, page 48
ϵ	the empty word, page 14
$\alpha[0 \ldots i]$	first $i+1$ letters of α, page 48
$L_\omega(\mathcal{A})$	the ω-language recognised by the DWA $\mathcal{A}$, page 59
$lim(U)$	limit language of the regular language U, page 59
$\mathcal{A}_q$	automaton $\mathcal{A}$ with initial state q, page 48
$G^{bi}_{\mathcal{A}}$	bisimulation game on the automaton graph of $\mathcal{A}$, page 53
$\simeq$	mutual simulation relation, page 50
$index(\sim)$	number of equivalence classes of equivalence relation $\sim$, page 32
$cl(\mathcal{A})$	closure of automaton $\mathcal{A}$, page 52
$\dot\cup$	disjoint union, page 11
$\mathbb{N}$	the set of natural numbers, page 11
2^Q	the power set of Q, page 26
$O(f)$	the complexity class of the function f, page 15
$\exists^\omega$	existential quantifier over the set of all infinite sets, page 56

Bibliography

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969. 5

[Bod05] Eric Bodden. J-LO, A tool for runtime-checking temporal assertions. Diplomarbeit, RWTH Aachen, 2005.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. 110

[Büc62] J. Richard Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, pages 1–12. Stanford University Press, 1962. 3

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981. 3

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, Cambridge, MA, USA, September 2001. 31

[EJ91] E. Allen Emerson and Charanjit S. Jutla. Tree Automata, Mu-Calculus and Determinacy (Extended Abstract). In *32nd Annual Symposium on Foundations of Computer Science*, pages 368–377, San Juan, Puerto Rico, 1–4 October 1991. IEEE. 5

[EWS05] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. *SIAM Journal on Computing*, 34(5):1159–1175, 2005. 6, 8, 29, 47, 49, 52, 81, 106

[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. 4

[Hop71] John E. Hopcroft. An $n \log n$-Algorithm for Minimizing States in a Finite Automaton. Technical report, Stanford, CA, USA, 1971. 30, 64, 87, 89, 105

[Hro01] Juraj Hromkovič. *Algorithmische Konzepte der Informatik: Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kryptographie — Eine Einführung*. Leitfäden der Informatik. B.G. Teubner, Stuttgart-Leipzig-Wiesbaden, 2001. 36

[Hüt03] Patrick Hütten. Automatische Synthese von Controllern für Request-Response-Spezifikationen. Diplomarbeit, RWTH Aachen, 2003. 6, 12, 25

[Löd01] Christof Löding. Efficient minimization of deterministic weak ω-automata. *IPL: Information Processing Letters*, 79:105–109, 2001. 7, 29, 40, 47, 106

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1 December 1993. 4

[Mos84] Andrzej 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. 5

[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. 5

[Mul64] David E. Muller. The place of logical design and switching theory in the computer curriculum. *Communications of the ACM*, 7(4):222–225, 1964. 5

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987. 53

[QS81] Jean-Pierre Queille and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981. 3

[Rab69] Michael O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969. 3

[Rab72] Michael O. Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, Boston, MA, USA, 1972. 5

[Sed91] Robert Sedgewick. *Algorithmen*. Addison-Wesley Publishing Company, Reading, MA, 1991. 62, 63, 89

[SW74] Ludwig Staiger and Klaus Wagner. Automatentheoretische und automatenfreie Charakterisierungen topologischer Klassen regulärer Folgenmengen. *Elektronische Informationsverarbeitung und Kybernetik*, 10(7):379–392, 1974. 7

[TB73] Boris A. Trakhtenbrot and Janis M. Barzdin. *Finite Automata, Behavior and Synthesis*. North Holland, Amsterdam, 1973. 3

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, volume 900 of *LNCS*, pages 1–13, Munich, Germany, 2–4 March 1995. Springer. 5

[TL03] Wolfgang Thomas and Christof Löding. Lecture Notes on Automata and Reactive Systems, 2003. http://www-i7.informatik.rwth-aachen.de/. 5, 13, 22, 27, 29, 81, 86

[TPC02] François Taïani, Mario Paludetto, and Thierry Cros. Avoiding State Explosion: A Brief Introduction to Binary Branching Diagrams and Petri Net Unfoldings. Technical report, October 2002. 110

[Wal03] Nico Wallmeier. Symbolische Synthese zustandsbasierter reaktiver Programme. Diplomarbeit, RWTH Aachen, 2003. 85, 86, 106, 110

[WHT03] Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas. Symbolic Synthesis of Finite-State Controllers for Request-Response Specifications. In *Proceedings of the 8th International Conference on the Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 11–22. Springer, 2003. 23, 25, 26, 86

[WIK] Wikipedia. http://de.wikipedia.org/wiki/Hauptseite.

Index