Hans Jürgen Gamauf, BSc

A Faster GR(1) Synthesis Algorithm

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Telematik

eingereicht an der

Technischen Universität Graz

Betreuer

Univ.-Prof. Roderick Bloem, Ph.D.

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie

Graz, Juli 2017

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum

Unterschrift

Abstract

With the invention of computers, the desire for robust and accurate solutions arose, too. There are different approaches or techniques to achieve this goal - like carefully developing and testing the solution upon a specification written in any natural language, or using a formal language which is well defined and doesn't lead to misunderstandings. And with the introduction of formal languages, also the possibility of computerized testing or even synthesis of such systems evolved. But subsequently there are different problems, one is that it is not very easy to write a formal specification at all that is complete and unambiguous. Additionally a great problem is that the algorithms currently known need a lot of time and memory for execution, even for rather simple specifications. One research area is the synthesis of reactive hardware circuits written in the formal language Linear Temporal Logic (LTL). This can be done by transfering this LTL specification into an automaton based game where the system (which has to be built) is playing against an environment. The next step is to find a strategy for the system to win this game (following the rules and goals defined in the specification) regardless what the opponent (the environment) is doing. The last step will be calculating a hardware circuit which follows this strategy. This master thesis addresses the second part, finding a strategy in such a game. We tried to transform the game to a more complex game for which better algorithms are known. But it turned out, that the faster algorithm didn't compensate for the greater complexity. At the end, the total runtime didn't decline but rose.

Keywords: Formal Specifications, Reactive Systems, Games on finite Graphs, Temporal Logic

Kurzfassung

Mit der Erfindung von Computern ist auch ein Wunsch nach Robustheit und Fehlerfreiheit mitentstanden. Es gibt verschiedene Ansätze bzw Technologien um dieses Ziel zu erreichen - wie sorgfältiges Arbeiten (entwickeln und testen) aufgrund einer in natürlicher Sprache verfassten Spezifikation, oder dem Benutzen einer formalen Sprache, welche keine Möglichkeit zu Missverständnissen zulässt. Und mit der Einführung von formalen Sprachen ist auch die Möglichkeit zum automatischen Testen und sogar der Synthese von solchen Systemen möglich geworden. Aber damit gehen einige Probleme einher: Eines wäre, dass es nicht einfach ist, ein System aufgrund einer formalen Sprache zu spezifizieren, welches komplett und eindeutig ist. Ein Weiteres ist, dass die aktuell bekannten Algorithmen sehr ressourcenhungrig sind (betreffend Speicherverbrauch und Rechenleistung), auch für einfachere Systeme. Ein Forschungsgebiet ist die Synthese von reaktiven Hardware-Schaltungen aufgrund einer Spezifikation, welche in der formalen Sprache ‚Lineare Temporale Logik ‘(LTL) spezifiziert ist. Dies kann bewerkstelligt werden, indem man die LTL-Spezifikation in ein Spiel transferiert, welches auf einem endlichen Automaten basiert und in dem ein Spieler, welcher das zu bauende System repräsentiert, gegen seine Umwelt agiert. Der nächste Schritt ist, eine Strategie für dieses System zu finden, um das Spiel zu gewinnen (nach den Regeln und Zielen, welche in der Spezifikation definiert sind), egal was sein Gegner (die Umgebung) tut. Im letzten Schritt wird eine Hardware-Logik-Schaltung aus dieser Strategie berechnet. Diese Master-Arbeit befasst sich mit dem zweiten Teil, dem Finden einer Strategie in solch einem Spiel. Wir versuchten das Spiel in ein komplexeres Spiel zu transformieren, für welches aber bessere Algorithmen bekannt sind. Aber es stellte sich heraus, dass die schnelleren Algorithmen die höhere Komplexität nicht kompensieren konnten und zum Schluss die Gesamtlaufzeit nicht besser, sondern schlechter wurde.

Schlagworte: Formale Spezifikationen, Reaktive Systeme, Spiele auf endlichen Graphen, Zeitliche Logik

Acknowledgements

I would like to thank numerous people who supported me in writing this thesis.

I am very grateful to my advisor Roderick Bloem, who patiently explained to me a lot of things regarding the mathematical background and other topics about this work. He always had the right answers.

Secondly, I want to thank the formal synthesis team which consisted of Karin Greimel, Georg Hofferek and Robert Könighofer, who where always open for a lot of questions that arose during the work.

Then I also want to thank my friends Isabella Lasch and Patrick Schöberl for proofreading this work.

Graz, in July 2017 Hans Jürgen Gamauf

Contents

Chapter 1

Introduction

1.1 Background and Motivation

With or without being aware of, we are surrounded and penetrated by a great variety of electronic systems, controllers, sensors and so on. Don't think of things we are using actively every day like our laptop or mobile phone, just look at your car, engine control, brake system control, navigation system and hundreds more. Or the traffic light controller, the washing machine, air conditioning or cardiac stimulator, even your electrical toothbrush is controlled by some silicon. The list can be expanded arbitrarily and it will grow even faster in the future: think of refrigerator with internet access, home automation, autonomous cars, robotics, ...

Daily life is meanwhile unimaginable without the use of embedded systems. But with the growth of applications and complexity of these systems also the failures are growing because it is very difficult or even impossible to design and manufacture faultless systems.

A bluescreen on your desktop or a phone which is rebooting right in the middle of a call is annoying, but when your ABS brake system stops working or rail traffic systems get out of control, things might get dangerous. Not to think of airplanes or nuclear power plants. A nice example is the first launch of the new developed Ariane 5 rocket in the year 1996. Only 40 seconds after launch, the unmanned rocket (valued about 500 million US dollars) exploded. Afterwards they found the root cause of the failure in the inertial reference system: the problem was the conversion of a 64 bit float number into a 16 bit integer, which caused an overflow and therefore a failure in the navigation system [1].

But how can we reach better systems? Possibilities are:

- design the systems carefully, however, this is intractable due to the growing complexity and therefore is only wishful thinking,

[1] http://www.ima.umn.edu/~arnold/disasters/ariane.html

- right after the design, we can test the system extensively, but from our experience we know that there are always parts which will not be tested (the Ariane 5 rocket was tested very extensively, but they didn't find the problem before the accident),
- or we can try to specify our design intent in a formal language and verify it formaly, which is, when the specification is complete and consistent, a perfect verification without any doubts.

The latter is getting more and more attention in the research area as well as in practical applications. However, it is very difficult and time-consuming to write down a formal specification of systems, in addition they are not errorless at the outset. So when we are dealing with formal specifications to verify given systems, the question arises if it will also be possible and useful to synthesize systems based on this formal specification. And indeed, there are different approaches and solutions for this synthesis.

What are the advantages or benefits of system synthesis, especially hardware synthesis?

There might be

- the possibility to debug a given specification, to look if it is satisfiable and realizable (satisfiability means that there is no contradiction and therefore a system satisfying such a specification exists, wheras realizability is the question whether such a system can be designed, also in respect to some unpredictable environment, as described later), and when we test a synthesized circuit it should behave as expected,
- or just to develop systems without explicitly designing them, but due to the high effort of synthesis this is still not practicable. Altough it can help, just to synthesize smaller parts to test them within the whole systems and replace them step by step with human designed parts.
- We can also use a formal specification as the perfect definition of a design intent for a hardware designer, without any doubts or uncertainties.

1.2 Synthesis

When we think of embedded systems which are working in the background, we usually mean systems which are providing their services for a long time period (how often do you turn off your phone, factories working round the clock, traffic lights have no weekend, ...). We call these things reactive systems because they are reacting at an environment (this can be the user, some sensors which provide information to control something or another client which uses some services provided by these reactive systems) and should provide their services continuously.

We can further classify these reactive systems open systems, as described before, which are reactive systems interacting with some environment, and closed systems, which do not have an environment or don't have to consider any environment, they just have to fullfill their specification. You can further imagine, that this environment isn't a good friend, it doesn't want to help us to fullfill our specification, it's more an antagonist which is unpredictable.

When you look at reactive systems in a formal way, you can think of a 2-player-game, where the system (the controller which should be synthesized) is playing against the above mentioned environment. This game consists of a lot of states, which are the representation of the several operation states (e.g. "green light is on and red light is off", "green light is off and red light is on", ...). These states are connected together through transitions (or edges), which define allowed crossings from one state to the possible successor states (e.g. to go from "green light 1 is on" to an other state "red light is on", we have to pass a state where the yellow light is on for a short time period, so there is no direct transition between the first two states). Playing on this game means that we place a token on some initial state and move it from one state to a next one and so on. A specification usually consists of different rules, some for the definition of the transition, and some to describe some aim, e.g. if a walker wants to cross a street, he should press the button to signal that, and the traffic lights controller (our system which we want to synthesize) should switch on the walker green light eventually in the near future (for which of course it has to switch on the red car light and switch off the green car light). But it should not only do this once in a while, but always whenever the walker button is pressed. We can translate such a rule into our graph by mark some states, e.g. a state where a walker button is pressed we mark with a red color, and the state where the green walker light is on, we mark green. Now we define a winning condition for the system: Whenever a red marked state is visited, try to reach a green marked state. After that, we will solve this game and look for the right combination of transitions through this graph to fulfill the above specification, which results in a receipt, through which the system knows what to do (which lights should be on or off) in which situation (which is defined by the state currently visited by the token). Rewrite this receipt in table form and we get a so called strategy, where, for each state, the successor state is defined and when the controller follows this strategy, it acts as desired.

1.2.1 Problems

As you can imagine, when we want to design bigger and more complex circuits, with a lot of input signals (lines which are going from the environment to the controller) output signals (lines that are going the other way), and more complex specifications. The whole process gets more and more unfeasible, because of different reasons:

- it is difficult to write complete and consistent specifications (as mentioned before),
- and if there is one, the synthesis process consumes a lot of time and has a great memory consumption,
- but the result is far away from being perfect, in the sense of needed gates as well as in time steps needed to react on some environment wishes.

Therefore there is still no notable practical use, of course. However, a lot of reasearch has already been done to address these problems. While formal verification enters more and more practical design processes, formal hardware synthesis still does not.

1.3 Problem addressed through this master thesis

In this master thesis we are concerned about the second problem and tried to improve and accelerate the synthesis process. The whole process consists of several steps, like building up the game graph, solving the game by computation of a viable strategy for the system, and transform this strategy into a mesh of gates to form a circuit.

While early work used the very complicated second order logic with one successor (S1S) [12, 9, 36] to specify the circuit, the Linear Temporal Logic [34] got more and more attention. Pnueli and Rosner [35] solved the so called synthesis problem, where the specification is written in LTL. A solution can be to transform the LTL specification into a nondeterministic Buechi automata (which results into an exponential increase of states in the size of the number of subformulas of the specification), this automata has to be determinized with the Safra construction [38], which also leads to an exponential increase of states, and is difficult to implement. While there are other methods to avoid this construction for the determinization of automata [29, 28, 22, 24, 31], the lower double exponential bound remains.

One possibility to make synthesis more practicable though, is to restrict the whole expressiveness of LTL formulas to a smaller subset. An important step in the evolution of synthesis was made with the work of Piterman et al. [33] which used this possibility. They invented an algorithm to solve so called general reactivity of rank 1 (GR(1)) games, which consists of a set of conditions which has to fulfill the environment, and if it fulfills them infinitely often, the system has to fulfill some set of guarantees, also infinitely often. Despite of the restriction in expressivity, there are several case studies [5, 6] of real world examples that show that this approach is quite useful. Though this work lead to a great speed up of the synthesis process, it is still to slow for industrial usage. Therefore we tried to use another algorithm for

solving this 2 player game instead the one in [33], in the hope to accelerate again synthesis.

1.4 Solution

Among other possibilities, the approach presented in [7] seemed very purposeful. The idea is to transform the game graph through applying a counting construction, similar to the reduction of generalized Büchi conditions to Büchi conditions (as will be explained in the following chapters). This counting construction introduces two counter, one for the assumptions and one for the guarantees. When you go along any path on this new game graph, the current counter value in each state tells you how much assumptions resp. guarantees you have seen so far. And if you consider only those states, where these counter values are equal to the total number of assumptions resp. guarantees, the winning condition conists of only these two sets, and can be rewritten to: If the environment visits at least one state out of the assumption set infinitely often, the system has also to visit at least one state out of its guarantee set infinitely often. Since this condition is exactly the same as a Streett winning condition with one pair, or as a parity winning condition with 3 colors (the assumption set, the guarantee set, all other states), we can use an algorithm for solving such games, and these are well studied in the literature [10, 23, 32, 26, 39, 27, 44, 20].

We tried out the recursive implementation of [39], because the research work of Friedmann [18] showed that despite of the bad upper bound of the computation time ($O(2^n)$, where n is the number of states of the game graph) it can perform often faster than the algorithm of Jurdziński [26] which is currently the one with the lowest upper bound ($O(dm(\frac{n}{d})^{\lceil d/2 \rceil})$, where n is the number of states, m is the number of transitions and d is the maximum priority of the game graph).

The algorithm of [33] has a upper runtime bound of $O(n^2mjk)$, where j is the number of assumptions and k is the number of guarantees.

All in all, it turned out that the new approach implemented by us didn't result in an increase, but in a decrease of the game solving runtime, which will be presented in the chapter 5.

1.5 Structure

The rest of this diploma thesis is structured as follows:

In chapter 2 we present some theoretical background about automata and games, together with an explanation of symbolic programming and the bridge between the theoretical concepts and real hardware circuits.

In chapter 3 we want to introduce the new approach from a theoretical point of view, including the introduction of the counting construction and

the algorithm to solve the actual game.

In chapter 4 we want to bring these theoretical approaches down to our practical environment, giving a detailed description of how to implement them in a symbolic manner, also we discuss some optimization possibilities.

In chapter 5 we want to introduce two case studies, which have been used in previous work to show real world examples of synthesized circuits, to compare the different game solving algorithms in a practical manner.

And last but not least we want to conclude this work and give an outlook in the last chapter 6.

Chapter 2

Preliminaries

2.1 Temporal Logic

2.1.1 Linear Temporal Logic

Linear Temporal Logic (LTL), nowadays very popular, was invented by Pnueli [34] in 1977 and was meant to specify reactive systems.

LTL formulas are constructed from a set of boolean variables together with boolean connectives and the use of the two temporal connectives X (can be read as "next") and U (can be read as "until").

It is defined in the following way [42, 43, 33]:

- Every atomic proposition is a LTL-formula.
- If φ and ψ are LTL formulas, so are $\neg\varphi, \varphi \vee \psi, \varphi \rightarrow \psi$.
- If φ and ψ are LTL formulas, so are $X\varphi, \varphi U \psi$.

LTL is interpreted over traces over a set of atomic propositions. For a trace τ and a point $i \in \mathbb{N}$, the notation $\tau, i \models \varphi$ indicates that the formula φ holds at the point i of the trace τ.

The semantics is defined as follows:

- $\tau, i \models p$ if p holds at $\tau(i)$,
- $\tau, i \models \neg\varphi$ if $\tau, i \not\models \varphi$
- $\tau, i \models \varphi \vee \psi$ if $\tau, i \models \varphi$ or $\tau, i \models \psi$
- $\tau, i \models X\varphi$ if $\tau, i+1 \models \varphi$ and
- $\tau, i \models \varphi U \psi$ if for some $j \geq i$, we have $\tau, j \models \psi$ and for all k, $i \leq k < j$, we have $\tau, k \models \varphi$.

There are two more temporal connectives, F ("finally") and G ("globally"), which will be introduced for better readability of LTL specifications, and are deduced from the above definition through the two rules:

$$F\varphi = \text{true } U \ \varphi$$

and

$$G\varphi = \neg F \neg \varphi$$

2.1.2 Generalized Reactivity of rank 1

Generalized Reactivity of rank 1 (short GR(1)) defines a subset of LTL.

GR(1) formulas where introduced in [33] to describe a game between a system and an environment. We will explain this game in the next sections, and here only the restriction of GR(1) formulas in comparison to full LTL. GR(1) formulas again consist of a set of boolean variables V, which are separated into a subset $\mathcal{X}$ of input variables, which are controlled by the environment, and the other part of V is the set of output variables $\mathcal{Y}$, controlled by the system.

There are three types of formulas, and each type is again dedicated to the environment and the system:

- Initial condition: Formulas consisting only of boolean connectives. The initial condition formula for the system (Θ_s) reasons only over output variables, while the initial condition formula for the environment (Θ_e) only reasons about the input variables.
- Transition relation: These formulas consist only of boolean connectives together with the temporal connective X.
  - Transition relation for the environment (ρ_e): all variables bound by X have to be input variables, all unbound variables can be either input or output variables
  - Transition relation for the system (ρ_s): all variables bound by X can be input or output variables, unbound variables as above
- Fairness condition φ^g: These formulas are of the form $GF\varphi$ where φ is a boolean formula over some or all variables.

2.2 Automata

We want to give the following definitions and theorems in dependence on [20, 42].

2.2.1 Notation

We will use the symbol ω to denote the set of non-negative integers ($\omega = \{0, 1, 2, 3, \dots\}$).

With Σ we mean a finite alphabet, symbols from a given alphabet are denoted by $a, b, c, \dots$.

Σ^* is the set of finite words over Σ, while Σ^ω is the set of infinite words over Σ. With the letters $u, v, w, \dots$ we indicate finite words, the letters $\alpha, \beta, \gamma, \dots$ are for infinite words. We write $\alpha = \alpha_0, \alpha_1, \alpha_2, \dots$ with $\alpha_i \in \Sigma$. A set of ω-words over a alphabet is called an ω-language. For words α and w, the number of occurences of the letter a in α and w is indicated by $|\alpha|_a$ and $|w|_a$.

Given an ω-word $\alpha \in \Sigma^\omega$, let

$$\mathrm{Occ}(\alpha) = \{a \in \Sigma \mid \exists i.\ \alpha(i) = a\}$$

be the set of letters occuring in α, and

$$\mathrm{Inf}(\alpha) = \{a \in \Sigma \mid \forall i \exists j > i.\ \alpha(j) = a\}$$

the set of letters occuring infinitely often in α.

2.2.2 ω-Automaton

Formally, a finite ω-automaton A is a tuple $(Q, \Sigma, \delta \text{ or } \Delta, q_0, Acc)$, where

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\Delta : Q \times \Sigma \to 2^Q$ (if the automaton is nondeterministic) or $\delta : Q \times \Sigma \to Q$ (if the automaton is deterministic) is the transition relation,
- $q_0 \in Q$ is the initial state, and
- Acc is the acceptance condition.

A run ρ of the ω-automaton A on an ω-word $\alpha = a_0, a_1, \dots \in \Sigma^\omega$ is an infinite state sequence $\rho = \rho_0, \rho_1, \dots$ with $\rho_0 = q_0$ and $(\rho_i, a_i, \rho_{i+1}) \in \Delta$ for $i \geq 0$ if A is nondeterministic, or $(\rho_i, a_i, \rho_{i+1}) = \delta$ for $i \geq 0$ if it is deterministic.

2.2.3 Acceptance Conditions

There exist different acceptance conditions, we want to mention the following.

Büchi Acceptance

An ω-automaton A, where the acceptance condition is given by a set $F \subseteq Q$, is called a Büchi automaton, when the acceptance condition is defined by

$$Acc(\rho) \leftrightarrow \mathrm{Inf}(\rho) \cap F \neq 0,$$

where $\mathrm{Inf}(\rho)$ denotes the states occuring infinitely often in a run ρ. That means, the autmaton accepts a word $\alpha \in \Sigma^\omega$ iff some state $\in F$ occurs infinitely often in the run $\rho(\alpha)$.

$L(A) := \{\alpha \in \Sigma^\omega \mid A \text{ accepts } \alpha\}$ is the ω-language recognized by A. An ω-language $L \subseteq \Sigma^\omega$ is Büchi recognizable, if a corresponding Büchi automaton A with $L = L(A)$ exists.

Streett acceptance

An ω-automaton A, where the acceptance condition is given by a set $S = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ with $E_i, F_i \subseteq Q$ and the acceptance condition is defined by

$$Acc(\rho) \leftrightarrow \bigwedge_{i=1}^{k} (\mathrm{Inf}(\rho) \cap F_i \neq 0 \rightarrow \mathrm{Inf}(\rho) \cap E_i \neq 0)$$

is called a Streett automaton.

This automaton accepts a word α iff there exists a run ρ on α where the following holds: if in any pair (E, F) a state of F occurs infinitely often, there must also a state of E occur infinitely often.

Parity Acceptance

We introduce a so called priority function $c : Q \rightarrow \{1, \ldots, k\}$ (where $k \in \omega$), which assigns to each state of A a priority. An ω-automaton A, where the acceptance condition is given by such a priority function c, is called a parity automaton, when the acceptance condition is defined by

$$Acc(\rho) \leftrightarrow \max\{c(q) \mid q \in \mathrm{Inf}(\rho)\} \text{ is even.}$$

This automaton accepts a word α iff there exists a run ρ on α, where the highest priority of all states which are occuring infinitely often, is even.

2.2.4 Generalizations

Büchi and parity acceptance conditions can be generalized. A generalized Büchi condition consists of a collection $\mathcal{F} \subseteq 2^Q$ of Büchi conditions. A run ρ is accepting for a generalized Büchi automaton if and only if it accepts each $F \in \mathcal{F}$. A generalized parity condition is a conjunctive or disjunctive

collection Π of some individual priority functions c. A run ρ is accepting if it is accepting according to each member of Π (in the case of a conjunctive generalized parity condition) or to some member of Π (in the other case) [40].

Each generalized Büchi automaton can be transfered into a Büchi automaton. [42] If we have a gerneralized Büchi automaton with the final state sets $F_1, \dots F_k \subseteq Q$, we can construct an equivalent Büchi automaton by attaching to each state a counter. This counter shows the next state set that should be visited. A state (q, i) means we are waiting for a state $\in F_i$. After visiting a state of F_i we increment the counter and are now waiting to visit a state in F_{i+1}. If we have visited a state in F_k, we will increment the counter for a last time and will then reset it to 1. Then we can declare all states, where the counter is equal to $k + 1$ to our final state set of the new Büchi automaton. [42]

2.3 Games

Here we want to introduce games which are played by two players, what is meant by winning regions or winning strategies and how a game can be won by a certain player (winning conditions).

First of all, games consist of an arena and a winning condition.

2.3.1 Arena

An arena (often also called a game graph) is defined by the triple

$$A = (V_0, V_1, E)$$

where

- V_0 is a set of 0-vertices,
- V_1 is a set of 1-vertices (disjoint from V_0), $V = (V_0 \cup V_1)$ and
- the edge relation $E \subseteq V \times V$, which is complete in the following sense: $\forall v \in V \exists v'.(v, v') \in E$ (every vertex has a successor)

2.3.2 Play

The game which we are interested in is played on such an arena in the following way.

At the beginning a token is placed on some initial vertex $v \in V$. if $v \in V_0$, then player 0 has the choice and moves the token alongside an arbitrary edge to a successor-vertex of the current, otherwise, if $v \in V_1$, Player 1 has the choice and does the same.

We can define a play as the sequence $\pi = r_0 r_1 r_2 \dots$ where $(r_i, r_{i+1}) \in E$.

2.3.3 Game

If A is an arena as above, and $W \subseteq V^\omega$, then the pair (A, W) is a game, where W is the winning set of the game.

A play π on a game is won by Player 0, if π is an infinite play and $\pi \in W$. If the play π is not won by Player 0, it is won by Player 1.

When we are talking about games for reactive systems, we usually denote player 0 by the system and player 1 by the environment.

Deterministic ω automata can be used to describe infinite games. Both players move a token along the transitions of the automaton, and player 0 wins the game if the resulting infinite sequence of states is accepted by the automaton. In turn based games the set of states is partitioned into a set belonging to player 0 and a set belonging to player 1, as defined before. Each player moves the token, when it is on one of its states by choosing a letter from Σ, which leads to a successor state through the transition relation δ. In input based games the alphabet Σ is the product $\Sigma_0 \times \Sigma_1$ of the two alphabets belonging to each player. When the token resides on a state q, both player choose a letter from their alphabet and the token is then moved to a state according to $\delta(q, (\sigma_0, \sigma_1))$, where $\sigma_i \in \Sigma_i$. There are different possibilities of the order:

- both players choose their letters at the same time,
- player 0 starts to choose his letter, than player 1 chooses his letter or
- vice versa, player 1 starts and player 0 is next.

There can also be a restriction of the information which each player gets from his opponents choices, he can get all, partial or no information. If we consider games where the two players choose alternatingly their letters and each player knows all about the other players choice, we can easily reduce input based games to turn based games [40]. Note that the transition relation of automata is complete in the sense that for each state and each possible letter of Σ a next state is defined, whereas this doesn't have to be valid for turn based games as is defined in section 2.3.3.

2.3.4 Strategy

A strategy for a player is a recipe that specifies how to move the token along the play. Formally, a strategy for Player $\sigma \in \{0, 1\}$ starting at vertex v_0 is a function $t : V^*V_\sigma \to V_{1-\sigma}$ that assigns to each play prefix $v_0 \dots v_k$, with $v_k \in V_\sigma$, a vertex $r \in V$ with $(v_k, r) \in E$. A play $\pi = r_0 r_1 r_2 \dots$ started in v_0 is played according to t if for every $v_i \in V_\sigma,\ v_{i+1} = t(v_0 \dots v_i)$ holds.

A strategy t is a winning strategy from v_0 for Player σ, if every play played according to t is in W, regardless what the other Player is doing, Player $1-\sigma$ must not have a possibility to win starting from v_0[1].

Usually strategies depend on their history and can be implemented as follows: let M be a set called memory (storing the moves of the play so far, so if the play is infinite it is also the memory), then you can view the strategy as a pair of functions:

- a function which updates the memory $t_M : V \times M \to M$, which takes the history of the play so far and updates its memory with the new state, and
- the next move function $t_N : V \times M \to V$, which delivers the next state to which the token should be moved.

If this memory M is finite (depending only on the winning condition and not on the game arena A), we talk about a finite memory strategy [44]. Such strategies are computable by a finite automaton (such as a Mealy machine)[41].

If we can ommit the memory M, we name this strategy a memoryless strategy (or positional). A memoryless strategy only depends on the current state, formally, $t_N : V \to V$.

It seems that in general the existence of a winning strategy depends on the initial state, where we place the token at the beginning. But it turns out, that it is more convenient, instead of calculating the winning strategy for a fixed initial position, calculate the whole winning region and the whole winning strategy for a player σ [44].

2.3.5 Winning region

The winning region for the Player σ is the set of all states from where the player has a winning strategy. Formally, the winning region

$$R_\sigma = \{v \in V \mid \text{Player } \sigma \text{ has got a winning strategy starting from v}\}$$

2.3.6 Winning Condition

Similar to ω-automata where we have different acceptance conditions, we have different winning conditions in the game theory.

We are not interested in enumerating all the different winning sets for each game, so we are looking for functions which describe them. We want to mention the following:

[1] in distinction to cooperating Players, where their strategy is predictable

Büchi Game

Consider a game arena $A = (V_0, V_1, E)$, and $F \subseteq V$. If the winning condition φ of the Player 0 for the play ρ is

$$\varphi : \rho \in W \leftrightarrow \mathrm{Inf}(\rho) \cap F \neq 0,$$

then this game is a Büchi Game. Player 0 wins the game if he can visit at least one state of F infinitely often. If player 0 doesn't win the game, then player 1 wins it.

Parity Game

Additionally to the game arena A, parity games consist of a parity function $c : V \to \{0, 1, \ldots, d\}$ for some integer d and the winning condition

$$\varphi : \rho \in W \leftrightarrow \max(\mathrm{Inf}(c(\rho))) \text{ is even.}$$

for Player 0, who wins the game, if the highest parity of the set of states which appear infinitely often in a play, is even.

Streett Game

If we enhance the game arena A with the set $S = \{(E_1, F_1), (E_2, F_2), \ldots, (E_k, F_k)\}$ with $E_i, F_i \subseteq V$ and the winning condition

$$\varphi : \rho \in W \leftrightarrow \bigwedge_{i=i}^{k} (\mathrm{Inf}(\rho) \cap E_i \neq 0 \vee \mathrm{Inf}(\rho) \cap F_i = 0)$$

for Player 0, we get a Streett Game, where Player 0 wins the game, if it holds for all pairs (E_i, F_i), that if a state of F_i appears infinitely often, then there must also appear a state of E_i infinitely often.

Rabin Game

Rabin games are the dual of Streett games, everything is the same apart the winning condition, which is the negation of the one of Streett games:

$$\varphi : \rho \in W \leftrightarrow \bigvee_{i=i}^{k} (\mathrm{Inf}(\rho) \cap E_i = 0 \wedge \mathrm{Inf}(\rho) \cap F_i \neq 0)$$

for Player 0, who wins the game, if there is at least one pair (E_i, F_i), where at least one state of F_i is visited infinitely often, but no state of E_i is in the set of the infinitely often visited states.

2.3.7 Game structure

A game structure [33] is a specialized redefinition of a two player game:

A game structure $G : (V, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \varphi)$ consists of the following components:

- A finite set of boolean state variables $V = \{u_1, \ldots, u_n\}$. A state is an interpretation of V, i.e. assigns to each variable u_i a value of $\{0, 1\}$. In the above definition of a game we are talking about vertices which usually are states, but we want to talk about vertices when their labelling can be somehow, and about states if we mean a labelling which is a valuation of boolean state variables. By Σ we denote the set of all states. A state s satisfies a boolean formula φ denoted by $s \models \varphi$, if $s\,[\varphi] = \text{true}$.

- $\mathcal{X} \subseteq V$ is a set of input variables which are controlled by the environment. D_X denotes the possible valuations of variables in $\mathcal{X}$.

- $\mathcal{Y} = V \backslash \mathcal{X}$ is the set of output variables which are controlled by the system, again, D_Y denotes the possible valuations of variables in $\mathcal{Y}$.

- Θ is the initial condition, which is a boolean formula over V. A state is is called initial if it satisfies Θ.

- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is the transition relation of the environment. It is a boolean formula over V relating a state s to a possible next input value $\xi' \in D_X$. So for a given state s ρ_e defines the possible input value which the environment is allowed to choose for the next step.

- $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ is the transition relation of the environment. It is also a boolean formula over V relating a state s and a next input value $\xi' \in D_X$ to a possible next output value $\eta' \in D_Y$. So it enumerates the possible next output variables from which the system can choose from for a given state s.

- φ is the winning condition given by a LTL formula.

For two states s and s', s' is a successor of s in the game structure G if $(s, s') \models \rho_e \wedge \rho_s$.

These games are played in the usual way, a token is placed on some initial state, then the environment begins by choosing a valuation of the input variables, then the system chooses based on this next input value a next output value and hence a new state is entered, and so on. Player 0 wins the game if the infinite sequence of visited states satisfy the winning condition φ.

GR(1) Games

Games with GR(1) properties are defined using the GR(1) temporal logic (as defined above) together with the above definition of game structures. We will slightly extend the above definition of game structures: Again it is a two player game, the system (player 0) against an environment (player 1). For each player the specification consists of a conjunction of the three parts:

- the initial condition Θ,
- the transition relation ρ and
- the fairness condition φ^g.

Each of this parts is itself a conjunction of the initial conditions of a specific player, resp. a conjunction of the different transition relations and also a conjunction of the fairness conditions (each fairness condition defines a set of states which have to be visited infinitely often like a Büchi automaton).

So we get a specification for the environment:

$$\varphi_e = \bigwedge_i \Theta_e^i \wedge \bigwedge_i \rho_e^i \wedge \bigwedge_{i=1}^{m} \varphi_e^{g,i}$$

Similar is the specification for the system:

$$\varphi_s = \bigwedge_i \Theta_s^i \wedge \bigwedge_i \rho_s^i \wedge \bigwedge_{i=1}^{n} \varphi_s^{g,i}$$

The winning condition is then defined on an infinite play as $\varphi : \varphi_e \rightarrow \varphi_s$, which means, if the game starts at some initial state then the system wins if: The system goes along its transition relation ρ_s and the token passes infinitely often at least one token out of each fairness state set $\varphi_s^{g,i}$ during an infinitely play. It wins also if the environment either violates its transition relation ρ_e or it visits all states of some fairness state set $\varphi_e^{g,i}$ only finitely often. Otherwise the environment wins.

2.3.8 Determinacy

If each vertex of a game belongs to either the winning region of player 0 or to the winning region of player 1, the game is determined. Formally, $R_0 \cup R_1 = V, R_0 \cap R_1 = 0$

Büchi games and parity games are determined and both players have a memoryless winning strategy[20]. Streett games and Rabin games are also determined, but only the player 0 of Rabin games and the player 1 of Streett games have a memoryless strategy, whereas in general only a finite memory strategy exists for player 0 of Streett games respectively for the player 1 of Rabin games [17, 21]. We show in section 3.2 that we can reduce a GR(1)

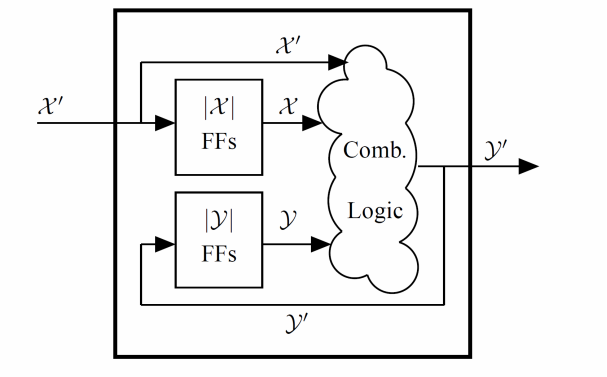

Figure 2.1: Diagram of a generated circuit [6]

game into a parity game, therefore GR(1) games are also determined. If you neglect the additional memory through the Counting Construction both players have a memoryless strategy.

2.4 From a strategy to a circuit

As we have already mentioned, synthesizing circuits is done by defining a 2 player game and solve this (calculate the winning strategy) for one designated player (the system). Since the states are represented by boolean variables, we can map to each boolean variable a hardware signal with which our synthesized circuit is commucicating with its environment. We have $|\mathcal{X}|$ (the number of variables in $\mathcal{X}$) input signals, and we have $|\mathcal{Y}|$ (the number of variables in $\mathcal{Y}$) output signals. In figure 2.1 we have depicted a diagram of such a synthesized circuit. It consists of a flipflop for each signal to store the current state, which is a valuation of all variables (or signals). After the calculation of the winning strategy, there will be calculated a specific output function for each output based on the strategy. These output functions are a mapping of the current state and additionally the next input signals to a new output signal valuation ($f : (\mathcal{X}, \mathcal{Y}, \mathcal{X}') \rightarrow \mathcal{Y}'$). These output functions can simply be encoded in combinatorial logic (just consisting of NOT, AND and OR gates). The initial value of the game structure denote the initial value of the flipflops. The mode of operation is then: Based on the initial state of the flipflops and the first values of the input signals, the first output signal valuation will be calculated. In the next step the flipsflops store this new input and output signals and a new output signal valuation will be calculated upon a new input signal valuation, and so on. A detailed description of how these output functions are calculated can be found in [6].

2.5 Symbolic Algorithm

In the field of verification and synthesis of reactive systems it turned out, that using algorithm, that are not dealing explicitely with vertices or edges rather using some other representations like Binary Decision Diagrams (BDDs) [8] are more useful. The main difference between symbolic and enumerative programming is the way how to handle sets: While in enumerative programming the single elements of sets are stored explicitely (e.g. in arrays or lists), you are dealing in the latter with whole sets at once, not elementwise.

BDDs are a special representation of boolean formulas of some boolean variables (internal they are organzized in a boolean tree graph). On this boolean formulas there are defined the boolean connectives ($\vee, \wedge, \neg$) in the usual way and also the quantification operators $\exists$ and $\forall$ together with some other operators. Through BDDs boolean formulas can be stored that is often substantially more compact than conjuctive or disjunctive normal forms, and the algorithms manipulating them are very efficient. Because the symbolic representation captures some of the regularity in the state space determined by circuits, it is possible to handle systems with an extremely large number of states, much bigger than by handling them in an enumerative way [13, 11, 30].

Since we are not dealing with explicit set elements rather manipulating the whole set, also runtimes of algorithms using BDDs can be significantly lower than by elementwise manipulation.

Also it can be very easy to implement some algorithms (especially such which are dealing with sets) through the usage of BDDs, but on the other hand not all algorithms can be implemented easily symbolically, so a decision for the one or the other can be a great discussion.

A very common function often used in symbolic programming is a fixpoint calculation, which is nothing else then a loop, which is started with some initial set and then some elements to/from this set are added or striked out until nothing more can be added or striked out. But it only works, if the function which is calculating the new state set of each iteration, is monotonic [13].

Chapter 3

Theoretical Approach

3.1 Current Solution - RATSY

RATSY (Requirements Analysis Tool with Synthesis)[1] [3] is an extension of the tool RAT (Requirements Analysis Tool)[2] [1], which provides a graphical interface for the development, analysis and management of hardware specifications. Additionally to the tool RAT, RATSY also provides an automaton editor, which assists the developer with a nice interface to define the specification of the hardware circuit, and the tool Marduk to generate a circuit from the given specification.

3.1.1 Marduk

Marduk is based on the tool Anzu [25], and is mainly a port to Python, using the software libraries NuSMV[3] (implementing the conversion of LTL formulas into Buechi automata) and CUDD[4] (handling the operations on BDDs).

Marduk takes as input a xml-file with the GR(1) specification and produces an output circuit in Blif or Verilog format (if the specification is realizable). The synthesis algorithm is based on the work of Piterman et al. [33].

In the following we will describe the different steps which have to be passed in order to get a hardware circuit from a formal specification.

Specification

Due to the fact that we handle only GR(1) games (see section 2.3.7), the specification (stored in an input file) is closely related to game structures

[1] http://rat.fbk.eu/ratsy
[2] http://rat.fbk.eu
[3] http://nusmv.fbk.eu/
[4] http://vlsi.colorado.edu/ fabio/CUDD/

and consists of the following parts:

- a list of the input signals or variables $\mathcal{X}$ (signals which are controlled through the environment),
- a list of the output variables $\mathcal{Y}$ (signals which are controlled through the system),
- the initial conditions for each player, $\Theta_e(\mathcal{X})$ and $\Theta_s(\mathcal{Y})$,
- the transition relation of the environment $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$,
- the transition relation of the system $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$,
- the environment fairness conditions φ_e^g of the form $G(F(\varphi(\mathcal{X}, \mathcal{Y})))$ (we will denote them also as environment assumptions) and
- the system fairness conditions φ_s^g of the same form as the environment fairness conditions (will be denoted also as system guarantees).

Game

This file is being processed through NuSMV, which itself returns a game which is stored in BDDs corresponding to the above parts of the specification file.

Through the input and output signals we get the state space ($2^{\mathcal{X} \cup \mathcal{Y}}$), which is a permutation of all possible values of all signals.

But this state space is not defined explicitly, but rather implicitly through the both transition relations ρ_e and ρ_s (the two endpoints of each transition are states).

The initial values from the specification file are transformed through NuSMV into two sets of states denoting the possible starting point of the winning strategy. If all signals have a corresponding initial value, only one state would be in the conjunction of these two sets (Θ_e resp. Θ_s), but not all signals have to be specified in the initial condition.

Additionally, the Fairness conditions are being translated to sets of state sets, i.e. every fairness condition represents a set of states from which at least one state has to be visited infinitely often and the resulting set is just a set of these state sets. We want to refer to these sets as $J^1_{1\ldots m}$ or assumption state sets for the environment assumptions, resp. $J^2_{1\ldots n}$ or guarantee state sets for the system guarantees.

Winning Region and Strategy

The calculation of the winning region (R) and the winning strategy (t) is explained in detail in [33]. In short, the winning region is calculated with a symbolic fixpoint-calculation with three nested loops. It returns the winning

regions of the two players (system and environment), regardless of the initial values. Right after this calculation the conjunction of the winning region of the system and the initial state set is calculated. If this conjunction is not empty, the specification is realizable and it denotes the initial values of the Flip-Flops used in the generated circuit (see [6]).

The calculation of the winning strategy for the system uses some intermediate values which are determined through the winning region calculation, resulting in a non-deterministic strategy. That means that there might be more than one possible output signal assignment for each state and each possible input signal.

Output function and code generation

The next and last stage of circuit synthesis uses the winning region and winning strategy only of the system. Its task is to determinize the strategy (for each state and each allowed next input should be only one next output) and pass it to the calculation of the output functions, as is explained in section 2.4. The result is a circuit and can be written into a file in Blif or Verilog format.

3.1.2 Interfaces

The idea is to replace the calculation of the winning region and the winning strategy through another algorithm. Therefore we get the game graph represented through the transition relations for the environment and the system (ρ_e and ρ_s), the initial state sets (Θ_e and Θ_s) and the fairness sets (J^1 and J^2), and we should deliver the winning strategy for the system (can be nondeterministic) so that the output function generation can calculate the resulting circuit. In Figure 3.1 you can see the function chain of Marduk.

3.2 Idea - Counting Construction

In [2] there was published the idea of reducing the GR(1) game into an one pair Streett game or a paritiy game through applying a counting construction. This is similar of reducing a generalized Büchi automaton to a Büchi automaton (see 2.2.4).

The idea behind this construction is to somehow count the visited assumption sets resp. guarantee sets. We have n guarantee state sets ($J^2_{1\ldots n}$), from which at least one state out of these n state sets has to be visited infinitely often. The count itself will be stored for each state, therefore we will expand the state space by adding boolean state variables to the existing state space. Then we increment an guarantee counter only by one to the value i (the count will be initialized with zero) along a transition only if the next state (s') of this transition is an element of the i-th guarantee set

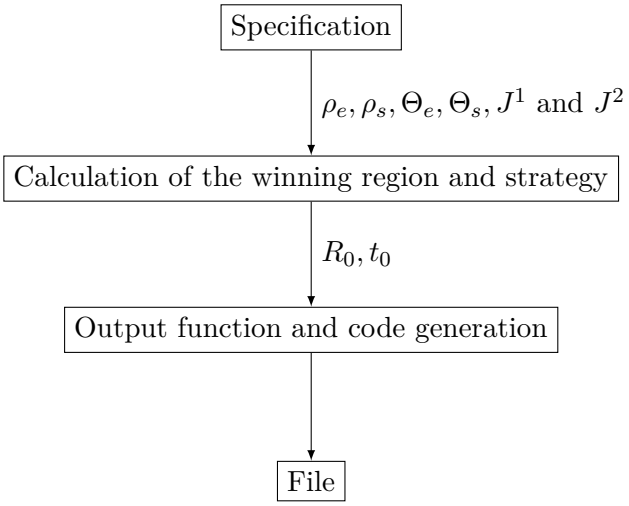

Figure 3.1: Function chain of Marduk

($s' \in J_i^2$). When we start a play with the counter value 0 and visit some states along the transitions, we know that if we reach a state where the count is equal to n we have visited at least one state of each guarantee state set. After this last state (where the counter is equal to n) we have to reset it because we have to deal with infinite plays and so we can guarantee that the token will visit all guarantees again and again because he has to pass the individual guarantee states to increment the counter to finally reach a state where the value is n infinitely often.

We can introduce a count similar for the assumptions, so we know if the environment fulfills its specification and can therefore reduce the winning condition to deal only with states where the two counts are equal to the number of guarantees res. assumptions.

Formally, we have a game $G = (V, E, \varphi)$ with $\varphi = \bigwedge_{i=1}^{m} \varphi_e^{g,i} \rightarrow \bigwedge_{i=1}^{n} \varphi_s^{g,i}$. This is not the exact definition of a GR(1) game but has nearly the same winning condition (we only consider the state sets which have to be visited infinitely often and omit initial conditions or the violation of transition relations). The formulas φ^g are denoting these state sets which have to be visited infinitely often, as defined in section 2.3.7. For this game we want to construct an equivalent one pair Street game $G' = (V', E', \varphi')$ with $\varphi' = \varphi_e'^g \rightarrow \varphi_s'^g$ with the following rules:

- The state space $V' = V \times \{0, 1, \ldots, m\} \times \{0, 1, \ldots, n\}$
- The edge set E' is the union of the following three sets:

a)	$((v,i,j),(v',i',j'))$	if $(v,v') \in E, i' = i+1$ if $v' \in J^1_{i+1}$ otherwise $i' = i$, and $j' = j+1$ if $v' \in J^2_{j+1}$ otherwise $j' = j$.
b)	$((v,i,n),(v',0,0))$	for $0 \leq i \leq m$ and $(v,v') \in E$
c)	$((v,m,j),(v',0,j'))$	if $j \neq n$, $(v,v') \in E$ and $j' = j+1$ if $v' \in J^2_{j+1}$ otherwise $j' = j$

- The Streett pair is $J'^1 = \{(v,m,j) \in V' \mid j \in \{0,\ldots,n\}\}, J'^2 = \{(v,i,n) \in V' \mid i \in \{0,\ldots,m\}\}$, where J'^1 is the assumption state set, denoted through φ'^g_e, and J'^2 is the guarantee state set, denoted through φ'^g_s

This counting construction works as follows: At the beginning, both counters are zero, each counter is incremented if the following state is a member of the set J^1_{i+1} resp. J^2_{j+1}. Since there is no order of the two sets J^1 or J^2, you can build up the counting construction in an arbitrary order as long as every member of J^1 and J^2 is considered (there is even the possibility to try different orders and to look if some leads to simpler transition relations). This is because we have to think in an infinite manner, it doesn't matter if we need 3 or 7 runs through a path to get the guarantee counter to n, because we will visit this path infinitely many times. So, when you visit a state with the assumption counter set to m (the number of the assumptions), then on this way you have seen all assumptions, which means the environment has fulfilled all requirements. Analogously, when you visit a state where the guarantee counter is set to n (the number of the guarantees), then you have visited all guarantees, so the system has fulfilled all requirements. In order to provide the possibility to loop over this construction, we need some resetting, which is defined as reset both counters if we have seen all guarantees ($j = n$), but reset only the assumption counter if we just have seen all assumptions ($i = m$). We could also view the two counter seperately and reset in the case $j = n$ only the guarantee counter, and reset the assumption counter only when $i = m$. We have this two possibilities because if we have seen all guarantees, it doesn't matter if the environment is able to fulfill its specification or not, since the system is winning because the right side of the winning condition implication is fulfilled. Of course, we must not reset the guarantee counter if its count is lower than n although we would reset the assumption counter because it has reached m, because then we would possibly find no winning strategy for the system.

We want to note that applying the counting construction on a game whith just one guarantee and one assumption is useless, because such a game has already the form of an one-pair Streett game (the counting construction would add only additional complexity).

Lemma 3.2.1 ([2]). *There exists a winning strategy for G iff there exists a winning strategy for G'.*

The above definition of the counting construction slightly differs from the definition in [2] in the way the reset is performed. In the construction of [2] the reset is done by inserting a new edge from a state where the reset should be done leading to the same state. But this leads to stuttering because the game can remain at the same state to perform the reset, but don't update other signals. Consider a specification rule $G(i \rightarrow Xo)$, and when by accident at this transition we will have this new reset transition, we would violate this specification rule because we need two steps instead of one tick. But this adaption doesn't hurt the operation mode of the counting construction, so the above lemma is also valid in this case.

To construct a parity game $G'' = (V, E, \varphi)$ where $\varphi : \max(\text{Inf}(c(\rho)))$ is even. additional to a parity function c, which is equivalent to the above GR(1) game G, we can use the above definition of the counting construction (the first two rules), and have to define the parity function c instead of the Streett pair:

$$c : \left\{ \begin{array}{ll} 2 & \text{if } (v, i, n) \text{ for } (0 \leq i \leq m) \\ 1 & \text{if } (v, m, j) \text{ for } (0 \leq j \leq n-1) \\ 0 & \text{if } (v, i, j) \text{ for } (0 \leq j \leq n-1) \text{ and } (0 \leq i \leq m-1) \end{array} \right\}$$

This means, we give each state the parity 2, which has the guarantee counter set to the number of guarantees (if we reach a state of this set, we have seen all guarantees so far), we also want to denote these states as green states. All states, which have the assumption counter set to the number of assumptions, but the guarantee counter is lower than the number of guarantees, get the parity 1 and are denoted by red states. All other states have the parity 0 (grey states). The game graph remains the same. The winning strategy for this parity game is the same as for the above Streett game, because the winning conditions are equivalent. Consider the case when the system wins because it is fulfilling its specification. Then in the Streett game the system visits infinitely often at least one state of J'^2. In the parity game the same is achieved when the highest priority seen infinitely often is even, which would be 2 (=even) in our case. Since the green states are equal to J'^2, this case is the same for the two games. The system can also win, if the highest priority seen infinitely often is 0 (=even), which is the case if only grey states are visited infinitely often. The grey states are equal to the set $V' \backslash (J'^1 \cup J'^2)$. Therefore this case is equal to the Streett game, because there is also a second chance for the system to win the game if neither J'^1 nor J'^2 are seen infinitely often. In the last case, when only red states are visited, the environment wins the parity games. This is again the same case with the Streett game, because there the environment fulfills the left side of the Streett winning condition implication, but the system cannot fulfill the right side (if, then we would see again green states). Therefore we

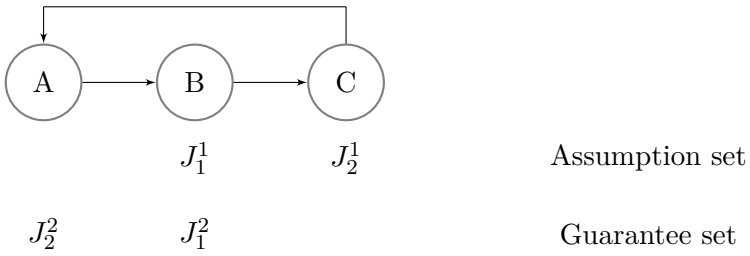

Figure 3.2: Example of a GR(1) game graph

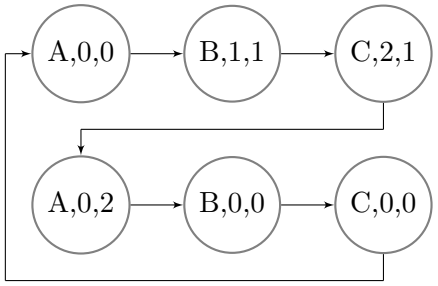

Figure 3.3: Example of an applied counting construction

can redefine the above Streett game as a parity game with this 3 parities and would get the same winning region for each player, but the winning strategies may differ due to different game solving algorithms.

Example: We want to show a small example of how this counting construction works. Consider a game graph with the three states A, B and C, and the transition relation as depicted in figure 3.2. Further we have two assumption state sets, $J_1^1 = \{B\}$ and $J_2^1 = \{C\}$. Additionally two guarantee state sets, $J_1^2 = \{B\}$ and $J_2^2 = \{A\}$.

After applying the counting construction we get a graph as depicted in figure 3.3. The state names consist of the name as before, the current assumption counter and the current guarantee counter. We have to note that the game graph after an applied counting construction would have more states then those we have depicted in the above figure, e.g. (A,1,0) to (B,1,1) or (C,1,0) to (A,1,0) because the counting construction calculates each permutation of the two counter values. But since we initialize both counter to zero, the 6 states from the above figure are the only which are reachable.

3.3 Solving the new game

Since we have transformed our game into a Street game with one pair or into a 3-color parity game, we now want to discuss some algorithms which

are solving them.

3.3.1 Streett Games

There are severeral possibilities to solve Streett games:

One of them is to transform the Streett game into a parity game and solve this resulting game, which is shown by Buhrke et al. [10]. The transformation is done by the use of the so called Index Appearance Record, a concept which was first introduced by Gurevich and Harrington [21]. This transformation can be done in time $O(n^2 2^{r \log r})$ and with memory size $r!r^2$, where n is the number of states and r is the number of Streett pairs in the acceptance condition.

Another possibility is to directly solve Streett games like Horn [23], where he uses an algorithm which is based on an algorithm of Zielonka [44], which will be discussed in the next chapter. The algorithm of Horn solves Streett games in time $O(r!n^{2r})$, again with n is the number of states and r is the number of Streett pairs.

A slightly improved algorithm is from Piterman and Pnueli [32], where they achieve runtimes of $O(mn^r rr!)$. But the great advantage of this solution is the possibility to easily implement this algorithm symbolically, which results in a runtime of $O(n^{r+1}r!)$, where m is the number of transitions, r and n denote the same as above. The algorithm is based on a recursive fixpoint calculation.

3.3.2 Parity Games

The fastest algorithm currently known to solve parity games is the Small Progress Measure algorithm developed by Jurdziński [26]. It achieves an upper runtime bound of $O(dm(\frac{n}{d})^{\lceil d/2 \rceil})$, where n is the number of states, m is the number of transitions and d is the maximum priority of the game graph. This algorithm has a major drawback: it is not easy to implement it symbolically, but there exists an approach from Bustan et al. [16], where he suggests Algebraic Desicion Diagrams (ADDs) to represent the ranking symbolically.

An algorithm, which can be implemented very easily in a symbolic manner, is from Sohail and Somenzi [39], which itself is an extension of an algorithm of Jurdziński et al. [27] to also calculate the winning strategy, whereas the latter only calculates the winning region. Both algorithm have a worst case runtime behaviour of $O(2^n)$, where n is again the number of the vertices of the game graph.

Another algorithm, which is symbolically implementable, is derived of the constructive proof from Zielonka [44], where he showed that parity games are determined and both players have a memoryless winning strategy. The derivation is from Grädel et al. [20], and has a worst case runtime of $O(mn^d)$

with the above denotations. This algorithm as well as the one before are defined recursively.

3.3.3 Discussion

Friedmann and Lange [18] compared different algorithms as well as some optimizations in a practical manner. They found out, that despite of the better upper runtime bound of the Progress Measure algorithm from Jurdziński, the recursive algorithms sometimes have better practical runtimes (e.g. 3 times faster), on other examples they can also be slower. So there is no clear winner in the practical challenge, which gives us hope that the recursive algorithms can perform quite well for our purposes. It is important to mention that Friedmann and Lange only used enumerative implementations for these comparisons. But since we are using symbolic implementations results can look quite better.

Due to this results and the slightly difficult algorithm of Piterman and Pnueli [32] for solving a Streett game, we decided to implement the last two recursive parity game algorithms and compare them with the original algorithm already implemented in the RATSY tool. Because there was no effort for implementation, we also used the GR(1) synthesis algorithm of Piterman et al. [33] for our algorithms comparison.

3.4 Recursive Algorithm for Solving Parity Games

3.4.1 Preliminaries

In the following we will further explain the algorithm of Jurdziński et al. [27] with the extension of Sohail and Somenzi [39]. For deeper explanation of the constructive proof of Zielonka [44], we refer to [20].

At first we want to state some key elements which are important for the algorithm, deeper explanation and proofs can be found in [20, 44, 39, 27].

Subarenas

Recall a game arena $A = (V_0, V_1, E)$ with the two vertex sets V_0 and V_1 and the edge relation E as introduced in section 2.3.1, but in this case of parity games enhanced with the parity function $c : V \to \{0, 1, \ldots, d\}$. We also call this type of an arena a coloured arena.

Also we want to state again one important restriction: the graph has to be bipartite, which means, each successor of a node belonging to Player 0 belongs to Player 1 and vice versa.

To get a game, we simply have to add a winning condition, which remains the same for all parity games in this section and is as defined before:

$$\varphi : \rho \in W \leftrightarrow \max(\mathrm{Inf}(c(\rho))) \text{ is even},$$

meaning Player 0 wins if the highest parity appearing infinitely often is even, otherwise Player 1 wins.

Hence we can use the two terms game and arena in this section virtually interchangeably.

For any subset $U \subseteq V$ of V, the subgraph induced by U as

$$A[U] = (V_0 \cap U, V_1 \cap U, E \cap (U \times U), c_{|U})$$

where $c_{|U}$ is the restriction of c to U (not all parities of A have to be also in U, also it may happen that the highest priority is lower than d).

There is also one important restriction: in a subarena every vertex has to have a successor, as in a normal game arena, so not all subgraphs are an arena. Also we want to mention that the bipartite property follows directly because we do not introduce new edges. So a subarena together with the above defined winning condition φ forms an ordinary parity game. To be more general, we will often talk about a Player σ ($\sigma \in \{0, 1\}$), and his opponent $1 - \sigma$.

We also want to define some abbreviations or functions: Given a parity game $G = (V_0, V_1, E, c)$ as defined before, we denote

- the union of the vertex sets $V_0 \cup V_1$ by $V(G)$,

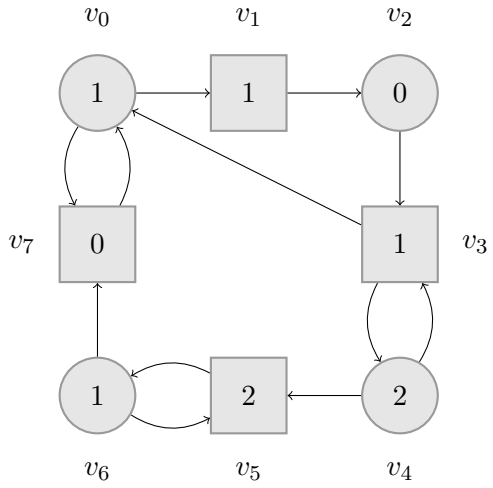

Figure 3.4: Example of a parity game

- $d(G) = \max\{c(v)|v \in V(G)\}$ be the highest priority of all vertices of the game G and
- $A_d(G) = c^{-1}(d)$ be the set of all vertices labelled by d.

Example: Look at the example game graph in Fig. 3.4, with vertices $v_0, ..., v_7$ and the three colors 0, 1, 2 (depicted inside the nodes), circles belong to Player 0 and boxes to Player 1.

The subgraph consisting of the three vertices $\{v_4, v_5, v_6\}$ together with the edges between these nodes form a subarena, whereas this subgraph enhanced witch the vertex v_7 does not, because v_7 would be a dead end (has no outgoing edge) in which it is not one in the whole arena.

Traps

A trap for a Player σ (called σ-trap) is subset $U \subseteq V$ so that if the token is on a node $v \subseteq U$, the Player σ cannot escape from this set U, because all successors of nodes belonging to him are part of U, and at least one successor of each node belonging to his opponent $1 - \sigma$ are also inside U. So while Player σ cannot escape, Player $1 - \sigma$ can always force the play to stay in U.

Formally, this is defined as

- $\forall v \in U \cap V_\sigma, \forall v' \ (v, v') \in E : v' \in U$ and
- $\forall v \in U \cap V_{1-\sigma}, \exists v' \ (v, v') \in E : v' \in U$.

Example: Consider again the game graph in Fig. 3.4: the subset $\{v_4, v_5, v_6\}$ is a 1-trap because Player 1 is not able to leave this part of the game. An-

other, the subset $\{v_0, v_1, v_2, v_3, v_7\}$ is a 0-trap. And both traps are also subgames as defined before.

Lemma 3.4.1 ([20]). *For every σ-trap U of G, $G[U]$ is a subgame.*

Example: The subset $\{v_4, v_5, v_6, v_7\}$ from the above figure is not a 1-trap because from v_7 there is an edge to a vertex outside the trap set but it is required through the definition that all edges from a players vertex are leading into the trap set. Another explanation is that Player 0 cannot force the token to stay in the subset $\{v_4, v_5, v_6, v_7\}$ because once the token is on vertex v_7 the only successor is vertex v_0. And this subset forms no subgame (as shown above). Conversely, we have a trap $\{v_4, v_5, v_6\}$ and this is a subgame by definition.

Attractor

The attractor is the key element for calculating the winning regions of a parity game.

It is a least fixed point operation and can be executed for both players, therefore we are talking about a 0-attractor (for Player 0) resp. 1-attractor (for Player 1).

The procedure is quite simple: We initialize the σ-attractor with any set of starting vertices X and add them to the current attractor set. In the next step we find those vertices of the current game that are not part of the current attractor set

- and have an edge leading to one of the vertices that are already in the attractor set if these new vertices belong to the Player σ,
- or if all edges leading to the current attractor set if these potential new vertices belong to Player $1 - \sigma$.

Repeat this latter step until no new vertices can be found.

The result is the greatest subset of the game vertices, from which the corresponding player can force the token to reach a vertex of the starting set X in finitely many steps.

Formally, the attractor set for a Player σ can be defined inductively: Start with the target set

$$X^0 = X,$$

And add new vertices:

$$\begin{aligned} X^{i+1} = & X^i \cup \{v \in V_\sigma \mid \exists v' : \ (v, v') \in E \wedge v' \in X^i\} \cup \\ & \{v \in V_{1-\sigma} \mid \forall v' : \ (v, v') \in E \wedge v' \in X^i\} \end{aligned}$$

We denote the attractor set for a Player σ starting with the set X by $Attr_\sigma(X)$.

Example: Look at the game graph in Fig. 3.4: The 0-attractor for the subset $X = \{v_4, v_5\}$ is just enhanced by the vertex v_6 because this vertex is controlled by Player 0 and has an edge to $v_5 \in X$, but the other adjacent vertices (v_7, v_3) are controlled by the other player and have edges not only to X. So the resulting attractor is $\{v_4, v_5, v_6\}$

Attractor Strategies

During the execution of the attractor we also want to remember the way how the starting set can be reached from within the attractor set, this is called the attractor strategy.

The calculation goes in parallel with the attractor calculation: initially start with an empty set and whenever a new vertex is added to the current attractor set (that belongs to the corresponding player), add all transitions leading to the current attractor set from this new vertex.

Formally, the attractor strategy for a Player σ can be defined inductively: Start with an empty set

$$\text{attr}^0 = \emptyset,$$

And add new vertices:

$$\text{attr}^{i+1} = \text{attr}^i \cup \{(v, v') \mid v \in V_\sigma \wedge v \in X^{i+1} \wedge v' \in X^i \wedge (v, v') \in E\}.$$

The set X^i is the attractor set of the i-th iteration as defined before.

We denote the attractor strategy for a Player σ starting with the set X by $attr_\sigma(X)$.

We have to mention one important thing: According to the definition of a strategy (see section 2.3.4), a strategy is a function that assigns to each vertex at most one outgoing edge. The above attractor strategy is therefore not a function, because it can assign more than one outgoing edge to a vertex. So we have to do a kind of postprocessing: After the calculation of the attractor strategy, we just have to go through all vertices and do the following for every vertex: delete an outgoing edge as long as the number of these edges is greater than one, or delete none if there is only one or none.

Lemma 3.4.2 ([27]). *Let $X \subseteq V$ be a subset of the vertex set V, then the set $V \backslash attr_\sigma(X)$ is a σ-trap in the game with vertex set V, for any Player σ.*

If you take any subset of vertices of a game G, calculate the attractor for any Player σ of this subset, divide G into two parts

- the calculated attractor set for this Player σ and
- everything else

then the Player σ cannot force the token into the calculated attractor set if the token is outside of this set. This also means, that if you calculate an

attractor of some subset of a game and subtract this attractor set from the whole game, the rest is also a subgame, or $\emptyset$ if every vertex is part of the attractor.

Example: Considering the game graph in Fig. 3.4, we will start with the subset $X = \{v_4, v_5\}$, the 0-attractor is $\{v_4, v_5, v_6\}$, and the game without these vertices is $\{v_0, v_1, v_2, v_3, v_7\}$, which is a 0-trap as shown in an example before.

The next lemma follows straigthly:

Lemma 3.4.3 ([44]). *Let $X \subseteq V$ be a σ-trap in the game G, then $attr_{1-\sigma}(X)$ is also a σ-trap.*

This means that an opponents trap can be extended through an attractor calculation.

Example: Again in the game graph in Fig. 3.4, take the subset $X = \{v_5, v_6\}$, this is also a 1-trap. Calculating the 0-attractor results in $\{v_5, v_6, v_4\}$, and this set is also a 1-trap.

The following lemmas are important for the algorithm in the next section:

Lemma 3.4.4 ([27]). *Let $G = (A, c)$ be a parity game, let σ denote a player (0 or 1) and $R_\sigma(G)$ is the winning region of G for the Player σ. If $X \subseteq R_\sigma(G)$, then $R_\sigma(G) = R_\sigma(G \backslash Attr_\sigma(A, X)) \cup Attr_\sigma(A, X)$ and $R_{1-\sigma}(G) = R_{1-\sigma}(G \backslash Attr_\sigma(A, X))$.*

This means that we can compose the winning region for a specific player through the union of the following two sets:

- the attractor of any subset of the winning region and
- the winning region of the same player of the smaller subgame (the original game without the vertices of the above attractor vertices)

The winning region of the other player is equal to his winning region of the smaller subgame.

Example: Look at the game graph in Fig. 3.5, obviously Player 0 can win from the vertices v_0 or v_1, because Player 1 can move the token only to v_1 and if Player 0 moves it back forever, the token visits infinitely often the two parities 1 and 2, so Player 0 wins. So we name $X = \{v_0, v_1\} \subseteq R_\sigma$. Calculating the 0-attractor of this set X results in the extension by vertex v_2, so $\mathrm{Attr}_\sigma(A, X) = \{v_0, v_1, v_2\}$. If we separate this attractor set of the game we get the subgame built by the vertices $\{v_3, v_4, v_5, v_6\}$. In this subgame it is easy to see that Player 1 can win from v_3 (choosing v_6, highest parity is 1), and Player 0 can win from v_5 (choosing always v_4 as the successor, highest parity is 0), so $R_{1-\sigma}(G \backslash \mathrm{Attr}_\sigma(A, X)) = \{v_3, v_6\}$ and $R_\sigma(G \backslash \mathrm{Attr}_\sigma(A, X)) = \{v_4, v_5\}$.
And through the above lemma we know that $R_\sigma(G) = \{v_4, v_5\} \cup \{v_0, v_1, v_2\}$ and $R_{1-\sigma}(G) = \{v_3, v_6\}$

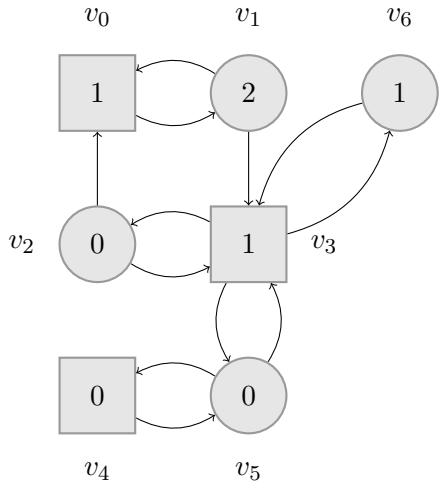

Figure 3.5: Example of a parity game

Lemma 3.4.5 ([27])**.** *Let $G = (A, c)$ be a parity game, $d = d(G)$ be the highest priority of the game G, $D = D_d(G)$ be the set of all vertices with the highest priority d and $\sigma = d \bmod 2$. Then, $R_{1-\sigma}(G \backslash Attr_\sigma(G, D)) \subseteq R_{1-\sigma}(G)$. Also, if $R_{1-\sigma}(G \backslash Attr_\sigma(G, D)) = \emptyset$, then $R_\sigma(G) = V(G)$.*

We take the highest parity of a game, assign it to a player (if it is even, Player 0, if it is odd, we will use Player 1). We will then calculate the attractor for that player of the set of vertices with the highest parity. Then we know that the winning set of the opponent of the subgame resulting from subtracting this calculated attractor set from the original game is a subset of the opponents winning region of the whole game. Also if the opponent cannot win within this subgame, the primary player would win from the entire game G.

Example: Consider again Fig. 3.5, the highest parity ocurring is 2, so we assign Player 0 to it. The vertex set denoted with this parity is $\{v_1\}$, the 0-attractor is $\{v_1, v_0, v_2\}$, the subgame without these vertices is built through the vertices $\{v_3, v_4, v_5, v_6\}$. Because Player 1 wins this subgame from the vertices $\{v_3, v_6\}$, we know that the Player 1 also wins the whole game from this vertices, so $\{v_3, v_6\} \subseteq R_1(G)$.

Take a look at the parity game in 3.6, it is nearly the same as before, but without the vertex v_6. The 0-attractor of v_1 is the same as calculated before ($\{v_0, v_1, v_2\}$), but now Player 1 has no chance to win in the subgame $\{v_3, v_4, v_5\}$, $R_1(G \backslash \mathrm{Attr}_0(G, D)) = \emptyset$, so Player 0 wins from each vertex in the game G, $R_0(G) = V(G)$

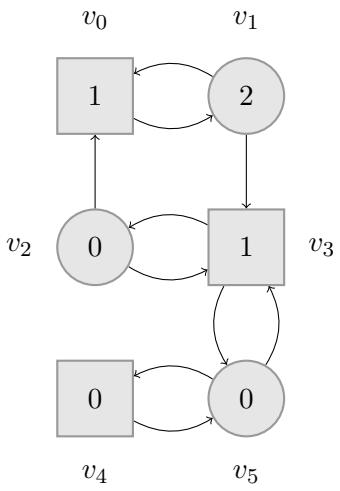

Figure 3.6: Example of a parity game

3.4.2 Algorithm for calculating the winning region

The actual algorithm to calculate the winning region of a parity game is shown in the Algorithm 1. It takes a parity game G as input and delivers the winning regions R_0 and R_1 for the two players.

Algorithm 1 winreg(G) [27]

1: **if** $V(G) = \emptyset$ **then**
2: **return** $(\emptyset, \emptyset)$
3: **end if**
4: $d = d(G)$
5: $A = A_d(G)$
6: $\sigma = d \bmod 2$
7: $(R_0, R_1) = \textbf{winreg}(G \setminus \mathrm{Attr}_\sigma(G, A))$
8: **if** $R_{1-\sigma} = \emptyset$ **then**
9: $R_\sigma = V(G)$
10: **else**
11: $(R_0, R_1) = \textbf{winreg}(G \setminus \mathrm{Attr}_{1-\sigma}(G, R_{1-\sigma}))$
12: $R_{1-\sigma} = V(G) \setminus R_\sigma$
13: **end if**
14: **return** (R_0, R_1)

Now we want to give a short explanation: The very first if-statement is obvious, if we don't have any vertices in our game, both winning regions (for each player) are empty, this is our termination condition for the recursion. In lines 4–6 we determine the highest priority in the game, the vertex set corresponding with this priority and the associated player. In line 7 we

are performing the first recursion step due to Lemma 3.4.5 by calculating the winning regions of the original game less the attractor of the Player σ corresponding with the highest priority (starting at the highest priority vertices). We are denoting this subgame as G'. If the other Player $1-\sigma$ has an empty winning region ($R_{1-\sigma} = \emptyset$) in this subgame, the Player σ wins from all vertices of G (lines 8 and 9) and we are done.

We also know from lemma 3.4.5, that the winning region of Player $1-\sigma$ of this subgame is a subset of his winning region of the whole game G. We are now at the beginning of line 11 and know that $R_{1-\sigma}$ of the game G' is a subset of the winning region $R_{1-\sigma}$ of G. So we calculate the $1-\sigma$-attractor of this subset $R_{1-\sigma}(G')$, because we know from lemma 3.4.4 that the winning region of the whole game is equal to the winning region of this subset for Player σ ($R_\sigma(G) = R_\sigma(G \backslash \text{Attr}_{1-\sigma}(G, X))$ where X is the winning region of the smaller game G' of Player $1-\sigma$). After the next recursion step we get the winning region R_σ for Player σ. And because parity games are fully determined (each vertex is assigned to one of the both winning regions of the two players, see chapter 2.3.8) we can calculate the winning region of Player $1-\sigma$ easily by taking all vertices that are not in R_σ (Line 12).

Since every recursion step operates on a strictly smaller game (because the attractor is not empty when there are still vertices in the current subgame, because we initialize the attractor with the highest priority vertices from which there is at least one), the whole recursion is terminating.

Example: Consider the example parity game in Fig. 3.4. We will show how the algorithm works and how the winning regions for both players are calculated:

1. recursion step: winreg($G = \{v_0, ..., v_7\}$) this is the main entrance:
Denoting some variables: $d = 2$, $A = \{v_4, v_5\}$, i = 0, j = 1
Calculating the attractor in line 7: $\text{Attr}_0(G, \{v_4, v_5\}) = \{v_4, v_5\_v_6\}$
(the _ in the set denotes new added vertices in the attractor calculation)
call winreg with $G \backslash \text{Attr}_0 = \{v_0, v_1, v_2, v_3, v_7\}$: winreg($\{v_0, v_1, v_2, v_3, v_7\}$) (as explained in recursion step 2)
this call returns $R_0 = \emptyset, R_1 = \{v_0, v_1, v_2, v_3, v_7\}$
so $R_1 \neq \emptyset$ (if-statement in line 8)
$\text{Attr}_1(G, \{v_0, v_1, v_2, v_3, v_7\}) = \{v_0, v_1, v_2, v_3, v_7\_\}$ (nothing added to the attractor)
call winreg($\{v_4, v_5, v_6\}$) (recursion step 3)
returns $R_0 = \{v_4, v_5, v_6\}, R_1 = \emptyset$
Result : $\mathbf{R_1} = V(G) \backslash R_0 = \{\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_7}\}$; $\mathbf{R_0} = \{\mathbf{v_4}, \mathbf{v_5}, \mathbf{v_6}\}$

2. winreg($G = \{v_0, v_1, v_2, v_3, v_7\}$)
$d = 1$, $A = \{v_0, v_1, v_3\}$, i = 1, j = 0
$\text{Attr}_1(G, \{v_0, v_1, v_3\}) = \{v_0, v_1, v_3, \_v_2, v_7\}$

call winreg($\emptyset$)
returns $R_0 = \emptyset, R_1 = \emptyset$
so $R_0 = \emptyset$
$\mathbf{R_1} = V(G) = \{\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_7}\}$; $\mathbf{R_0} = \emptyset$

3. winreg($G = \{v_4, v_5, v_6\}$)
$d = 2$, $A = \{v_4, v_5\}$, i = 0, j = 1
$\text{Attr}_0(G, \{v_4, v_5\}) = \{v_4, v_5, \_v_6\}$
call winreg($\emptyset$)
returns $R_0 = \emptyset, R_1 = \emptyset$
so $R_1 = \emptyset$
$\mathbf{R_0} = V(G) = \{\mathbf{v_4}, \mathbf{v_5}, \mathbf{v_6}\}$; $\mathbf{R_1} = \emptyset$

So the result of the whole game can be seen in the first recursion step: Player 0 wins from $\{v_4, v_5, v_6\}$, while Player 1 wins from the other vertices $\{v_0, v_1, v_2, v_3, v_7\}$.

3.4.3 Algorithm for calculating the winning strategy

Sohail and Somenzi [39] have enhanced algorithm 1 with the additional calculation of the winning strategy of a given parity game. It takes again a parity game G as input and delivers both winning regions (R_0 and R_1) together with the winning strategy for both players (t_0 and t_1).

Algorithm 2 winstrat(G) [39]

1: **if** $V(G) = \emptyset$ **then**
2: **return** $(\emptyset, \emptyset, \emptyset, \emptyset)$
3: **end if**
4: $d = d(G)$
5: $A = A_d(G)$
6: $\sigma = d \bmod 2$
7: $(R_0, R_1, t_0, t_1) = \textbf{winstrat}(G \setminus \text{Attr}_\sigma(G, A))$
8: **if** $R_{1-\sigma} = \emptyset$ **then**
9: $R_\sigma = V(G)$
10: $t_\sigma = \text{attr}_\sigma(G, A) \cup t_\sigma \cup \{(u, v) \mid u \in A \cap V_\sigma, v \in R_\sigma\}$
11: **else**
12: $(R_0, R_1, w_0, w_1) = \textbf{winstrat}(G \setminus \text{Attr}_{1-\sigma}(G, R_{1-\sigma}))$
13: $R_{1-\sigma} = V(G) \setminus R_\sigma$
14: $t_\sigma = w_\sigma$
15: $t_{1-\sigma} = t_{1-\sigma} \cup \text{attr}_{1-\sigma}(G, R_{1-\sigma}) \cup w_{1-\sigma}$
16: **end if**
17: **return** (R_0, R_1, t_0, t_1)

The only difference between the two algorithms is the enhancement by strategy calculation, which is done in the lines 10, 14 and 15. So the core operating mode remains the same as explained before.

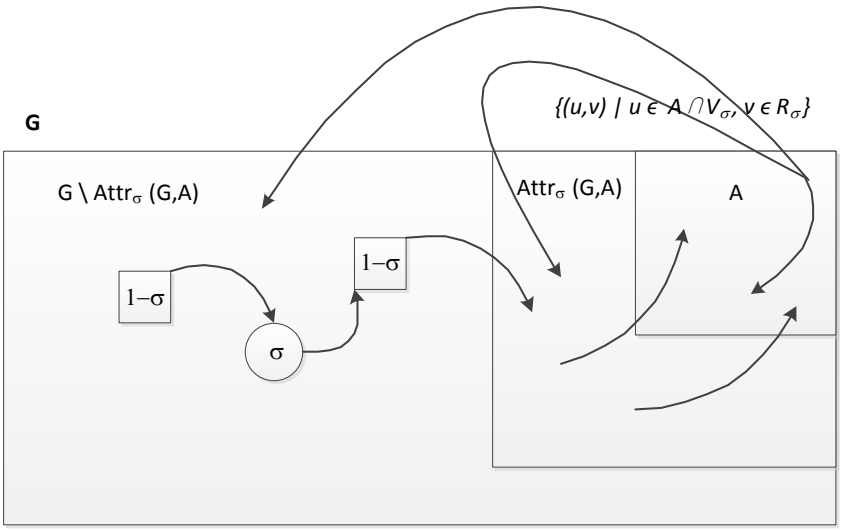

Figure 3.7: Scheme of a game G

Consider line 10: In this state we have a game G that consists of three parts, as depicted in figure 3.7:

- the set A including the vertices with the highest priority,
- the attractor for Player σ of A in the game G ($\text{Attr}_\sigma(G, A)$) and
- everything else ($G \setminus \text{Attr}_\sigma(G, A)$).

Because Player $1 - \sigma$ has no possibility to win from anywhere in G, his winning strategy is empty (as returned by the recursive call in Line 7). The Player σ has all possibilities to win and therefore we can connect the winning strategies of the different parts through the union of the following three strategies:

- The attractor strategy $\text{attr}_\sigma(G, A)$, which defines a strategy from a state anywhere in $\text{Attr}_\sigma(G, A)$ into the set A in finitely many steps.
- The winning strategy t_σ which defines a strategy inside the subset $G \setminus \text{Attr}_\sigma(G, A)$ of G as calculated in the beforehand recursion.
- The last thing that is missing is the connection between A and the whole game. Because we need a strategy out of the vertices of A that are reached through the attractor strategy to close the circle to define a strategy that never ends. It doesn't matter to which subset of G our transition(s) out of A are leading, because from anywhere in G a strategy is defined, as long as every vertex in A has an outgoing edge.

Note that we don't need (more exactly: we don't have a possibility) to connect $(G \setminus \text{Attr}_\sigma(G, A))$ with $\text{Attr}_\sigma(G, A)$, because if there would be a vertex controlled by Player σ in this subset that is leading into $\text{Attr}_\sigma(G, A)$, this vertex would be part of the attractor (consider the definition of the attractor strategy). So only Player $1 - \sigma$ can move the token into $\text{Attr}_\sigma(G, A)$, and he has no other possibilities because Player σ wins from all vertices in the current subgame.

Now we want to explain the second strategy calculation as it is defined in the lines 14 and 15 in the case that both players have winning vertices: Since the winning region for Player σ in G is the same as the recursive call in line 12 returns (as explained in 3.4.2), we can also take the winning strategy for Player σ from this recursive call as in line 14 defined.

The calculation of the strategy for Player $1-\sigma$ is a bit more complicated:

After the second recursion at line 12 we can separate the vertices of G into the following four disjoint sets (as depicted in Fig. 3.8):

- the winning region of Player $1 - \sigma$ as determined through the first recursion from which we know that it is a subset of the winning region for Player $1 - \sigma$ of the whole game G, denoted by $R''_{1-\sigma}$,
- the attractor of this winning region $(\text{Attr}_{1-\sigma}(G, R''_{1-\sigma}))$ but without the set $R''_{1-\sigma}$,
- the winning region of Player σ of the subgame $G \setminus \text{Attr}_{1-\sigma}(G, R''_{1-\sigma})$ (now denoted by R'_σ) and
- the winning region of the other player of the subgame $G \setminus \text{Attr}_{1-\sigma}(G, R''_{1-\sigma})$ (now denoted by $R'_{1-\sigma}$)

Now the calculation of the winning strategy for Player $1 - \sigma$ is obvious, it consists of

- his winning strategy $w_{1-\sigma}$ of the subgame $G \setminus \text{Attr}_{1-\sigma}(G, R''_{1-\sigma})$,
- the attractor strategy $\text{attr}_{1-\sigma}(G, R''_{1-\sigma})$ and
- the winning strategy $t_{1-\sigma}$ calculated in the first recursion that connects the vertices in $R''_{1-\sigma}$

The situation is similar as before, the token can stay in $R'_{1-\sigma}$ forever, or Player σ moves it into $\text{Attr}_{1-\sigma}(G, R''_{1-\sigma})$ (since the subgame $G \setminus R'_\sigma$ is a σ-trap, he cannot move it into R'_σ).

If the token finally reaches through the attractor strategy a vertex of $R''_{1-\sigma}$, he can stay there forever because this is a valid subgame and Player $1 - \sigma$ is also winning there and does not have to move out.

Example:

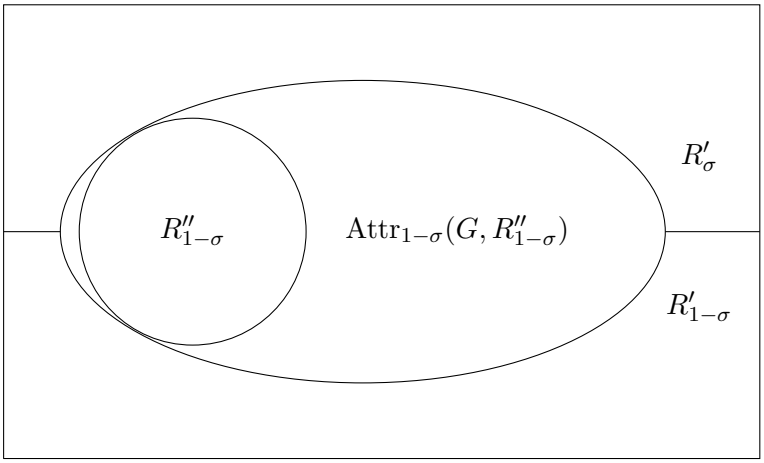

Figure 3.8: Different Subsets in of a game

Consider again the example parity game in Fig. 3.4. We will show now the enhancement by the calculation of the winning strategy:

1. winstrat$(G = \{v_0, ..., v_7\})$
$d = 2$, $A = \{v_4, v_5\}$, i = 0, j = 1
$\mathrm{Attr}_0(G, \{v_4, v_5\}) = \{v_4, v_{5\_}v_6\}$
$\mathrm{attr}_0(G, \{v_4, v_5\}) = \{v_6 \to v_5\}$
call winstrat$(\{v_0, v_1, v_2, v_3, v_7\})$ (as explained in recursion step 2)
returns $R_0 = \emptyset, R_1 = \{v_0, v_1, v_2, v_3, v_7\}, t_0 = \emptyset, t_1 = \{v_7 \to v_0, v_1 \to v_2, v_3 \to v_0\}$
so $R_1 \neq \emptyset$
$\mathrm{Attr}_1(G, \{v_0, v_1, v_2, v_3, v_7\}) = \{v_0, v_1, v_2, v_3, v_{7\_}\}$
$\mathrm{attr}_1(G, \{v_0, v_1, v_2, v_3, v_7\}) = \emptyset$
call winstrat$(\{v_4, v_5, v_6\})$ (step 3)
returns $R_0 = \{v_4, v_5, v_6\}, R_1 = \emptyset, w_0 = \{v_6 \to v_5, v_4 \to v_5\}, w_1 = \emptyset$
$\mathbf{R_1} = V(G) \backslash R_0 = \{\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_7}\}$; $\mathbf{R_0} = \{\mathbf{v_4}, \mathbf{v_5}, \mathbf{v_6}\}$
$t_1 =< t_1$ from line 7 $> \cup <$ attr$_1$ from line 12 $> \cup < w_1$ from line 12 $>$:
$\mathbf{t_1} = \{\mathbf{v_7} \to \mathbf{v_0}, \mathbf{v_1} \to \mathbf{v_2}, \mathbf{v_3} \to \mathbf{v_0}\} \cup \emptyset \cup \emptyset$; $\mathbf{t_0} = \{\mathbf{v_6} \to \mathbf{v_5}, \mathbf{v_4} \to \mathbf{v_5}\}$

2. winstrat$(G = \{v_0, v_1, v_2, v_3, v_7\})$
$d = 1$, $A = \{v_0, v_1, v_3\}$, i = 1, j = 0
$\mathrm{Attr}_1(G, \{v_0, v_1, v_3\}) = \{v_0, v_1, v_3, \_v_2, v_7\}$
$\mathrm{attr}_1(G, \{v_0, v_1, v_3\}) = \{v_7 \to v_0\}$
call winstrat$(\emptyset)$
returns $R_0 = \emptyset, R_1 = \emptyset, t_0 = \emptyset, t_1 = \emptyset$
so $R_0 = \emptyset$
$\mathbf{R_1} = V(G) = \{\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_7}\}$; $\mathbf{R_0} = \emptyset$;
$t_1 =<$ attr$_1$ from line 7$> \cup < t_1$ from line 7$> \cup <$strategy out of A (line

10)>:
$\mathbf{t_1} = \{\mathbf{v_7} \to \mathbf{v_0}\} \cup \emptyset \cup \{\mathbf{v_1} \to \mathbf{v_2}, \mathbf{v_3} \to \mathbf{v_0}\}$; $\mathbf{t_0} = \emptyset$

3. winstrat$(G = \{v_4, v_5, v_6\})$
$d = 2$, $A = \{v_4, v_5\}$, i = 0, j = 1
$\text{Attr}_0(G, \{v_4, v_5\}) = \{v_4, v_5, \_v_6\}$
$\text{attr}_0(G, \{v_4, v_5\}) = \{v_6 \to v_5\}$
call winstrat($\emptyset$)
returns $R_0 = \emptyset, R_1 = \emptyset, t_0 = \emptyset, t_1 = \emptyset$
so $R_1 = \emptyset$
$\mathbf{R_0} = V(G) = \{\mathbf{v_4}, \mathbf{v_5}, \mathbf{v_6}\}$; $\mathbf{R_1} = \emptyset$; $\mathbf{t_0} = \{\mathbf{v_6} \to \mathbf{v_5}\} \cup \emptyset \cup \{\mathbf{v_4} \to \mathbf{v_5}\}$; $\mathbf{t_1} = \emptyset$

The result of the whole game again can be seen in the first recursion step: Player 0 wins from $\{v_4, v_5, v_6\}$, while Player 1 wins from the other vertices $\{v_0, v_1, v_2, v_3, v_7\}$. The strategy for Player 0 is $\{v_6 \to v_5, v_4 \to v_5\}$, while for Player 1 is $\{v_7 \to v_0, v_1 \to v_2, v_3 \to v_0\}$.

Consider the example game in figure 3.9. We have 6 vertices and 3 priorities (the priority is denoted inside of each vertex). The vertices denoted by a circle are controlled by Player 0, the others by Player 1. It is very easy to see that Player 1 has no chance to win this game, hence the winning region for Player 0 consists of all 6 states. But Player 0 has two winning strategies to win the game:

- the first goes through the vertices v_4, v_5, v_3, v_2, where we will see infinitely often the highest priority 2 which is even,
- and the second strategy that alternates between the vertices v_0 and v_1, where the highest priority is 0 which is also even.

According to our definition of the attractor calculation, that an outgoing edge from a new vertice is going to a vertex that is one step closer, we will calculate the latter strategy.

We want now to show the complete algorithm execution:

1. winstrat$(G = \{v_0, ..., v_5\})$
$d = 2$, $A = \{v_4, v_5\}$, i = 0, j = 1
$\text{Attr}_0(G, \{v_4, v_5\}) = \{v_4, v_5\}$
$\text{attr}_0(G, \{v_4, v_5\}) = \emptyset$
call winstrat($\{v_0, v_1, v_2, v_3\}$) (is explained in recursion step 2)
returns $R_0 = \{v_0, v_1, v_2, v_3\}, R_1 = \emptyset, t_0 = \{v_1 \to v_0, v_3 \to v_0\}, t_1 = \emptyset$
so $R_1 = \emptyset$
$\mathbf{R_0} = V(G) = \{\mathbf{v_0}, ..., \mathbf{v_5}\}$; $\mathbf{R_1} = \emptyset$; $\mathbf{t_0} = \emptyset \cup \{\mathbf{v_1} \to \mathbf{v_0}, \mathbf{v_3} \to \mathbf{v_0}\} \cup \{\mathbf{v_4} \to \mathbf{v_5}\}$; $\mathbf{t_1} = \emptyset$

2. winstrat($G = \{v_0, v_1, v_2, v_3\}$)
$d = 1$, $A = \{v_2, v_3\}$, i = 1, j = 0
$\mathrm{Attr}_1(G, \{v_2, v_3\}) = \{v_2, v_3\}$
$\mathrm{attr}_1(G, \{v_2, v_3\}) = \emptyset$
call winstrat($\{v_0, v_1\}$) (step 3)
returns $R_0 = \{v_0, v_1\}, R_1 = \emptyset, t_0 = \{v_1 \to v_0\}, t_1 = \emptyset$
so $R_0 \neq \emptyset$
$\mathrm{Attr}_0(G, \{v_0, v_1\}) = \{v_0, v_1, \_v_3, v_2\}$
$\mathrm{attr}_0(G, \{v_0, v_1\}) = \{v_3 \to v_0\}$ (here we can only add this edge)
call winstrat($\emptyset$)
returns $R_0 = \emptyset, R_1 = \emptyset, w_0 = \emptyset, w_1 = \emptyset$
$\mathbf{R_1} = \emptyset$; $\mathbf{R_0} = V(G) = \{\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}\}$; $\mathbf{t_1} = \emptyset$; $\mathbf{t_0} = \{\mathbf{v_1} \to \mathbf{v_0}\} \cup \{\mathbf{v_3} \to \mathbf{v_0}\} \cup \emptyset$
;

3. winstrat($G = \{v_0, v_1\}$)
$d = 0$, $A = \{v_0, v_1\}$, i = 0, j = 1
$\mathrm{Attr}_0(G, \{v_0, v_1\}) = \{v_0, v_1\}$
$\mathrm{attr}_0(G, \{v_0, v_1\}) = \emptyset$
call winstrat($\emptyset$)
returns $R_0 = \emptyset, R_1 = \emptyset, t_0 = \emptyset, t_1 = \emptyset$
so $R_1 = \emptyset$
$\mathbf{R_0} = V(G) = \{\mathbf{v_0}, \mathbf{v_1}\}$; $\mathbf{R_1} = \emptyset$; $\mathbf{t_0} = \emptyset \cup \emptyset \cup \{\mathbf{v_1} \to \mathbf{v_0}\}$; $\mathbf{t_1} = \emptyset$

As you can see, the algorithm delivers the following strategy: $t_0 = \{v_1 \to v_0, v_3 \to v_0, v_4 \to v_5\}$.

When you reconsider the definition of the priorities (we have denoted all vertices, where all guarantees are fulfilled, with the priority 2, and all vertices where we have neither seen all guarantees nor all assumptions with priority 0), it becomes clear, that the system has the possibility to win the game by only circulating over vertices with priority 0. This means that the system is avoiding to fulfill all guarantees which usually is not the intention of the hardware designer, the goal is that the circuit generated through the winning strategy visits infinitely often all guarantees in order to do useful work. This is still an open problem and can only be addressed by writing complete specifications that don't allow such traps. Further reading can be done in [2].

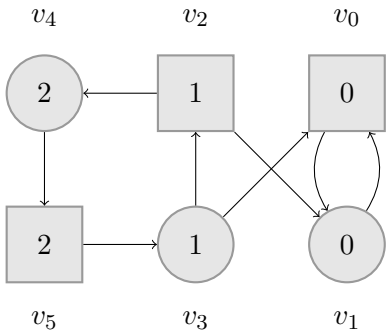

Figure 3.9: Example game

Chapter 4

Implementation

4.1 Counting Construction

Since the game graph of the GR(1) game is defined as the conjunction of the two transition relations ($\rho_e \wedge \rho_s$, see section 2.3.7) it is sufficient to apply the counting construction only on one transition relation in order to be valid on the whole game graph. Since the additional state variables, which are needed to store the two counter, are only needed by the system to find the right path to fulfill its specification, we will apply the counting construction only on the system transition relation. In the following we will talk about transitions, but we mean system transitions.

4.1.1 Extension of the state space

This is done in a simple way, we only have to add enough state variables to the transition relation to store all m different values of the assumption counter, exactly $\lceil \log_2 m \rceil$ variables. Similar we have to add $\lceil \log_2 n \rceil$ state variables for the guarantee counter.

4.1.2 Inserting the edges

After we have extended our state space, we have to insert the edges to implement the counting construction. This could be done very easily to loop over all edges of the GR(1) game and to insert for each original edge new edges in the new game for each possible count permutation. But because we operate with states and transition relations in a symbolic manner, we cannot iterate over single states or edges. Therefore we have to find a way to work with sets of states or transitions. Elements of such sets should share similar properties in an arbitrary way. An obvious possibility might be the property: All members of a transition set are leading to states which belong to the same assumption resp. guarantee state set.

So we take an individual assumption state set (J_i^1) and an individual guarantee state set (J_k^2). Then we divide the set of all transitions into the following four parts: The set of transitions, which is leading into

- both, the assumption and guarantee state set (both counters have to be incremented),
- only to the assumption set (only the assumption counter has to be incremented),
- only to the guarantee set (only the guarantee counter has to be incremented),
- or to neither the assumption set nor the guarantee set (both counters keep their values).

Note, that these 4 parts (subsets) are distinct and the union results in the original transition set.

If we have a transition set, where all members are leading to the assumption state set J_i^1 and to the guarantee state set J_k^2, then we have only to add to all these transitions the following counting transition:
assumption counter: $i - 1 \rightarrow i$
guarantee counter: $k - 1 \rightarrow k$
Similar to the other three cases es drafted above.

Such a composition can be done very easily symbolic: We have introduced new state variables for both counters, we will denote them $\mathcal{C}_e$ for the variables which hold the assumption counter, and $\mathcal{C}_s$ for the guarantee counter. Since we work with transitions, we will define a counter transition for the assumption counter by $\rho_a^c(i \rightarrow j)$, where i is the source count of the transition and j is the target count. This counter transition ρ_a^c is defined over the current and next states of variables of $\mathcal{C}_e$. We will do the similar for the guarantee counter transition ρ_g^c which is defined over $\mathcal{C}_s$ and $\mathcal{C}_s'$. Since we have a system transition relation $\rho_g(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ and the two counter transitions $\rho_a^c(\mathcal{C}_e, \mathcal{C}_e')$ and $\rho_g^c(\mathcal{C}_s, \mathcal{C}_s')$, we can simply compose them by conjunction because they have distinct state variables.

In order to generate the given assumption resp. guarantee sets for the partitioning from above, we have to create all possible permutations of them. This is done by two nested for loops, one is iterating over the different assumption state sets, the inner one over the guarantee state sets.

We want to show the implementation in algorithm 3. With $\rho_s(\mathcal{X}'\mathcal{Y}' \in J)$ we want to denote the subset of transitions which are leading into a state in J.

In line 5 we take all transitions which are leading into the sets J_k^2 and J_i^1 and add to them the counting transition of increment both counters. This

Algorithm 3 Implementation of the counting construction

1: result = 0
2: **for** $i = 0 \rightarrow m - 1$ **do** {loop over the assumption sets}
3: **for** $k = 0 \rightarrow n - 1$ **do** {loop over the guarantee sets}
4: result = result $\vee$
5: $(\rho_s(\mathcal{X}'\mathcal{Y}' \in (J_k^2 \cap J_i^1)) \wedge \rho_a^c(i \rightarrow i+1) \wedge \rho_g^c(k \rightarrow k+1)) \vee$
6: $(\rho_s(\mathcal{X}'\mathcal{Y}' \in (\neg J_k^2 \cap J_i^1)) \wedge \rho_a^c(i \rightarrow i+1) \wedge \rho_g^c(k \rightarrow k)) \vee$
7: $(\rho_s(\mathcal{X}'\mathcal{Y}' \in (J_k^2 \cap \neg J_i^1)) \wedge \rho_a^c(i \rightarrow i) \wedge \rho_g^c(k \rightarrow k+1)) \vee$
8: $(\rho_s(\mathcal{X}'\mathcal{Y}' \in (\neg J_k^2 \cap \neg J_i^1)) \wedge \rho_a^c(i \rightarrow i) \wedge \rho_g^c(k \rightarrow k)) \vee$
9: $(\rho_s \wedge \rho_a^c(- \rightarrow 0) \wedge \rho_g^c(n \rightarrow 0)) \vee$
10: $(\rho_s(\mathcal{X}'\mathcal{Y}' \in J_k^2) \wedge \rho_a^c(m \rightarrow 0) \wedge \rho_g^c(k \rightarrow k+1)) \vee$
11: $(\rho_s(\mathcal{X}'\mathcal{Y}' \notin J_k^2) \wedge \rho_a^c(m \rightarrow 0) \wedge \rho_g^c(k \rightarrow k))$
12: **end for**
13: **end for**
14: **return** result

is similar with the next 3 lines. In line 9 to 11 we are implementing the reset is it was defined earlier. A note to line 9, we could add this transitions outside of the for loops because it doesn't depend on i or k, but it is easier to keep it inside and it is negligible in terms of program runtime. The minus in $\rho_a^c(- \rightarrow 0)$ means that the source state is not specified (is set to don't care), therefore all possible values would be considered.

4.2 Old RATSY synthesis algorithm

As we have already mentioned, we want to also use the original synthesis algorithm for GR(1) games from Piterman et al. [33], which is already implemented in the RATSY tool. We can do this without any further code changes because the new game resulted by applying the counting construction is nothing else than a well defined GR(1) game. But now with only one assumption set (all states where the assumption counter = the number of assumptions) and one guarantee set. There are two reasons for that:

- We want to verify our implementation of the counting construction, because when the new synthesis fails, and we have used a well known and implemented algorithm, the only cause for failure can be in the counting construction.
- The main difference of solving a game where the counting construction is applied to one without, is the increase of the state space (which means more variables in our BDDs and therefore greater BDDs with more nodes) and therefore an increase in the manipulation time of some particular BDDs, which is also very interesting.

4.3 Recursive algorithm

4.3.1 Attractor

Idea

This is the most important part of the new algorithm. As it is defined in subsection 3.4, we start from some subset of the vertices of the game G and add vertices to the current attractor set as long as new vertices are leading into this current attractor set (at least one edge is leading into this attractor set in the case that the corresponding player is controlling the considered vertex or all edges from the considered vertex are leading into the current attractor set in the case the other player is controlling this vertex).

One can imagine a simple program doing this as defined in algorithm 4. This algorithm takes the game G and the start set A and returns the attractor set for player σ together with the attractor strategy t.

Algorithm 4 $\mathbf{attr}_\sigma(G, I)$

1: $B = I$
2: $t = 0$
3: **while** B is changing **do**
4: $\quad B^t = 0$
5: $\quad$ **for all** $\{v \in V_\sigma \mid \exists v' : \ (v, v') \in E \wedge v' \in B\}$ **do**
6: $\quad\quad B^t = B^t \cup \{v\}$
7: $\quad\quad t = t \cup \{(w, w') \mid (w, w') \in E, w = v, w' \in B\}$
8: $\quad$ **end for**
9: $\quad B = B \cup B^t$
10: $\quad$ **for all** $\{v \in V_{1-\sigma} \mid \forall v' : \ (v, v') \in E \wedge v' \in B\}$ **do**
11: $\quad\quad B = B \cup \{v\}$
12: $\quad$ **end for**
13: **end while**
14: **return** (B, t)

Note that we build up a temporary vertex set B^t(t ... temporary) in line 6 which is added to B after the first for loop, because in the calculation of the attractor strategy (line 7) we want to force that only edges into B are added (to ensure that with each while loop step only edges to vertices with a smaller rank are added). The states, from which one player can reach a given set in one step regardless what the other player does, we will also denote by the controlled predecessors (calculated in the lines 5 and 10).

In order to use this algorithm in our synthesis tool RATSY, we have to address the following problems resp. incompatibilities:

- Since we are using symbolic implementation, we should not iterate over individual vertices or edges.

ρ_e:			ρ_s:			$\rho_e \wedge \rho_s$:		
$\mathcal{XY}$	$\rightarrow$	$\mathcal{X}'$	$\mathcal{XYX}'$	$\rightarrow$	$\mathcal{Y}'$	$\mathcal{XY}$	$\rightarrow$	$\mathcal{X}'\mathcal{Y}'$
0-	$\rightarrow$	0	00-	$\rightarrow$	0	00	$\rightarrow$	-0
00	$\rightarrow$	1	01-	$\rightarrow$	-	10	$\rightarrow$	1-
11	$\rightarrow$	0	10-	$\rightarrow$	-	11	$\rightarrow$	-1
1-	$\rightarrow$	1	11-	$\rightarrow$	1	01	$\rightarrow$	0-

Table 4.1: Two transitions functions and the conjunction of them

- We have no explicit distinction between system and environment states.

Considering the first problem, we have to deal with vertex or transition sets instead of iterating over individuals.

Introduction of system states

Now we want to discuss the latter: The current framework for circuit syntheses used in RATSY is based on [33]. As defined in chapter 2, the game structure doesn't consist of two distinct sets of vertices or states, just only of one type, which is defined as the current valuation of all state variables. Whereas the transitions are well defined: $\rho_e \wedge \rho_s$.

Consider the small example in figure 4.1 of the game structure of a simple arbiter with two variables, one input variable (request, r) and one output variable (grant, g). Each state is labelled with the combination of these two variables, the first digit means the current valuation of request, the latter one of the grant. When we play a game, we place the token initially at the initial state (e.g. state 00). The first decision has to be made by the environment, which has two possibilities: either it can set its request signal to 1 or to 0, because the state 00 has to successors (00 and 10), and the request signal has two valuations in them. After that, the system has the choice: But all successor states of 00 have 0 as the grant signal valuation, therefore the system can only set grant to 0, and so on. The transition functions are depicted in table 4.1 (a minus '-' as a valuation means the corresponding bit can either be 0 or 1).

With this consideration, we can introduce system states in distinction to environment states: We can leave the states as before (a current valuation of all variables), just rename them environment states, because the environment has the choice whenever the token will be placed on such a state. After each environment state we will insert a system state for each possible valuation of the input variables, which is also allowed by the environment transition relation ρ_e and draw an edge from the environment state to all these new inserted system states. Also we will have to draw the outgoing edges for the system states according to the system transition relation ρ_s:

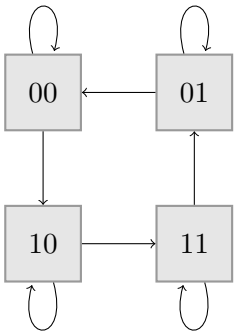

Figure 4.1: Example game structure of a simple arbiter

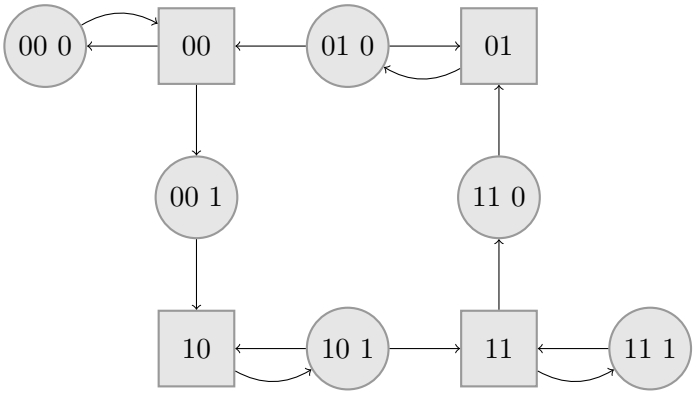

Figure 4.2: Example game structure of a simple arbiter enhanced by system states

For each possible valuation of the output variables of each system state we draw an edge to the corresponding existing environment state (through ρ_s we get the next variable valuations and those denote the next environment state). The enhanced game structure in this way of figure 4.1 is depicted in figure 4.2. A general description is given in figure 4.3. We start from an environment state $(\mathcal{X}, \mathcal{Y})$ which is followed by a system state defined by the current valuation plus the next input valuation $(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ then again an environment state, defined by the subsequent valuation of the input and output signals $(\mathcal{X}', \mathcal{Y}')$.

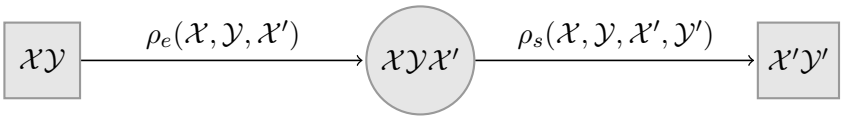

Figure 4.3: General discription of the introduction of the system states

Now we can calculate the attractor of a given game. Therefore we don't have to rebuild the whole game graph to insert the system states. It is enough to consider them in the following way: If we want to calculate for a given set of environment states all system states where the system can force the play into those environment states (system controlled predecessors), we can do this by simply calculating $\exists_{o'} : \rho_s \wedge S'_e$, where S_e are a set of environment states and a primed environment state set means to swap the present and the next part of all variables. Similar with o', where we want to filter out those states where at least on edge is leading to our desired target state set. This formula can be read as follows: We want to reach all states where at least one edge ($\exists_{o'}$) is leading to our target state set (S_e) traversing along the allowed system transitions (ρ_s).

When we want to go one step further, i.e. we want to calculate those environment states, which are leading to our desired system state set, but regardless what the environment does, we can calculate this set by $\forall_{i'} : \neg\rho_e \vee S_s$. This formula can be read as follows: The environment reaches the destination state set (S_s) or it does something wrong (i.e. it violates its transition relation ρ_e), and this for all possible inputs ($\forall_{i'}$).

If we are looking for the controlled predecessors, where the environment can force the play into a given state set, we simply have to exchange the relations, sets and quantifiers as shown in the following table:

Player	Destination states	
System	environment states	$\text{coax}_s(S_e) = \exists_{o'} : \rho_s \wedge S'_e$
System	system states	$\text{coax}_s(S_s) = \forall_{i'} : \neg\rho_e \vee S_s$
Environment	environment states	$\text{coax}_e(S_e) = \forall_{o'} : \neg\rho_s \vee S'_e$
Environment	system states	$\text{coax}_e(S_s) = \exists_{i'} : \rho_e \wedge S_s$

Subgame

The last thing which is open for the complete attractor computation is to determine the controlled predecessors of a state set in a given subgame. We can define the subgame by the set of states which are occuring and the corresponding transition functions. If we get the state sets as a parameter, we can calculate the transition functions $\overline{\rho_s}$ and $\overline{\rho_e}$ by conjunction with the state sets like

$$\overline{\rho_s} : \rho_s \wedge S_s \wedge S'_e$$

$$\overline{\rho_e} : \rho_e \wedge S_e \wedge S_s$$

By using these subgame transition functions together with the above definition of the controlled predecessors (coax functions) we can work also in subgames.

Complete attractor computation

No we can calculate the attractor set together with the attractor strategy within a given subgame, similar to algorithm 4, but now inside the RATSY framework. This adapted algorithm is shown in algorithm 5, which takes the state sets of the considered subgame (S_s and S_e), the transition relations of the whole game (ρ_s and ρ_e) and the initial set from which the attractor computation should start, also separated into environment and system states (I_s and I_e) as the input. It then delivers the attractor state set (separated into system and environment states) and the attractor strategy only for the system, because we are only interested in the winning strategy of the system for the circuit synthesis and can omit the environment strategy due to performance reasons.

Algorithm 5 $\mathbf{attr}_\sigma(S_s, S_e, \rho_s, \rho_e, I_s, I_e)$

1: $B_e = I_e$
2: $B_s = I_s$
3: $\overline{\rho_s} = \rho_s \wedge S_s \wedge S'_e$
4: $\overline{\rho_e} = \rho_e \wedge S_e \wedge S_s$
5: **while** $B_e \vee B_s$ is changing **do**
6: **if** $\sigma == 0$ **then**
7: $B^t_s = \exists_{o'} : \overline{\rho_s} \wedge S'_e$
8: **else**
9: $B^t_s = \forall_{o'} : \neg\overline{\rho_s} \vee S'_e$
10: **end if**
11: $B^t_s = B^t_s \wedge \neg B_s \wedge S_s$
12: **if** $\sigma == 0$ **then**
13: $t = B^t_s \wedge B'_e \wedge \rho_s$
14: **end if**
15: $B_s = B_s \vee B^t_s$
16: **if** $\sigma == 0$ **then**
17: $B^t_e = \forall_{i'} : \neg\overline{\rho_e} \vee S_s$
18: **else**
19: $B^t_e = \exists_{i'} : \overline{\rho_e} \wedge S_s$
20: **end if**
21: $B^t_e = B^t_e \wedge \neg B_e \wedge S_e$
22: $B_e = B_e \vee B^t_e$
23: **end while**
24: **return** (B_s, B_e, t)

Short Description: We start with the calculation of the subgame transition relations, because they are the same for the whole attractor computation. Inside the loop we start with the computation of the controlled predecessor states leading into environment states, but we could do it also

in the other way round and start with environment states leading into system states. In line 11 and 21 we have to shrink the recently calculated predecessor states to be not in the current attractor set (B) but to be in the subgame (S). In line 13 we calculate the strategy, only for player 0 (which is the system), by determining the transitions going from the new states (B^t) into the current attractor states (B) alongside the system transition relation ρ_s. We end the loop when there are no new states which can be added to the current attractor set. It is also important to note that this calculation only works when only system states are predecessors of environment states and vice versa (the game graph has to be bipartite), which is the case as defined above.

4.3.2 Calculation of the winning region and strategy

Now as we have defined the attractor computation, the implementation of the algorithm 2 is quite straightforward, because the algorithm deals mainly with state sets, and these are interpreted internally in our RATSY framework one to one through BDD variables, similar with transition relations. Therefore we only have to exchange the union and intersection operators of the algorithm by logical OR resp. AND operators on BDDs and we are done. Since we now have a split up of the states into environment and system states, we have to do all state set manipulations twice. The same is valid for comparisons: if we want to check if a given state set is empty, we just have to check if there are no environment states and if there are no system states. To find out the highest priority of a given subset, we only have to test the following:

- If states are existing in this subset with the guarantee counter equal to the number of guarantees (green states), the highest priority is 2 and the state set associated to this priority is the state set consisting of the green states.
- If this is not the case, we look if the subgame consists of red states, then the highest priority is 1 and the associated state set is the set consisting of red states.
- If this is not the case, then there can be only grey states in the subgame, therefore the highest priority is 0 and the associated state set is the whole subgame.

4.4 Optimizations

After the calculation of the winning strategy for a given game, we want to show some optimization possibilities, in order to accelerate the calculation

time, reduce the amount of memory needed for this calculation or to reduce the complexity (=needed gates) of the resulting circuit.

In this section we only want to enumerate the different possibilities, we will try some out on real synthesis examples in the next chapter.

We can divide these efforts roughly into two approaches:

- Reduction of the game graph to get smaller BDD sizes resulting in faster operations on BDDs,
- try to accelerate BDD operations itself and
- reorder the BDDs.

optimize this calculation in respect to calculation time and needed memory, and also to yield a smaller result blif-file.

4.4.1 Simplification/Reduction of a game graph

Consider the input game graph which is defined by the specification of our resulting synthesized circuit. After applying the counting construction, we will get in the most cases a lot of new states, which can never be reached (as shown in section 4.1). Apart of that, there might also be states which are unreachable due to the specification itself (e.g. forbidden states).

So it might be useful if we try to shrink the game graph onto only those states, which are actual reachable from the initial state set. We can do this simply by calculating all reachable states (again a fixpoint calculation, beginning with the initial states, and add all successors as long as there are any to the result set), and combine this set with a logical AND operation with the transition relation of the system, because only there the counting construction is defined.

This results in a smaller game graph, but not necessarily into a smaller BDD (in the sense of BDD nodes), because we could introduce additional complexity.

But there is a special BDD operation defined and implemented in the CUDD package, named restrict [15, 14].

The restrict operator takes two arguments (f and g), the function to be simplified (f) and the function to which the first function should be restricted to (g). The result is equivalent to f where g is true and should be simpler than f. If it is not simpler, f will be returned. The result of restrict should satisfy the following [1]:

- if $(f \wedge g = 0)$ then $\text{restrict}(f, g) = 0$
- if $(f \vee \neg g = 1)$ then $\text{restrict}(f, g) = 1$

[1] From http://www-verimag.imag.fr/~raymond/tools/bddc-manual/bddc-manual-pages_10.html

- otherwise, it returns a function satisfying $(f \wedge g) \rightarrow \text{restrict}(f, g) \rightarrow (f \vee \neg g)$

Let f be the transition relation which we want to restrict and g be set of all reachable states in our game, the the function restrict(f, g) would return a transition relation which is the same inside all reachable states, but it is undefined outside. Since we will never reach unreachable states, we can apply the restrict operation on our transition relation and nothing bad would happen. Furthermore we can also apply the same restrict operation on all reachable states to our winning strategy right after the calculation of it before we will pass it to the output function generation.

Another approach is to try the two possible implementations of the reset of the two counters in the calculation of the counting construction, as mentioned in section 3.2. The first possibility is to reset the assumption counter also when the guarantee counter will be resetted. The second one is to perform the resets independently of each other. However, after some example synthesis runs we found out that there is no difference (less than one percent).

4.4.2 Accelerating BDD operations

After performing a profiling on the synthesis of some examples, we saw that most of the running time is consumed by the attractor computation, and there especially by the calculation of the controlled predecessors of the environment state sets. This is clear, because in these two operations (lines 7 and 9 of algorithm 5) the system transition relation is used which is quite huge through the applied counting construction. Beside the implementation of the two quantifier operators $\exists$ and $\forall$ in CUDD, there is a third one, a combined operator for $\exists$ and AND, which calculates $\exists : a \wedge b$ for two given BDD variables a and b in one step. It is also optimized for the combination of these two operations and therefore faster than a sequential calculation of $a \wedge b$ and applying $\exists$ onto this result.

Since we can express a $\forall$ operator with a $\exists$ operator, because

$$\forall x \equiv \neg\exists\neg x,$$

we can rewrite the line 9 of algorithm 5 and write

$$B_s^t = \neg\exists_{o'} : \overline{\rho_s} \wedge \neg S_e'$$

instead of

$$B_s^t = \forall_{o'} : \neg\overline{\rho_s} \vee S_e'$$

4.4.3 Reordering

Since the main disadvantage for using BDDs is their heavy dependence on the right variable ordering, CUDD provides two methods to do some reordering of the BDD variables to cut down the nodes of the individual BDDs:

- do some dynamic reordering which means that we give CUDD the freedom to decide the optimal moments for reordering, which is not influenceable by us, or
- give CUDD the explicit order to perform a reordering. This is useful when we have some moments in the execution of our whole program where we have done some calculations with BDDs and only need the result BDD without some parts of it. It seems useful to perform this type of reordering right after the construction of the counting construction and after the calculation of the winning strategy.

We can also combine these two methods and use both of them, which we have also tried in the next chapter.

Chapter 5

Comparison

In this chapter we want to try out the newly implemented algorithm (see section 3.4) for solving the parity games on some examples. Instead of give this comparison on some theoretical examples, we will present some real world designs (resp. specifications), which are not too small, so they give a good indication for some conclusion about the performance.

5.1 Case Study: Generalized Buffer

The generalized buffer (genbuf) was used earlier, also to build up GR(1) games and synthesize its specification, see [6]. There is also a detailed definition of the operation mode of the genbuf. Genbuf is a family of buffers parameterized by a number s. It transmits data from s senders to two receivers. Data can be sent by the senders in an arbitrary order and will be received by the receivers in round-robin order. The buffer implements a handshake protocol with each receiver and each sender. Each sender uses a signal to signalize the buffer that has a sending request, which the buffer will answer by an acknowledge, similar on the receiver side. We will not further explain the genbuf, because a complete specification already exists an we are only interested on the synthesis performance with different algorithms.

In figure 5.1 we have depicted some parts of this specification for a genbuf with one sender and two receiver.

5.1.1 Problem

But there is one great problem with this specification. The game graph built upon this specification has dead ends, which means that there are states in the graph which have no successor. And this violates the prerequisites of the algorithm of Jurdziński [26] (see section 2.3 and 3.4).

Because the specification is built up mostly direct of LTL-formulas which are not derived from complete Büchi automata, there may arise the following

```
[INPUT_VARIABLES]
StoB_REQ0;
RtoB_ACK0;
RtoB_ACK1;
:

[OUTPUT_VARIABLES]
BtoS_ACK0;
BtoR_REQ0;
BtoR_REQ1;
:

[ENV_INITIAL]
StoB_REQ0=0;
:

[ENV_TRANSITIONS]
G((StoB_REQ0=1 * BtoS_ACK0=0) -> X(StoB_REQ0=1));
G(BtoS_ACK0=1 -> X(StoB_REQ0=0));
G(BtoR_REQ0=0 -> X(RtoB_ACK0=0));
G((BtoR_REQ0=1 * RtoB_ACK0=1) -> X(RtoB_ACK0=1));
:
:

[ENV_FAIRNESS]
G(F(BtoR_REQ0=1 <-> RtoB_ACK0=1));
G(F(BtoR_REQ1=1 <-> RtoB_ACK1=1));

[SYS_INITIAL]
BtoS_ACK0=0;
BtoR_REQ0=0;
:

[SYS_TRANSITIONS]
G((BtoR_REQ0=1 * RtoB_ACK0=0) -> X(BtoR_REQ0=1));
G((BtoR_REQ0=0) + (BtoR_REQ1=0));
G(RtoB_ACK0=1 -> X(BtoR_REQ0=0));
G((BtoR_REQ1=1 * RtoB_ACK1=0) -> X(BtoR_REQ1=1));
G(RtoB_ACK1=1 -> X(BtoR_REQ1=0));
G((BtoR_REQ0=1 * BtoR_REQ1=1) -> FALSE);
:
:

[SYS_FAIRNESS]
G(F(StoB_REQ0=1 <-> BtoS_ACK0=1));
G(F(stateG12=0));
```

Figure 5.1: Part of the specification of genbuf

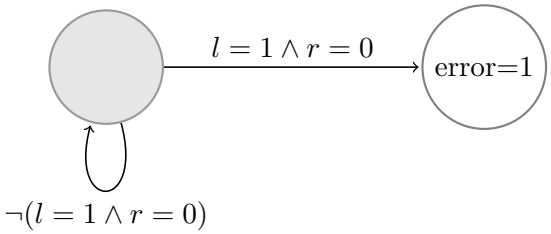

Figure 5.2: Example automata for the formula $G(l \rightarrow r)$ (the grey shaded state is the accepted state)

problem: Consider the following two LTL formulas:

- $G(a \rightarrow Xb)$
- $G(a \rightarrow X\neg b)$

Those two denote two transition relation formulas, and since the right parts (Xb and $X\neg b$) are contrary together, there will be no chance to fulfill both formulas if we reach a state where a holds. Therefore there will be no transition outside of states where a holds, therefore these states are dead ends.

And it is very likely that this is the case with the specification of the genbuf. Of course, not so obviously but in some nested form.

5.1.2 Possible solution

A possible solution might be to translate each LTL formula to an equivalent Büchi automaton, complete it and rewrite it as a LTL-formula [1]. Completing means that we have to draw for each state and for each variable valuation an outgoing edge if it is not existing. And these new outgoing edges will lead to some new generated error state because they where not defined and therefore forbidden. Consider the example formula $G(l \rightarrow r)$, which is well defined for each valuation of the two variables l and r, except $l = 1$ and $r = 0$, which is forbidden by the specification. Therefore we will introduce a new state $error = 1$ which will be a successor of the above forbidden transition, as depicted in figure 5.2.

Since we have two players we will introduce two error states, one for the system and one for the environment. But in the follwing we will rename this state to correct with the opposite valuation, correct=1 means everything ok, correct=0 depicts this erroneous state.

Instead of converting all LTL formulas to Büchi automata, we will just add to each transition relation formula the additional possibility to go to

[1] Translating a Büchi automaton to LTL with introduction of new state variables is possible in general

this errorstate as well (for a formula $G(\varphi)$ we write $G((\varphi) \vee correct = 0)$). So a player has two possibilities, he can fulfill the specification formula as desired, or go to an error state. Because he can do this from each state, we have no dead ends anymore, at least on transition to an error state exists. But we have to add some rules to deal with an error state, because it should be an objective to not enter it. So we will add the new transition condition $G(correct = 0 \rightarrow X(correct = 0))$ to define that once an errorstate is entered, it can never be left. Also we need to add the fairness condition $GF(correct = 1)$ to state that fulfilling the specification is only possible without entering the errorstate. It is also very important to define the initial condition for the error signal with $correct = 1$. An extract of the genbuf specification with this enhancements is depicted in figure 5.3. This specification has no dead ends anymore.

5.1.3 First synthesis run

Now we are ready to try out the new synthesis algorithm on this example. But the first result was a bit strange: It was very fast and the resulting circuit was very small. After deeper analysis we encountered, that the system was going with its correct signal in the first tick to 0 and therefore indicated that it is in an error state. But so it would loose the game because it was not fulfilling its specification. Unless the environment is also violating its specification. Since the environment will not violate its transition relation, the system must force the environment somehow to violate its fairness conditions. When we look at those environment fairness conditions at figure 5.1, there are two:

- `G(F(BtoR_REQ0=1 <-> RtoB_ACK0=1))` and
- `G(F(BtoR_REQ1=1 <-> RtoB_ACK1=1))`.

The REQ signal is an output signal, the ACK signal is controlled by the environment. This rules mean that sometimes (infinitely often) the environment has to set its input signal equal to some output signal. After some simulation runs on the synthesized circuit it turned out that the system was forcing the environment to violate this both conditions. Since the environment has to make the first choice and define the valuation of the input variables, the system comes next. So it is very easy for the system to set the REQ signal to the negated value of the ACK signal. Whatever the environment does, it cannot win this game.

But why is this problem only now arising? When we look at the system transitions at figure 5.1, there are some formulas over this variables, like

- `G(((BtoR_REQ0=1 * RtoB_ACK0=0) -> X(BtoR_REQ0=1)))`
- `G(((BtoR_REQ0=0) + (BtoR_REQ1=0)))`

```
[INPUT_VARIABLES]
StoB_REQ0;
:
ecorrect;

[OUTPUT_VARIABLES]
BtoS_ACK0;
:
scorrect;

[ENV_INITIAL]
StoB_REQ0=0;
ecorrect=1;
:

[ENV_TRANSITIONS]
G(((StoB_REQ0=1 * BtoS_ACK0=0) -> X(StoB_REQ0=1)) + X(ecorrect=0));
G((BtoS_ACK0=1 -> X(StoB_REQ0=0)) + X(ecorrect=0));
:
G((ecorrect=0  ) -> X(ecorrect=0));

[ENV_FAIRNESS]
G(F(BtoR_REQ0=1 <-> RtoB_ACK0=1));
:
G(F(ecorrect=1));

[SYS_INITIAL]
BtoS_ACK0=0;
:
scorrect=1;

[SYS_TRANSITIONS]
G(((BtoS_ACK0=1 * StoB_REQ0=1) -> X(BtoS_ACK0=1)) + X(scorrect=0));
:
G((scorrect=0  ) -> X(scorrect=0));

[SYS_FAIRNESS]
G(F(StoB_REQ0=1 <-> BtoS_ACK0=1));
:
G(F(scorrect=1));
```

Figure 5.3: Part of the specification of genbuf, enhanced by the introduction of an error state

- `G((RtoB_ACK0=1 -> X(BtoR_REQ0=0)))`

So there are transition formulas which force the system, for example, to leave the REQ0 signal high if the ACK0 signal is low. Therefore the system cannot do with the REQ0 signal what it wants to, it has to fulfill these rules. And then the environment can fulfill the above fairness conditions.

Due to the implementation of the synthesis algorithm, the winning strategy is always a subset of the transitions of the game graph. Therefore it is impossible for the system to violate its transition relation, it will always go ahead with it.

But now we have enhanced the above system transition relations in the following way:

- `G((((BtoR_REQ0=1 * RtoB_ACK0=0) -> X(BtoR_REQ0=1))) + X(scorrect=0))`
- `G((((BtoR_REQ0=0) + (BtoR_REQ1=0))) + X(scorrect=0))`
- `G(((RtoB_ACK0=1 -> X(BtoR_REQ0=0))) + X(scorrect=0))`

This means, that we have converted the transition relations to fairness conditions, i.e. the system has now every freedom for the output variables. If it does something wrong, it will enter an error state. It is not enough to formulate the fairness condition for the system to stay in a nonerror state forever. If it can force the environment to violate its part, it will win the game, and we have seen, that it does so.

A possible workaround for this dilemma might be to transform all transition formulas, where these REQ signals are used, back to the original form without the error state transitions. But we have done this because we had dead ends in our game graph. In this case we had luck, and by bringing this formulas into the old form, but leaving the other formulas dealing with an error state, led again to a game graph without dead ends. Otherwise we would had to transform all formulas to Büchi automata as we have described above.

5.2 Case Study: AMBA AHB

ARM's Advanced Microcontroller Bus Architecture (AMBA) defines the Advanced High-Performance Bus (AHB), an on-chip communication standard connecting such devices as processor cores, cache memory, and DMA controllers. Up to 16 masters and up to 16 slaves can be connected to the bus. Access to the bus is controlled by an arbiter, which is subject of this specification [5].

There exists also a well defined formal specification for GR(1) games for this arbiter, as explained in detail in [5].

We will denote in the following this arbiter shortly by amba, followed by a number denoting the number of clients which the arbiter has to deal with.

Again, when we used the amba-specification for circuit synthesis, we encountered exactly the same problems as we had with the genbuf. We had dead states, tried to workaround by the introduction of error states, but the system found a way to force the environment to a violation of its specification, also by going into an erroneous state. But in the end we got a specification without dead ends in the same way as with the genbuf.

5.3 Experimental Results

5.3.1 Simplification of the BDDs

At first we want to try out how the simplification ideas, which we discussed in chapter 4.4.1, are performing.

Therefore we synthesized both examples, an amba02 and a genbuf02. For each synthesis run we tried out the two possible BDD reductions on the system transition, after the counting construction was applied. Together with the case that we don't do any optimization. We denote this three cases by

- none
- and
- restrict

Also we tried out to optimize the strategy before the output functions will be created, we have therefore the same three possibilities as enumerated above. We tried out the combinations of these optimization methods with the specification of amba02 and with the algorithm 2. In the following table, the first column (cc) denotes the BDD optimizations method for the counting construction, the second for the strategy. The third column is the runtime of the whole synthesis process (incl. output function generation) and the last one is the filesize of the result blif-file. It is sorted by the runtime.

cc	strategy	runtime [s]	filesize [KB]
and	none	129	765
and	and	131	740
restrict	and	188	1700
none	and	189	1700
and	restrict	208	2100
none	none	231	2100
restrict	none	231	2100
restrict	restrict	280	2700
none	restrict	280	2700

We have done the same with the specification of the genbuf02:

cc	strategy	runtime [s]	filesize [KB]
none	none	39	1100
restrict	none	42	1100
none	restrict	45	1100
restrict	restrict	48	1100
none	and	62	872
restrict	and	65	872
and	and	84	604
and	none	114	1400
and	restrict	363	3400

As we can see, there is no optimal combination of the two optimization methods. It seems that applying to the counting construction the restrict operator and optimize the strategy with the and-operator is a good compromise.

5.3.2 Reordering

Now we want to compare the different reordering methods as described in section 4.4.3. There are the following possibilities for reordering:

- none,
- dynamic,
- static and
- both, dynamic and static

We have tried them out with the same algorithm, again with amba02 and genbuf02:

Reorder method	specification	runtime [s]	filesize [KB]
none	amba02	188	1700
dynamic		28	399
static		35	485
dynamic and static		45	498
none	genbuf02	65	872
dynamic		4	109
static		16	214
dynamic and static		6	96

It turns out that reordering has a great impact on both the runtime and the result file size. The complexity of the resulting circuit will be decreased through simpler BDDs because they are being converted more or less directly into combinatorial logic. Dynamic reordering performs best with both specifications, so we use it for the further comparisons.

5.3.3 Comparison of the different algorithms

Now we want to compare the following four algorithms:

- The already implemented synthesis algorithm without using the counting construction ([33]), in the following denoted by 'pit',
- the same algorithm but used with the counting construction, denoted by 'pitcc',
- the algorithm 2, denoted by 'jurd' and
- the algorithm of Zielonkas proof of parity games from [20], denoted by 'ziel'.

In the following table we list different measurements of the synthesis of this four algorithms with the specification of amba02:

algorithm	runtime [s]	filesize [KB]	strat [s]	mem [MB]	pre
pit	2.65	32	1.23	170	644
pitcc	21	83	5.26	322	329
jurd	28	399	3.56	363	269
ziel	36	447	8.41	375	504

The columns denote:

- algorithm: the used algorithm
- runtime: the total runtime of the synthesis process
- filesize: the file size of the resulting blif file
- strat: the runtime of the calculation of the winning strategy, without code generation
- mem: the maximum resident size of main memory needed for the synthesis
- pre: the number of controlled predecessor calculations

In the following table we show the same, but for the specification of genbuf03:

algorithm	runtime [s]	filesize [KB]	strat [s]	mem [MB]	pre
pit	1	12	0.36	130	411
pitcc	8	54	3.64	272	117
jurd	15	414	1.1	353	134
ziel	40	504	2.2	377	183

Discussion

As we can see, the algorithm pit performs in both examples a magnitude better than the others working with the counting construction, in the runtime as well as the file size of the generated circuit. He has to calculate more controlled predecessors, but since he works on BDDs with less variables, he works faster. Interesting is also, that the algorithm pit used with the counting construction works in the complete process of synthesis faster then jurd, but in terms of only strategy calculation he needs more time. It seems that this can be caused by a simpler strategy of pitcc, which is also apparent when we look onto the file sizes. Also we can see that the more complex and bigger the BDDs, the more main memory is used for synthesis.

We have also tried to profile a synthesis run, which means to look how often each function is called and how long does it take to return. We tried it with the synthesis of amba02 with the algorithm jurd and all optimizations as noted above, and only the attractor computation (accumulated over all function calls) took 4.15 seconds, the whole strategy computation 4.29 seconds. This as a bit more than in the table above due to the profiling overhead. But we can see that nearly the whole time of the strategy calculation is needed for the attractor computation.

Chapter 6

Conclusion

6.1 Summary

In this work, we addressed a great problem of synthesis of hardware circuits, which is slow synthesis performance. We tried to reduce a GR(1) game with different assumption and guarantee sets to one with just one guarantee set and one assumption set, which then can be seen as a parity game with 3 priorities or as a Streett game with one pair. These games are very popular in the science community, and great algorithms exist solving them. But this game reduction, based on a counting construction, can only be made at the price of a great state space increase. And it turned out, that the eventually better algorithms for parity or Streett games could not beat the old algorithm because of the increased complexity through the BDDs. We showed in this work that with this simple method it is not possible to achieve a performance increase.

6.2 Future Work

There is still great potential in this approach, and with the usage of better algorithms or enhancements of the existing ones maybe there can be a real performance increase.

One possibility might be to use the method which Jurdziński describes in [27]. There he proofs that the decomposition of the whole game graph into smaller SCCs (strongly connected components) could increase the performance with an upper bound of $n^{O(\sqrt{n})}$, where n is the number of vertices in the game graph. Finding SCCs symbolically is also well researched, e.g. in [19, 37, 4]

Or to use Streett game solvers as we have mentioned in section 3.3.1, e.g. Piterman and Pnueli [32]. Or to try out the Progress Measure Algorithm from Jurdziński [26], but symbolically, based on the work of Bustan et al. [16].

Bibliography

[1] R. Bloem, R. Cavada, I. Pill, M. Roveri, and A. Tchaltsev, *Rat: A tool for the formal analysis of requirements*, Computer Aided Verification, 2007, pp. 263–267.

[2] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, and B. Jobstmann, *Robustness in the presence of liveness*, Proc. Computer Aided Verification, 2010, To Appear.

[3] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Koenighofer, M. Roveri, V. Schuppan, and R. Seeber, *RATSY — a new requirements analysis tool with synthesis*, Proc. Computer Aided Verification, 2010, LNCS 6174, pp. 425–429.

[4] R. Bloem, H. N. Gabow, and F. Somenzi, *An algorithm for strongly connected component analysis in $n \log n$ symbolic steps*, Formal Methods in Computer Aided Design (W. A. Hunt, Jr. and S. D. Johnson, eds.), Springer-Verlag, November 2000, LNCS 1954, pp. 37–54.

[5] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, *Automatic hardware synthesis from specifications: A case study*, In Proceedings of the Design, Automation and Test in Europe, 2007, pp. 1188–1193.

[6] ———, *Specify, compile, run: Hardware form PSL*, 6th International Workshop on Compiler Optimization Meets Compiler Verification, 2007, Electronic Notes in Theoretical Computer Science http://www.entcs.org/.

[7] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann, *Robustness in the presence of liveness*, Lecture Notes in Computer Science **6174** (2010), 410–424.

[8] R. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers, vol. C-35, 1986, pp. 677–691.

[9] J. R. Büchi and L. H. Landweber, *Solving sequential conditions by finite-state strategies*, Trans. Amer. Math. Soc. **138** (1969), 295–311.

[10] N. Buhrke, H. Lescow, and J. Vöge, *Strategy construction in infinite games with Streett and Rabin chain winning conditions*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 96) (Passau, Germany), Springer, March 1996, LNCS 1055, pp. 207–225.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation **98** (1992), 142–170.

[12] A. Church, *Logic, arithmetic and automata*, Proceedings International Mathematical Congress, 1962.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*, MIT Press, Cambridge, MA, 1999.

[14] O. Coudert, C. Berthet, and J. C. Madre, *Verification of sequential machines based on symbolic execution*, Automatic Verification Methods for Finite State Systems (J. Sifakis, ed.), Springer-Verlag, 1989, LNCS 407, pp. 365–373.

[15] O. Coudert and J. C. Madre, *A unified framework for the formal verification of sequential circuits*, November 1990, pp. 126–129.

[16] Orna Kupferman Doron Bustan and Moshe Y. Vardi, *A measured collapse of the modal μ-calculus alternation hierarchy*, Lecture Notes in Computer Science, vol. 2996, 2004, pp. 522–533.

[17] E. A. Emerson and C. S. Jutla, *The complexity of tree automata and logics of programs (extended abstract)*, Proc. Foundations of Computer Science, 1988, pp. 328–337.

[18] Oliver Friedmann and Martin Lange, *Solving parity games in practice*, Lecture Notes in Computer Science, vol. 5799, 2009, pp. 182–196.

[19] R. Gentilini, C. Piazza, and A. Policriti, *Computing strongly connected componenets in a linear number of symbolic steps*, Symposium on Discrete Algorithms (Baltimore, MD), January 2003.

[20] E. Grädel, W. Thomas, and T. Wilke (eds.), *Automata, logics, and infinite games: A guide to current research*, Lecture Notes in Computer Science, vol. 2500, Springer, 2002.

[21] Y. Gurevich and L. Harrington, *Trees, automata, and games*, Proc. 14th ACM Symp. Theory of Comp. (San Francisco, CA), 1982, pp. 60–65.

[22] T. A. Henzinger and N. Piterman, *Solving games without determinization*, Proc. 15th Conference on Computer Science Logic, 2006, pp. 395–410.

[23] F. Horn, *Streett games on finite graphs*, Workshop on Games in Design and Verification (Edimburgh, UK), July 2005.

[24] B. Jobstmann and R. Bloem, *Optimizations for LTL synthesis*, 6th Conference on Formal Methods in Computer Aided Design (FMCAD'06), 2006, pp. 117–124.

[25] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, *Anzu: A tool for property synthesis*, Computer Aided Verification, 2007, pp. 258–262.

[26] M. Jurdziński, *Small progress measures for solving parity games*, STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science (Lille, France), Springer, February 2000, LNCS 1770, pp. 290–301.

[27] M. Jurdziński, M. Paterson, and U. Zwick, *A deterministic subexponential algorithm for solving parity games*, Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA 2006 (Miami, FL), January 2006, pp. 117–123.

[28] O. Kupferman, N. Piterman, and M. Y. Vardi, *Safraless compositional synthesis*, Eighteenth Conference on Computer Aided Verification, 2006, LNCS 4144, pp. 31–44.

[29] O. Kupferman and M. Y. Vardi, *Safraless decision procedures*, Foundations of Computer Science (Pittsburgh, PA), October 2005, pp. 531–542.

[30] K. L. McMillan, *Symbolic model checking*, Kluwer Academic Publishers, Boston, MA, 1994.

[31] N. Piterman, *From nondeterministic Büchi and Streett automata to deterministic parity automata*, 21st Symposium on Logic in Computer Science (LICS'06), 2006, pp. 255–264.

[32] N. Piterman and A. Pnueli, *Faster solutions of Rabin and Streett games*, Logic in Computer Science, 2006, pp. 275–284.

[33] N. Piterman, A. Pnueli, and Y. Sa´ar, *Synthesis of reactive(1) designs*, 7th International Conference on Verification, Model Checking and Abstract Interpretation, Springer, 2006, LNCS 3855, pp. 364–380.

[34] A. Pnueli, *The temporal logic of programs*, IEEE Symposium on Foundations of Computer Science (Providence, RI), 1977, pp. 46–57.

[35] A. Pnueli and R. Rosner, *On the synthesis of a reactive module*, Proc. Symposium on Principles of Programming Languages (POPL '89), 1989, pp. 179–190.

[36] M. O. Rabin, *Automata on infinite objects and Church's problem*, Regional Conference Series in Mathematics, American Mathematical Society, Providence, RI, 1972.

[37] K. Ravi, R. Bloem, and F. Somenzi, *A comparative study of symbolic algorithms for the computation of fair cycles*, Formal Methods in Computer Aided Design (W. A. Hunt, Jr. and S. D. Johnson, eds.), Springer-Verlag, November 2000, LNCS 1954, pp. 143–160.

[38] S. Safra, *On the complexity of ω-automata*, Symposium on Foundations of Computer Science, October 1988, pp. 319–327.

[39] S. Sohail and F. Somenzi, *Safety first: A two-stage algorithm for LTL games*, 9th International Conference on Formal Methods in Computer Aided Design (FMCAD'09), 2009, pp. 77–84.

[40] S. Sohail, F. Somenzi, and K. Ravi, *A hybrid algorithm for LTL games*, Verification, Model Checking and Abstract Interpretation (San Francisco, CA), January 2008, LNCS 4905, pp. 309–323.

[41] W. Thomas, *On the synthesis of strategies in infinite games*, Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, 1995, LNCS 900, pp. 1–13.

[42] Wolfgang Thomas, *Automata and reactive systems*, 2003, Lecture Note.

[43] Moshe Y. Vardi, *From church and prior to psl*, Lecture Notes in Computer Science, vol. 5000.

[44] W. Zielonka, *Infinite games on finitely coloured graphs with applications to automata on infinite trees*, Theoretical Computer Science **200** (1998), no. 1-2, 135–183.