Master's Thesis

Synthesizing Robust Systems

Bettina Könighofer

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie
Technische Universität Graz

Betreuer: Roderick Bloem

Graz, im Dezember 2012

Abstract

Property synthesis allows the automatic creation of systems from formal specifications [Chu62, PR89, BCG+10b]. Synthesized systems are *correct by construction*. There has been a lot of progress recently in making property synthesis practicable [PPS06, BGJ+07b, BGJ+07a, FJR11, MS08, SS09, SB00, SL09, VYY10, LNP+12, HJK10, GKP11]. Although one of the problems that remains is that synthesized systems often do not behave reasonably in unexpected situations, i.e., when environment assumptions are violated.

Many specifications consist of environment assumptions and system guarantees. Guarantees must be fulfilled only if all assumptions are satisfied. If assumptions are violated, than the system can behave arbitrary. For both assumptions and guarantees, we may distinguish between safety and liveness properties. Safety properties specify that *"something bad never happens"* and liveness properties specify that *"something good will happen eventually"* [MP92, AS85]. Also, it is not possible to detect violations of liveness properties at any point of time [AS85], whereas violations of safety properties are immediately apparent. For this reason, we build systems that are robust to safety failures. In order to define robustness, we define a system failure to be a violation of a safety guarantee, and an environment failure to be a violation of a safety assumption. *We define a system to be robust if a finite number of environment failures induces a finite number of system failures* [BGHJ09]. Let's assume the environment produces an environment failure for one tick. After some time, a robust system should recover and shouldn't produce system failures any more. Even if there is a finite number of environment failures, a robust system should still fulfill all liveness guarantees. Liveness properties state that some property will hold eventually. If there are finitely many environment failures, then the system works correctly for an infinitely long time, and should be able to fulfill all liveness guarantees.

This work presents an extension of the requirements analysis and synthesis tool RATSY [BCG+10b] that allows the synthesis of robust systems from GR(1) specifications [PPS06]. Our work is based on ideas from [BGHJ09] and [BCG+10a]. We turn a GR(1) specification into a one-pair Streett game such that a winning strategy corresponds to a correct implementation [BCG+10a]. Furthermore, we add a second pair such that the strategy corresponds to a robust system. We show how to compute the strategy, based on the algorithm of [PP06].

Keywords: Robust Systems, Synthesis, Reactive Systems, Formal Specifications, Streett Games, Counting Construction.

Kurzfassung

Formale Synthese ist in der Lage, automatisch *korrekte* Systeme aus formalen Spezifikationen zu erstellen [Chu62, PR89, BCG$^+$10b]. In letzter Zeit gab es viele wissenschaftliche Publikation, welche sich damit beschäftigten, den Synthese Prozess zu optimieren, um ihn auf praktische Probleme anzuwenden [PPS06, BGJ$^+$07b, BGJ$^+$07a, FJR11, MS08, SS09, SB00, SL09, VYY10, LNP$^+$12, HJK10, GKP11]. Jedoch gibt es noch immer offene Probleme, zum Beispiel dass sich synthetisierte Systeme in unerwarteten Situationen, in welchen Umgebungsannahmen verletzt sind, oft nicht wie gewünscht verhalten.

Viele Spezifikationen bestehen aus *Umgebungsannahmen* und *System-Garantien*. Garantien müssen nur dann erfüllt werden, wenn alle Annahmen erfüllt sind. Wenn Annahmen verletzt sind, darf sich das System beliebig verhalten. Sowohl für Annahmen als auch für Garantien unterscheiden wir zwischen so genannten Safety-Eigenschaften und Liveness-Eigenschaften. Safety-Eigenschaften spezifizieren, dass *etwas Schlechtes nicht eintreten wird* und Liveness-Eigenschfaten besagen, dass *irgendwann etwas Gutes eintreten wird* [MP92, AS85]. Weiteres ist es nicht möglich, zu irgendeinem Zeitpunkt eine Verletzung einer Liveness-Eigenschaft zu erkennen [AS85], jedoch kann eine Verletzungen einer Safety-Eigenschaft sofort detektiert werden. Aus diesem Grund konstruieren wir Systeme, welche robust sind hinsichtlich Verletzungen von Safety-Annahmen. Wir nehmen an, dass während der Ausführungszeit alle Liveness-Annahmen erfüllt werden. Diese Annahme ist gerechtfertigt, da die Umgebung zu jedem Zeitpunkt in der Lage ist, alle Liveness-Annahmen zu erfüllen, unabhängig von bisherigen Eingabe- und Ausgabewerten. Ein Systemfehler ist definiert als eine Verletzung einer Safety-Garantie und ein Umgebungsfehler ist definiert als eine Verletzung einer Safety-Annahme. *Ein System ist per Definition robust, wenn endlich viele Umgebungsfehler auch nur endlich viele Systemfehler induzieren* [BGHJ09]. Angenommen, die Umgebung erzeugt einen Umgebungsfehler für einen Tick. Nach einiger Zeit sollte sich das System erholen, und keine Systemfehler mehr produzieren. Auch wenn es während einer Ausführungszeit endlich viele Umgebungsfehler gibt, sollten alle Liveness-Garantien trotzdem erfüllt werden. Liveness- Eigenschaften besagen, dass eine bestimmte Eigenschaft irgendwann erfüllt werden muss. Auch wenn es eine beliebige endliche Anzahl an Umgebungsfehlern gibt, arbeitet das System trotzdem für eine unendlich lange Zeit korrekt und sollte in der Lage sein, alle Liveness-Garantien zu erfüllen.

Diese Arbeit präsentiert eine Erweiterung des Anforderungsanalyse und Synthese Tools RATSY [BCG$^+$10b] Diese Erweiterung ermöglicht es uns nun robuste Systeme von GR(1) Spezifikationen zu erzeugen [PPS06]. Die Arbeit basiert auf den Ideen von [BGHJ09] und [BCG$^+$10a]. Eine GR(1) Spezifikation kann in ein *Streett-game* mit einem Paar überführt

werden. Die Gewinnstrategie dieses Spieles korrespondiert mit einer korrekten Implementierung der Spezifikation [BCG+10a]. Ein zweites Paar wird zum Spiel hinzugefügt, sodass die Strategie nun mit einem robusten System korrespondieren würde. Wir präsentieren einen Algorithmus zum Berechnen der Strategie, welcher auf den Algorithmus in [PP06] basiert.

Schlagwörter: Robuste Systeme, Synthese, Reaktive Systeme, Formale Spezifikationen, Streett-Games, Counting-Constructions.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläe 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 entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Acknowledgements

This thesis was written in 2012 at the Institute for Applied Information Processing and Communications at Graz University of Technology.

First of all, I want to thank my supervisor Roderick Bloem, who encouraged me to work on this topic, who spent a lot of time and patience teaching me the required basics and who woke my interest in formal methods and verification. I learnt a great deal during this time, and for that I am very grateful. Although, he has the busiest schedule ever, he always found the time to answer questions, discuss new ideas and to eliminate ambiguities. Furthermore, I thank him for the opportunity of writing a paper about this work and being able to present it at the SYNT 2012 Workshop in Berkeley, California.

Special thanks to Georg Hofferek and Robert Könighofer, for valuable tips on the implementation, for their help in writing the paper and for explaining things from other areas in synthesis and verification to me. I really enjoyed working on my thesis, and they were largely responsible for that. Further, I want to thank Ayrat Khalimov and Swen Jacobs, for their help in preparing the presentation of the paper, and for the great time we spent together at the conference.

I thank my boyfriend Philip Weber for proofreading this work and for motivating and supporting me during my study. Moreover, I express my gratitude to my parents Franz Könighofer and Birgit Schiestl, for encouraging me to visit a technical school and consequently giving me the opportunity to attend the university in the first place.

Graz, Austria, September 2012 Bettina Könighofer

Danksagung

Diese Masterarbeit wurde im Jahr 2012 am Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie an der Technischen Universität Graz durchgeführt.

Zu Beginn möchte ich meinen Betreuer Roderick Bloem danken. Er ermutigte mich, an diesem Thema zu arbeiten, lehrte mich mit viel Zeit und Geduld die benötigten Grundlagen und weckte mein Interesse an Verifikation und formale Methoden. Ich habe während meiner Masterarbeit sehr viel gelernt, und dafür möchte ich mich herzlich bedanken. Obwohl sein Terminkalender immer voll ist, fand er stets die Zeit, um Fragen zu beantworten, neue Ideen zu diskutieren und Unklarheiten zu beseitigen. Des Weiteren danke ich ihm für die Gelegenheit, aus dieser Arbeit ein Paper verfassen zu dürfen, welches auf der SYNT 2012 in Berkeley, USA akzeptiert wurde und ich präsentieren durfte.

Ganz besonderer Dank gilt Georg Hofferek und Robert Könighofer. Sie gaben mir wertvolle Tips für die Implementierung meiner Arbeit, halfen mir beim Verfassen des Papers und gaben mir Einblicke auch in andere interessante Bereiche der formalen Methoden. Vor allem durch sie genoss ich die Zeit sehr, in der ich an dieser Thesis gearbeitet habe. Des Weiteren möchte ich Ayrat Khalimov und Swen Jacobs danken, für die Hilfe bei der Vorbereitung der Präsentation des Papers, und für die schöne Zeit, welche wir gemeinsam auf der Konferenz verbrachten.

Ich danke meinen Freund Philip Weber für sein Korrekturlesen und für die Motivation und Unterstützung während meines gesamten Studiums. Des Weiteren möchte ich meinen Eltern Franz Könighofer und Birgit Schiestl danken. Sie ermutigten mich in meiner Schulzeit eine Höhere Technische Lehranstalt zu besuchen und ermöglichten mir danach mein Studium.

Graz, im September 2012 Bettina Könighofer

Contents

1 Introduction **1**

1.1 Motivation for Correct Systems . . . 1
1.1.1 Motivation for Correct Systems . . . 1
1.1.2 Motivation for Robust Systems . . . 2
1.2 Property Synthesis . . . 3
1.2.1 History of Synthesis . . . 4
1.3 Our Method of Synthesizing Robust Systems . . . 7
1.4 Structure of this work . . . 8

2 Preliminaries **10**

2.1 Kripke Structures . . . 10
2.1.1 Computation Trees . . . 11
2.2 Labeled Transition System . . . 12
2.3 Automaton . . . 13
2.3.1 Words and Languages . . . 13
2.3.2 Finite Automata on Finite Words . . . 14
2.3.3 Finite ω-Automata . . . 15
2.3.4 Symbolic Representation of ω-Automata . . . 18
2.4 Mealy and Moore Machines . . . 18
2.4.1 Reactive Systems . . . 18
2.4.2 Mealy and Moore Machines . . . 19
2.5 Temporal Logics . . . 21
2.5.1 LTL . . . 23
2.5.2 LTL-Properties . . . 25
2.5.3 GR(1) . . . 26
2.5.4 CTL* and CTL . . . 28
2.5.5 Modal μ-Calculus . . . 29
2.6 Games . . . 31
2.6.1 Strategies . . . 32
2.6.2 Safety Games . . . 33
2.6.3 Reachability Games . . . 33
2.6.4 Büchi Games . . . 33
2.6.5 co-Büchi Games . . . 34
2.6.6 Muller Games . . . 34
2.6.7 Rabin Games . . . 34
2.6.8 Streett Games . . . 34
2.6.9 Parity Games . . . 34

2.6.10 GR(1) Games . . . 35
2.6.11 The modal μ-Calculus over Game Structures . . . 35
2.7 Implementation . . . 36
2.7.1 RAT . . . 36
2.7.2 RATSY . . . 38
2.7.3 Marduk . . . 40
2.8 Related Work on Robust Synthesis . . . 41

3 Robust Synthesis from GR(1) Specifications 44
3.1 Idea . . . 44
3.2 Definition of Robustness . . . 44
3.3 Illustration of the problem . . . 46
3.4 Example of Environment Failures and System Failures . . . 47
3.5 Robust Synthesis Algorithm . . . 49
3.5.1 GR(1) Specification to GR(1) Game . . . 49
3.5.2 GR(1) Game to one-pair Streett Game . . . 50
3.5.3 Robustness Streett pair . . . 52
3.5.4 Winning Region . . . 54
3.5.5 Winning Strategy . . . 60
3.6 Example of Robust Synthesis . . . 61
3.7 Recovery Time . . . 64

4 Implementation of Robustness and Results 66
4.1 Implementation of Robustness in RATSY . . . 66
4.1.1 RATSY . . . 66
4.1.2 Marduk . . . 67
4.2 Results . . . 68

5 Conclusions and Future Work 70
5.1 Conclusions . . . 70
5.2 Future Work . . . 70

List of Symbols 73

Bibliography 74

List of Figures

2.1 Kripke Structure Example. 11
2.2 Computation Tree Example. 12
2.3 Labeled Transition System Example. 13
2.4 Reactive System. 19
2.5 Graph of Mealy Machine M_1. 21
2.6 Graph of Moore Machine M_2. 22
2.7 Moore Machine for simple Arbiter. 27
2.8 Typical Design Workflow. 38
2.9 RATSY's User Interface while playing a Counter Game. 39

3.1 Non-robust Moore Machine. 46
3.2 Robust Moore Machine. 46
3.3 Possible Signal Trace, without Violation of Safety Properties. 47
3.4 Environment Transitions Relation ρ_e of one-Pair Streett Game. 48
3.5 System Transitions Relation ρ_s of one-Pair Streett Game. 48
3.6 Transitions Relation $\rho_e \wedge \rho_s$ of one-Pair Streett Game. 48
3.7 Robust System Generation Process. 50
3.8 Game Graph before Applying Counting Construction. 52
3.9 Game Graph after applying Counting Construction. 52
3.10 Strategy for States in Y_1 for one-Pair Streett Game. 59
3.11 Strategy for States in Y_{i+1} for a one-Pair Streett Game. 59
3.12 Illustration of the Iterates of the Fixpoint Computation. 60
3.13 Cutout of the Transition Relation of the two-Pair Streett Game. 62
3.14 Transition Relation of the two-Pair Streett Game. 63
3.15 Arbiter Example: Illustration of the Iterates. 63

4.1 Control Flow of extended MARDUK. 67

List of Tables

2.1 Mealy Machine M_1.
Transition relation δ. 20
2.2 Mealy Machine M_1.
Output function λ_1. 20
2.3 Moore Machine M_2.
Transition relation δ. 21
2.4 Moore Machine M_2.
Output function λ_2. 21

3.1 Winning Strategy for two-Pair Streett Game. 61

4.1 Synthesis Time for Arbiter. 68
4.2 Implementation Size for Arbiter. 69

List of Algorithms

1 `main_Streett`: Main Function to compute Winning Region 55

2 `Streett`: Recursive Function to compute Winning Region 56

3 `m_Streett`: Helper Function to compute Winning Region 56

1 Introduction

1.1 Motivation for Correct Systems

1.1.1 Motivation for Correct Systems

Today we are surrounded by computer systems and we need these systems to work correctly. If they do not, it could lead considerable financial implications or even put human lives in danger. Both hardware and software may be flawed. Errors in hardware are most likely irreversible and in such cases the hardware has to be replaced. In the past there have been many cases where both hardware and software errors caused great damage.

A famous example of a hardware error is the Pentium-FDIV-Bug. In the year 1994 a bug in the Pentium processor was discovered. This bug caused certain floating point divisions to go wrong. It cost Intel 475 million US dollars to replace the faulty CPUs [Pra95, Kro99].

In 1996, a software error caused massive financial losses. There was a bug in the navigation software of the Ariane 5 rocket, which caused the rocket to crash. During the conversion of a 64-bit float variable into a signed 16-bit integer variable an arithmetical overflow occurred, and this led to a total loss of steering and positioning data. The rocket left its anticipated course and had to be destroyed. The rocket had a value of 370 million US dollars [Nus97, Dow97]. Further examples of fatal software bugs can be found in [Gar05].

To prevent such errors, formal and informal methods have been developed. Testing and simulation are popular informal methods of finding bugs in hardware and software systems. Unfortunately, since today's systems are becoming larger and even more complex, these methods cannot be carried out exhaustively anymore [MS04]. This means that it is impossible to check the system's output for all the possible inputs and some cases stay untested. Therefore errors cannot be ruled out and the need for formal methods ensues.

Formal methods can guarantee the correctness of both hardware and software systems, with respect to a given specification. This is particularly pertinent when it comes to safety-critical systems such as vehicles, planes or banking networks which have to be correct under all circumstances.

Formal verification is a method of proving whether a system satisfies its formal specification. A formal specification is a mathematical description of a system and describes what the system should do (not how it should do it). These specifications can be written in temporal logic, like linear temporal logic (LTL) or computational tree logic (CTL), or in a specification language, like Z-notation or VDM-SL. In verification, a typical work flow to obtain a correct system works as follows. First, the user writes a formal specification and then implements

the system. Secondly, the system is tested against the specification. If the system violates the specification, the user has to debug the system and has to fix the errors. In the next step, the corrected system is tested again. All steps except the first one are repeated, until the system finally satisfies its specification.

Formal synthesis allows the automatic creation of systems from formal specifications. These systems are correct by construction. In verification, the user has to write the specification and the implementation for the system. In synthesis, the user only has to write the specification and the synthesis process automatically derives a correct implementation from the specification.

In 1962, the idea of property synthesis was proposed by Church [Chu62]. Recently there has been a lot of progress in making property synthesis practicable. Nevertheless, there are still a lot of open problems regarding the synthesis process, such as the high complexity of the synthesis algorithm and the size of the systems generated. Another problem is that synthesized systems often do not behave reasonably in unexpected situations. This problem is called the robustness problem, and is addressed in this work.

1.1.2 Motivation for Robust Systems

In this work we consider the synthesis of reactive hardware systems. Reactive systems are constantly interacting with their environment. First the environment provides some input. Then the system reads these input values, performs some internal calculation and provides a proper output to the environment. This process is repeated over and over again.

Specifications of systems describe how the system should respond to the inputs of the environment. In many cases, these specifications consists of environment assumptions and on system guarantees. Environment assumptions restrict the behavior of the environment. Since synthesized systems are correct by construction, they have to behave as prescribed, if the environment does not violate any assumption. The question is what happens if the environment does violate an assumption? In this case, what the synthesized system has to do has not been defined and no matter how it responds to further inputs, it is still a correct system.

For instance, let's take a look at the specification of an airport ground control system [Dav90]. The specification consists of one assumption and one guarantee. If the assumption is fulfilled, than a correct system has to satisfy the guarantee. The assumption requires, that there are less than or equal to 100 requests per second. The guarantee requires, that the system handles all plane requests within 0.1 seconds. If during operation a situation occurs, where 101 requests per second arrive at the ground control, than the assumption is violated. In this case, a correct system do not have to satisfy his guarantee and can behave unpredictably. There are various possibilities, as to how a system could react. It could stop responding to any requests at all, ignore all requests after the 100th one, or respond to all 101 requests, violating the 0.1 second response-time constraint for the 101th request. In this example the system's most preferable behavior in the case of an environmental error would be the third one. It is definitely not desirable that the system ignores aircraft, not even one.

Another example is the specification of a cash-operated beverage vending machine. The specification consists of two environment assumptions and one system guarantee. First we assume, that the customer is only allowed to enter his choice of beverage after supplying the machine with cash. Secondly, we assume that after the customer supplied the money, he chooses only one beverage. The guarantee requires, that after a system receives a choice, it delivers the beverage to the customer. As in the ground control example, not all possible environment behaviors are covered with the specification. What happens, when the costumer enters his choice of beverage, but did not enter any money. In this case, the first assumption is violated. The costumer can violate also the second assumption by entering more than one choice of beverage. In both cases, the system is allowed to behave arbitrary. It could terminate and stop any further processing, ignore the input and wait for a valid one, dispense multiple beverages at once, or deliver all it's available beverages.

Each of these solutions is a correct way for the system to handle the situation, but some solutions are better than others. It's not in the interest of the owner of the vending machine, if the machine stops working every time someone presses two buttons, or neither that the vending machine gives away free beverages. So even in non-security relevant systems we need our systems to be robust against environment errors. Furthermore we cannot expect the user to specify every possible environment error that could occur. Nevertheless, the synthesized system should be able to recover if the environment does.

In this context a system is defined as robust if it behaves reasonably, even under circumstances which have not been anticipated in the requirements specification [AS10]. Environment errors can always occur, throw transmission errors, a faulty environment or radiation related bit flips. The latter issue is becoming more serious, due to continuously decreasing feature sizes [SKK+02]. Therefore automatically synthesized systems should always be robust in order to be prepared for environment errors.

This thesis presents a new way of synthesizing robust systems.

1.2 Property Synthesis

Before we deal with robust synthesis, this section gives a brief overview of the characteristics of synthesis and it's history.

Property synthesis is the process of deriving a system from a high level behavioral specification. The derived system is guaranteed to satisfy the specification. This property of synthesis is called *correctness by construction*. Moreover, when the synthesis algorithm does not find a system for a particular specification, then the specification is unrealizable, and developers know at an early stage that their specification is likely to be faulty.

The behavioral specification aims to describe only the functionality of the system, so what the system has to do during its operation time, and not how it should do it. One of the main advantages of property synthesis is that specifications are generally shorter compared to a concrete implementation. This makes behavioral specifications less error-prone and easier to write and change. Thereby, systems can be simulated faster, and so property synthesis is

widely-used in prototyping. There are still a lot of open problems regarding synthesis, such as automatically synthesized systems are much bigger than systems made by hand. Thus, synthesis is still rarely used in practice. Recently, there has been a lot of work put into making synthesis more practicable.

1.2.1 History of Synthesis

In 1962, Alonzo Church [Chu62] first mentioned the problem of finding an implementation for a given specification. This problem is nowadays called the *Synthesis Problem* or *Church's Problem* and is defined as follows:
Given is a set of Boolean input variables I, a set of Boolean output variables O and a relation $R \subseteq (2^I)^\omega \times (2^I)^\omega$. The task is to find a function $f : (2^I)^* \to 2^O$, such that for all possible assignments of the input variables $x = x_0x_1x_2 \cdots \in (2^I)^\omega$ the function f generates an assignment of the output variables $y = y_0y_1y_2 \cdots \in (2^O)^\omega$ with $y_i = f(x_0x_1 \ldots x_{i-1})$ for $\forall i > 0$ and $(x, y) \in R$ holds.
R was originally defined in *restricted recursive arithmetic* (S1S) and defines pairs of permitted input and output sequences. The function f is called a strategy and maps every possible input sequence into a correct output sequence. The relation R can be viewed as a linear specification. The realizability problem is to decide if there exists such a strategy f for a given specification R.

In the following years the Church Synthesis Problem was theoretically solved in two different ways.
Büchi and Landweber [BL69] solved Church's problem based on infinite games. The specification $R(x, y)$ is given in Monadic second-order logic of order (MSO). The Büchi-Landweber Theorem says that for every MSO formula $R(x, y)$, it is decidable whether there is a function f which implements R, and f is computable from R. There is also a game version of the Büchi-Landweber Theorem. Büchi and Landweber transformed a given MSO formula $R(x, y)$ in a two player Game G_R. The theorem says that one of the players always has a winning strategy, and it is decidable which one has it. Also, for every MSO specification $R(x, y)$, there exists an algorithm that constructs a finite-state winning strategy f for the winner in G_R.

Rabin [Rab72] independently presented an alternative solution of Church's Problem, based on tree automata. Even if we have a linear-time specification, we have to consider all of the environment's possible behaviors. This branching-time behavior can be described by a tree. The edges of the input tree are labeled with all possible input values chosen from the environment and the nodes are labeled with output values chosen by the system. A tree automaton is able to check if all paths of a tree are satisfying the specification. Finally, Rabin reduced the realizability problem to the *emptiness problem of tree automata*.

In the primary studies of synthesis, specifications were given as S1S-formulas. To write specifications for reactive systems in S1S is cumbersome, therefore the need for new specification languages arose. In 1977, Pnueli proposed the idea of using temporal logics for specifications of non-terminating programs and introduced *Linear Temporal Logic* (LTL)[Pnu77]. LTL is nowadays widely used and has been further studied [Pnu86, MP79, MP81, MP92, MP95]. In 1981, Clarke and Emerson [CE81] introduced the principle of *model checking* and a branching

time logic called *Computation Tree Logic* (CTL). Through the application of model checking, linear temporal logics and branching time logics became very famous. Both, CTL and LTL have their respective pros and cons. Through the restricted syntax of CTL, writing specifications in LTL is easier than in CTL. However, while we can efficiently model check CTL specifications, LTL model checking is in theory PSPACE-complete [SC85] and takes in praxis exponential time and space in the size of the formula.

In computer science, we distinguish between open and closed systems. In the closed setting, we do not distinguish between input and output variables. The behavior of a closed system is completely determined by the internal state of the system. Here, we can think of the environment as a friend who cooperates with the system to find the correct output. In contrast, an open system (also called reactive system) interacts with its environment by reading input variables and setting output variables. In the open setting, we see the environment as an enemy, who tries to force the system to violate its specification and to produce incorrect output.

Earlier works concentrated on synthesis of closed systems. Emerson and Clarke [EC82] and Manna and Wolper [MW84] reduced the synthesis problem to the satisfiability problem. Let's suppose the specification is satisfiable. In this case, a system that meets the specification is constructed by using the proof that the specification is satisfiable. It was shown in [PR89] that this approach is only capable of synthesizing closed systems. Therefore, later works on synthesis considered open systems.

In 1989, Pnueli and Rosner [PR89] introduced the classical method of synthesizing open systems. Three years later, Rosner proved that LTL synthesis is 2EXPTIME-complete[Ros92]. Since the synthesis problem and the realizablilty problem are closely related, the realizability problem for full LTL is also 2EXPTIME-complete.
In the first step of the classical method of synthesis, the LTL specification is transformed into a non-deterministic Büchi word automaton. Ways of performing this step were proposed by Vardi et al. [VW86] and by Emerson et al. [ES84]. The language of the constructed Büchi automaton is the set of words that satisfy the specification. This translation leads to an exponential blowup of the size of the Büchi automaton in respect to the size of the LTL specification. In the second step, the Büchi automaton is determinised into a deterministic Rabin word automaton. Again, this results in an exponential blowup of the state space. Finally, the deterministic Rabin word automaton is translated into a nondeterministic Rabin tree automaton, which is checked for emptiness. If the Rabin tree automaton is empty, then the specification is not realizable. If not, than a system which implements the specification can be constructed from the witness of non emptiness. Although LTL synthesis was solved in theory, the high complexity discouraged many practitioners and no work on full LTL synthesis was published for a long time.

In the following years, the translation from LTL to Büchi word automaton improved a great deal [SB00, GO01, DGV99, GPV$^+$95]. Less progress was made in the determinisation step. The most popular way for determining Büchi automata is via Safra's construction. In 1988, this method was proposed by Safra in [Saf88] and was the first method which was asymptotically optimal. Unfortunately, this approach has two problems. First, it is very difficult to implement Safra's construction efficiently [KB05]. Second, Safra's construction

creates a very complex state space and there is no known symbolic data structure that is able to handle such state spaces [Mor10].

In 2005, Kupferman and Vardi [KV05] presented a new approach that avoided Safra's construction. They translated the non-deterministic Büchi word automaton into a generalized universal co-Büchi tree automata and then transformed this automata into a nondeterministic Büchi tree automata. Although this construction has the same complexity as Safra's construction, it is much simpler to implement and to optimize. Further, this approach can be implemented symbolically and is therefore much more practicable.

Nowadays, some tools exist for full LTL synthesis like LILY [JB06], ACACIA [FJR09] and UNBEAST [Ehl11b]. All these tools implement efficient algorithms, to make LTL synthesis more practical. In 2006, Jobstmann and Bloem presented the first tool for full LTL synthesis called LILY. LILY implements the Safraless approach from Kupferman and Vardi with additional optimizations for all intermediate automata, in order to achieve a better performance. Nevertheless, since the tool is implemented explicitly, LILY is only able to synthesize small examples [Ehl11a]. In 2009, Filiot et al. proposed a new symbolic antichain synthesis algorithm [FJR09] based on the principle of bounded synthesis [SF07]. The algorithm was implemented in the tool ACACIA [FJR09] and reduces the LTL synthesis problem to safety games. In practice, the tool outperforms LILY and performs similar to the tool UNBEAST. The main advantage of ACACIA is that in general it creates small winning strategies. In 2010, Filiot et al. presented a new version of ACACIA [FJR10]. The new version implements a compositional game solving approach for large conjunctions of LTL formulas [Ehl11a]. In 2011, Ehlers presented the tool UNBEAST [Ehl11b] that implements the bounded synthesis ideas of [SF07] and [Ehl10]. Unlike ACACIA, UNBEAST uses BDDs instead of anti-chains.

Another stand of research tried to find interesting subsets of LTL with lower complexity. It is possible to solve the synthesis problem in polynomial time, if the specification is restricted to a certain subsets of LTL. Asarin et al. [AMP94] presented polynomial solutions for the LTL formulas $\mathsf{G}\,\varphi$, $\mathsf{F}\,\varphi$, $\mathsf{G}\,\mathsf{F}\,\varphi$ and $\mathsf{F}\,\mathsf{G}\,\varphi$. More recently in 2004, Alur et al. [AT04] achieved the same for formulas consisting of Boolean combinations of formulas of the form $\mathsf{G}\,\varphi$. In 2003, Wallmeier et al. [WHT03] presented an efficient algorithm with only exponential complexity for the LTL fragment of request-response specifications. A request-response specification is of the form $\bigwedge_i \mathsf{G}(\varphi_i \rightarrow \mathsf{F}\,\psi_i)$.

Piterman, Pnueli and Sa'ar [PPS06] considered the synthesis problem for *Generalized Reactivity of rank 1* formulas (GR(1)), see Section 2.5.3. A GR(1) formula is of the form $\bigwedge_i \mathsf{G}\,\mathsf{F}\,\varphi_i \rightarrow \bigwedge_j \mathsf{G}\,\mathsf{F}\,\psi_j$. The left side of the implication consists of a set of *environment assumption*, and the right side of a set of *system guarantees*. A system which implements the specification has to fulfill all guarantees infinitely often, if all environment assumptions are fulfilled infinitely often. Most specifications can be transformed into GR(1) formulas [BGJ+07b, BGJ+07a]. However, it is not possible to formulate all LTL properties in GR(1), such as strong liveness properties ($\bigwedge_i (\mathsf{G}\,\mathsf{F}\,\varphi_i \rightarrow \mathsf{G}\,\mathsf{F}\,\psi_i)$). The symbolic algorithm presented in [PPS06] solves GR(1) specifications in N^3-time, where N is the size of the state space of the design. This algorithm was implemented in the tools Anzu [JGWB07], RAT [BCP+07] and its successor RATSY [BCG+10b].

1.3 Our Method of Synthesizing Robust Systems

This Section gives an overview of our algorithm to synthesize robust reactive modules from formal specifications.

Specifications consist of a set of logical temporal properties. For these properties we distinguish between safety and liveness properties. Alpern and Schneider [AS85] presented the first definition for both safety and liveness properties. Informally, safety properties state that *"something bad "* will not happen during system execution and liveness properties state that eventually *"something good "* must happen during system execution. Formally, safety and liveness properties are defined as follows:
A system execution π can be modeled as an infinite sequence of states: $\pi = s_0, s_1 \ldots$. A partial execution π_i consists of the first i states of an execution π. A property is a safety property, if and only if any execution π violating the property contains a partial execution π_i and all of whose infinite extensions violate the property. A property is a liveness property, if and only if any partial execution π_i can be extended to an infinite sequence π satisfying the property. For further details, see Section 2.5.2.

In this work, we consider specifications written in *Generalized Reactivity of rank 1* (GR(1), [PPS06]). GR(1) specifications are defined as follows:

$$\varphi = A \rightarrow G = A^i \wedge \bigwedge_i A_i^t \wedge \bigwedge_j A_j^l \rightarrow G^i \wedge \bigwedge_m G_m^t \wedge \bigwedge_n G_n^l. \tag{1.1}$$

A GR(1) specification describes the interaction between an environment and a system and consists of assumptions A and guarantees G. Assumptions and guarantees define the allowed actions of the environment and of the system, respectively. The formulas A^i and G^i define the initial states of the environment and the system, respectively. The formulas A^t describe the transition relation for the environment. These formulas define for all time steps the possible next values for the input variables, depending on the current input and output values. The formulas G^t describe the transition relation for the system. There is a small difference between the formulas A^t and G^t. In all time steps, the next values of the output variables depend on the current input and output values and on the next input values. The initial formulas and the transition formulas form the safety component of the GR(1) specification. The formulas A^l and G^l form the liveness component of the specification. They are of the form $\mathsf{G}\,\mathsf{F}\,p$, where p is some Boolean formula that has to be fulfilled infinitely often. For a formal definition of GR(1), see Section 2.5.3.

According to the definition of GR(1), guarantees must be fulfilled only if all assumptions are satisfied. At any point of time during operation, if the environment chooses a single invalid input value that violates a safety assumption the system can behave arbitrarily. Even if the environment works correctly for the rest of the time, the system do not have to fulfill any guarantees. Clearly, if assumptions are violated, the system may not be able to fulfill all guarantees. However, it should try to recover, if the environment does. Unfortunately, synthesized systems sometimes stop performing any useful interaction once a assumption has been violated. This is clearly not what anyone would expect from a robust system.

In order to define robustness, we introduce two kinds of errors, environment failures and system failures. The environment causes an **environment failure** if it chooses an input values that violates at least one safety assumption. The system causes a **system failure**, if it chooses an output values that violates at least one safety guarantee. *We define a system to be robust if a finite number of environment failures only induce a finite number of system failures* [BGHJ09]. Systems synthesized with our method fulfill this robustness criterion.

Our robust synthesis algorithm turns a GR(1) specification into a two-pair Streett game. The first Streett pair is called the *Correctness Pair* and the second is called the *Robustness Pair*. In the first step of the algorithm the specification in GR(1) is turned into a one-pair Streett game such that a winning strategy corresponds to a correct implementation [BCG$^+$10a]. The Streett pair of this game is the Correctness Pair. In the second step we add the Robustness Pair so that the strategy corresponds to a robust system. Finally we solve the game with the algorithm of [PP06].

The transformation from a GR(1) Specification into a one-pair Streett game is done by applying a *counting construction* [BCG$^+$10a]. The safety properties are encoded directly into the transition relation of the Streett game. The liveness properties are expressed via the *Correctness Pair*. For m liveness assumptions $\mathsf{G}\,\mathsf{F}\,A_i$ (with $1 \leq i \leq m$) and n liveness guarantees $\mathsf{G}\,\mathsf{F}\,G_j$ (with $1 \leq j \leq n$), the state-space is extended with two counters $x \in \{0, \dots m\}$ and $y \in \{0, \dots n\}$, which can be encoded with $\lceil \log_2(m+1) \rceil + \lceil \log_2(n+1) \rceil$ additional bits. The counter x is incremented modulo $m+1$ whenever assumption A_x (corresponding to the current counter value) is satisfied; similarly for y, G_y, modulo $n+1$. If a counter has the special value 0, it is always incremented. The counter value $x = 0$ indicates that all A_i have been satisfied; $y = 0$ indicates the same for all G_j. Hence, the condition $(\mathsf{G}\,\mathsf{F}\,x = 0) \rightarrow (\mathsf{G}\,\mathsf{F}\,y = 0)$, expressed by the Correctness-Pair $\langle (x = 0), (y = 0) \rangle$, ensures that the liveness part of the specification is encoded properly in the game. A winning strategy for this game corresponds to a correct implementation.

The second step of the algorithm is to add the *Robustness Pair*. In order to do that, we extend the state-space of the Streett game by two additional Boolean variables env_{err} and sys_{err}. Variable env_{err} is set to true whenever the environment produces an environment failure, sys_{err} is set to true iff the system produces a system failure. Initially, both env_{err} and sys_{err} are set to false. Our notion of robustness can now be formulated using the condition $(\mathsf{G}\,\mathsf{F}\,sys_{err}) \rightarrow (\mathsf{G}\,\mathsf{F}\,env_{err})$, which is expressed by the Robustness Pair $\langle (sys_{err}), (env_{err}) \rangle$. An infinite number of system errors is only allowed if there is an infinite number of environment errors.

A winning strategy for the two-pair Streett game corresponds to a correct and robust implementation. We use a recursive fixpoint algorithm to compute the winning region [PP06]. Intermediate results of this computation can be used to obtain the winning strategy.

1.4 Structure of this work

This work is based on the paper *Synthesizing Robust Systems with* RATSY at the Synth 2012 Workshop in Berkeley, USA and is organized as follows.

Chapter 2 introduces the mathematical background for understanding this work and gives the necessary definitions and notations used in the following chapters. Section 2.7 gives a short introduction on the tools RAT, RATSY and Marduk and gives a brief overview of their software design.
The last section of this chapter, Section 2.8 summarizes related work on robust synthesis.

Chapter 3 covers the core part of this work. First, Section 3.1 presents the idea of our algorithm for synthesizing a robust system. This leads us to the definition of robustness presented in Section 3.2. Section 3.3 uses an example to illustrate, what can happen if a system is not robust. Section 3.4 explains on an example system failures and environment failures. Next follows a detailed description of the algorithm in Section 3.5. The subsections contain the single steps of the algorithm and provide a detailed explanation. 3.5.1 3.5.2 and 3.5.3 explain the steps that are necessary to transform the specification into two-pair Streett game. Section 3.5.4 and Section 3.5.5 explain the computation of a winning region and winning strategy for two-pair Streett games in more detail. Section 3.6 of this chapter, the algorithm is applied to an example to demonstrate how it works. The last Section 3.7 discusses, how much time steps the system needs to recover from an environment failure.

Chapter 4 provides details of the implementation of robustness in RATSY and presents experimental results. Chapter 5 concludes by discussing the most important facts and gives an outlook on future work.

2 Preliminaries

2.1 Kripke Structures

The Kripke Structure was introduced by Saul Kripke in 1963 [Kri63] and is used in model checking to model the behavior of a reactive hardware and software. The behavior of a system is often modeled through temporal properties. *Temporal logics* are most commonly used for the formal specification of such properties and are interpreted traditionally in terms of Kripke Structures.

Kripke Structures allow the representation of a reactive system as a set of all possible system states and transitions between the states. States are labeled with *atomic propositions*. An atomic proposition is a Boolean expression over system variables and predicates, and represents a relationship among variables. All valid transitions between system states are defined by the transition relation.

Definition 2.1. (Kripke Structure)
Let AP be the set of all atomic propositions. Clarke et al. [CGP01] defined a Kripke Structure M over AP as a 4-tuple $M = (S, S_0, R, L)$, when

- S is a finite set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a left-total transition relation, meaning $\forall s \in S \; \exists s' \in S : (s, s') \in R$, and
- $L : S \to 2^{AP}$ is a labeling function, which labels a state $s \in S$ with a subset of AP, the atomic propositions that are valid in s.

A **path** π is an infinite sequence of states: $\pi = s_0, s_1, s_2 \ldots$ with $s_0 \in S_0$ and $\forall i \geq 0 : (s_i, s_{i+1}) \in R$. Each path π defines a corresponding infinite **word** $\bar{\sigma}$ over the alphabet 2^{AP}, $\bar{\sigma} = \sigma_0, \sigma_1, \sigma_2, \ldots$ with $\forall i \geq 0 : \sigma_i = L(s_i)$.

Definition 2.2. (State Transition Graph)
A Kripke structure $M = (S, S_0, R, L)$ can be represented as a total *state transition graph*, whose nodes represent the states in S and whose edges represent the transitions in R. The nodes of the graph are labeled according to the labeling function L.

Consider an example of a state transition graph of a Kripke structure.

Example 2.1. Let $M_1 = (S, S_0, R, L)$ be a Kripke structure. The set AP consists of the atomic propositions a, b and c. M_1 is defined by:

- $S = \{s_0, s_1, s_2\}$.
- $S_0 = \{s_0\}$.
- $R = \{(s_0, s_0), (s_0, s_1), (s_0, s_2)(s_1, s_2), (s_2, s_2)\}$.
- $L : \{(s_0, \{a\}), (s_1, \{b\}), (s_0, \{a, c\})\}$.

Figure 2.1 shows the state transition graph of the Kripke structure.

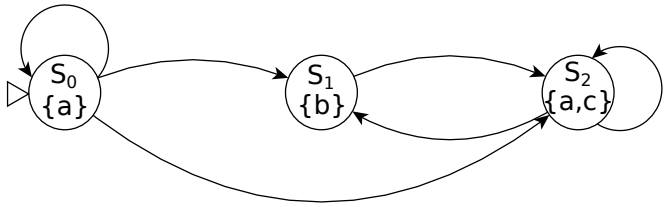


Figure 2.1: Kripke Structure Example.

One possible path of M_1 would be $\pi = s_0, s_0, s_1, s_2, s_2, ...$ This path would generate the word $\bar{\sigma} = \{a\}\{a\}\{b\}\{a, c\}\{a, c\}...$

2.1.1 Computation Trees

Unwinding the transition relation graph of the Kripke structure leads to a *computation tree*. A computation tree defines the set of all possible behaviors of a Kripke structure. This means that the branches of the tree represent all possible paths π of the corresponding Kripke structure.

Definition 2.3. (Computation trees). Let $M = (S, S_0, R, L)$ be a Kripke structure over AP. The computation tree for M is defined as follows:

- The nodes of the tree are labeled with states $s \in S$.
- The root node is labeled with a sate $s_0 \in S_0$.
- If a node in the tree is labeled with $s \in S$, then its children are all nodes labeled with $s' \in S$ for which holds that $(s, s') \in R$.

Figure 2.2 shows the computation tree for the Kripke Structure M_1 of example 2.1.

Since all possible behaviors of a Kripke structure are contained in its computation tree, it is possible to formulate properties of Kripke structures via properties of paths in computation trees and states along these paths using temporal logic. E.g. a property of a Kripke Structure could be: “There are no states where a and b hold at the same time ”. We can express this property in the tree with: “For every path and for all nodes on this path, a and b never hold at the same time ”.

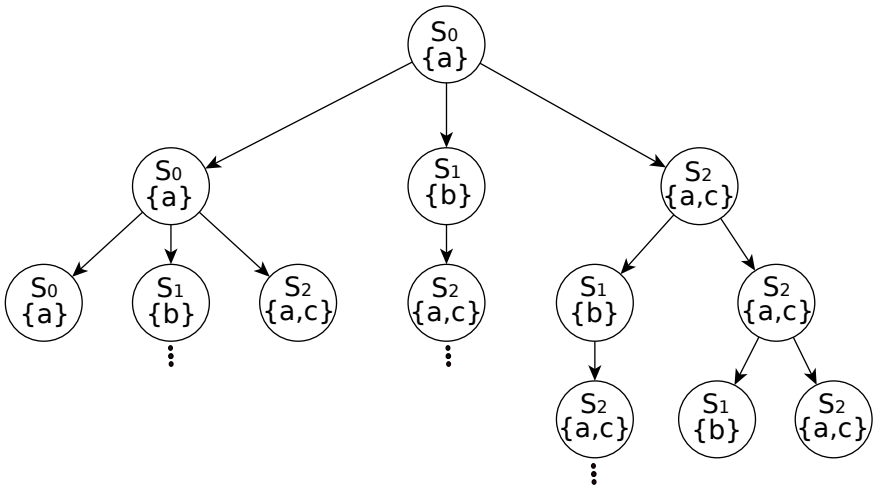


Figure 2.2: Computation Tree Example.

2.2 Labeled Transition System

Besides Kripke structures, *Labeled Transition Systems* (LTS) [Pnu77], [Kel76] are also used to model the behavior of reactive systems. Unlike Kripke structures, in LTS transitions are labeled, not states. Also Labeled Transition Systems allow states without outgoing transitions, so called *deadlocked states*. While Kripke structures are more convenient for defining temporal logics, transitions systems are better suited for modeling reactive systems.

Also, labeled transition systems differ from finite state automata (defined in 2.3). The main difference between LTS and automata is that the set of states and the set of labels may be infinite in a LTS. Also a state in a LTS may have an infinite branching. An automaton has a set of final states (also called accepting states), a LTS doesn't.

Definition 2.4. (Labeled Transition System)
A Labeled Transition System T is defined over a set of actions Act as a 4-tuple $T = (Act, S, s_0, R)$, when

- Act is a set of transition labels, so called *actions*.
- S is a finite set of states.
- $s_0 \in S$ is the initial state.
- $R \subseteq S \times Act \times S$ is a transition relation.

S and Act are either finite or countably infinite.

A path π of T is defined as sequence a of states and actions: $\pi = s_0, a_0, s_1, a_1, s_2...$ with $\forall i \geq 0 : s_i \in S$, $a_i \in Act$ and $(s_i, a_i, s_{i+1}) \in R$.

The graph of an LTS is called *labeled transition relation graph*.

Example 2.2. (LTS)
Consider an example of a Labeled Transition System that models a light switch. Let $T_1 = (Act, S, s_0, R)$ be a LTS defined by:

- $Act = \{\text{off,on}\}$
- $S = \{\text{dark,light}\}$.
- $S_0 = \{\text{dark}\}$.
- $R = \{(\text{dark,on,light}), (\text{light,off,dark})$.

Figure 2.3 shows the state transition graph of the LTS.

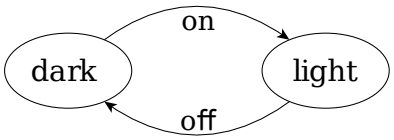


Figure 2.3: Labeled Transition System Example.

2.3 Automaton

An automaton is a mathematical construct that takes an input word and decides if it is accepted or not. Automata are used to model systems. Since in reality, systems differ greatly from each other, there are also many different types of automata. Automata can be classified by their *input*, (finite or an infinite sequence of symbols or trees of symbols), their number of *states* (finite or an infinite), their *transition function* (deterministic, nondeterministic or alternating) and their *acceptance condition*.

In this work, we will discuss finite state infinite word automata, so called ω-automata. Since the basic for ω- automata are automata with finite input, we will start with them.

For further information about automata see [Tho96].

2.3.1 Words and Languages

Within the context of automata we use the following terminology:

- The **alphabet** Σ is a finite set of letters.
- A **word** $\bar{\sigma}$ over an alphabet Σ is defined as a sequence of letters. A word can be finite: $\bar{\sigma} = \sigma_0, \sigma_1 \ldots \sigma_n$ or infinite: $\bar{\sigma} = \sigma_0, \sigma_1, \sigma_2 \ldots$ with $\forall i \geq 0 : \sigma_i \in \Sigma$.
  - ε denotes the empty word.
  - We define Σ^n to be the set of words over Σ of length n.

  - The set of finite words over Σ is defined by $\Sigma^\star = \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \ldots$.
  - Σ^ω is the set of all infinite words over Σ.
  - $\Sigma^\infty = \Sigma^\star \cup \Sigma^\omega$ denotes the set of all finite and infinite words over Σ.
- A **language** L is a set of words from $\Sigma^\star$, Σ^ω or Σ^∞.

2.3.2 Finite Automata on Finite Words

First we define finite automata that accept finite input words $\bar{\sigma} = \sigma_1, \sigma_2, \sigma_3 ... \sigma_n \in \Sigma^\star$.

Definition 2.5. (Finite Automata on Finite Words)
Automaton are defined over a 5-tuple $A = (S, \Sigma, \Delta \text{ or } \delta, s_0, F)$.

- S is a finite set of states.
- Σ is a finite alphabet.
- $\Delta \subseteq S \times \Sigma \times S$ is a non-deterministic transition relation.
- $s_0 \in S$ is an initial state.
- $F \subseteq S$ is a set of accepting states.

Automata can be deterministic or non-deterministic, wherein a deterministic automaton is a special case of a non-deterministic automaton. In the case of a non-deterministic automaton we use the transition relation Δ.

Definition 2.6. (Deterministic Automata)
An automaton is called deterministic if

$$\forall s \in S : \forall \sigma \in \Sigma : \left|\{s' \in S : (s, \sigma, s') \in \Delta\}\right| \leq 1. \tag{2.1}$$

A deterministic transition function specifies only one possible transition for each state and letter. In this case, we use the transition function $\delta : S \times \Sigma \to S$.

An automaton with finite input always starts in an initial state, and **runs** on a finite input word $\bar{\sigma}$. At any time step, an automaton is in a current state s_i and reads an input letter σ_i. The transition function takes s_i and σ_i and determines the next state s_{i+1}. The finite sequence of visited states is called a run on $\bar{\sigma}$. If there exists a run on $\bar{\sigma}$ that ends in an **accepting state**, the automaton accepts the word $\bar{\sigma}$, otherwise the word is rejected. The set of all accepted words is called the **language** of the automaton.

Definition 2.7. (Complete Automata)
An automaton is called *complete*, if

$$\forall s \in S : \forall \sigma \in \Sigma : \left|\{s' \in S : (s, \sigma, s') \in \Delta\}\right| \geq 1. \tag{2.2}$$

In this case, the transition function Δ is complete.

Definition 2.8. (Runs on Automata)
Let $\bar{\sigma} = \sigma_1, \sigma_2, \sigma_3 ... \sigma_n \in \Sigma^\star$ be a word and let $A = (S, \Sigma, \Delta, s_0, F)$ be an automaton. A run π of A on $\bar{\sigma}$ is a finite sequence of states $\pi = s_0, s_1 \dots s_n$ such that $\forall i \geq 0 : (s_i, \sigma_{i+1}, s_{i+1}) \in \Delta$.

While a deterministic automaton has only one possible run for a certain input word $\bar{\sigma}$, several runs of $\bar{\sigma}$ can exist on a non-deterministic automaton.

A word $\bar{\sigma}$ is an **accepted word** of the automaton A if there exists a run π of A on $\bar{\sigma}$ with last state s_n and $s_n \in F$. In that case, we say π is an accepting run. The set of all accepted words defines the **language** L of an automaton A :

$$L(A) = \{\bar{\sigma} \in \Sigma^\star : \bar{\sigma} \text{ is accepted by A}\}. \tag{2.3}$$

2.3.3 Finite ω-Automata

Finite ω-automata accept infinitely long input words $\bar{\sigma} = \sigma_1, \sigma_2, \sigma_3 ... \in \Sigma^\omega$ and are very important for modeling reactive systems. A reactive system is in constant interaction with its environment. This leads to an infinite sequence of input letters, and hence we need automata that can deal with such input.

Definition 2.9. (ω-Automata)
ω-Automaton are defined over a 5-tuple $A = (S, \Sigma, \Delta, s_0, Acc)$ [Tho96].

- S is a finite set of states.
- Σ is a finite alphabet.
- $\Delta \subseteq S \times \Sigma \times S$ is a non-deterministic transition relation or
  $\delta : S \times \Sigma \to S$ is a deterministic transition relation.
- $s_0 \in S$ is an initial state.
- $Acc : S^\omega \to \{\mathsf{true}, \mathsf{false}\}$ is an acceptance condition.

Since ω-automata deal with infinitely long words, runs on ω-automata don't have final states. Thus, acceptance conditions are defined over states that have been visited infinitely often during a run π.

The ω-language $L(A)$ of an ω-automaton A defines the set of ω-words accepted by the automaton.

Definition 2.10. (Infinity Set of a Run π)
Let $\pi = s_0, s_1, s_2 \dots \in S^\omega$ then we define:

$$\mathsf{inf}(\pi) = \{s \in S : \text{ for infinitely many } i \geq 0 : s_i = s\}. \tag{2.4}$$

So the infinity set $\mathsf{inf}(\pi) \subseteq S$ is the set of states occurring infinitely often in π. Since runs on ω- automata are infinite, this set is never empty. If $\mathsf{inf}(\pi)$ satisfies the winning condition Acc, than the run is an *accepting run*, otherwise it's a *rejecting run*.

Definition 2.11. (Runs on ω-Automata)
Let $\bar{\sigma} = \sigma_1, \sigma_2, \sigma_3 \cdots \in \Sigma^\omega$ be an **infinite** word.

- If $A = (S, \Sigma, \Delta, s_0, F)$ is a non-deterministic automaton, than a run π of a A on $\bar{\sigma}$ is an **infinite** sequence of states $\pi = s_0, s_1, s_2 \ldots$ with $(s_i, \sigma_{i+1}, s_{i+1}) \in \Delta$.
- If $A = (S, \Sigma, \delta, s_0, F)$ is a deterministic automaton, than a run π of a A on $\bar{\sigma}$ is an **infinite** sequence of states $\pi = s_0, s_1, s_2 \ldots$ with $s_{i+1} = \delta(s_i, \sigma_{i+1})$.

For all runs $\pi = s_0, s_1, s_2 \ldots$ holds that, $\forall i \geq 0 : s_i \in S$ and s_0 is the initial state.

We differ ω-automata only by their acceptance condition. There are several ways to define the acceptance condition. In the following, we discuss some commonly used acceptance condition: Büchi-, co-Büchi-, Muller-, Rabin-, Streett- and Parity- acceptance.

Rabin, Parity, Streett and Muller automata, each in their deterministic and non-deterministic form and non-deterministic Büchi automata are equally expressive. They all recognize the *regular ω-languages*. Only deterministic Büchi automata are strictly less expressive than the others. So there is no algorithm to transform Büchi automata in deterministic Büchi automata. To determinise Büchi automata they are transformed into deterministic Rabin or Muller automata via Safra's construction [Saf88] or via McNaughton's Theorem [McN66]. Safra's construction transforms a non-deterministic Büchi automaton with n states in an equivalent deterministic Rabin or Muller automaton with $2^{O(n*\log n)}$ states. Safra's construction is proven to be optimal for Rabin automata.

Büchi-Automata

In 1962, Büchi automata were invented by the mathematician Julius Richard Büchi [Büc62].

Definition 2.12. (Büchi Acceptance Condition)
The Büchi acceptance condition Acc is a finite set of states: $Acc = F \subseteq S$. A run π satisfies the acceptance condition, if $\mathsf{inf}(\pi) \cap F \neq \emptyset$.

Here, F is a set of accepting states. A run π is accepted if and only if at least one of the infinitely often occurring states in π is in F.

Generalized Büchi-Automata

Definition 2.13. (Generalized Büchi Acceptance Condition)
The Generalized Büchi acceptance condition Acc is defined by a set of sets of states: $Acc = F_1, F_2, \ldots, F_k$ with $F_i \subseteq S$ for $i = 1 \ldots k$. A run π satisfies the acceptance condition, if $\mathsf{inf}(\pi) \cap F_i \neq \emptyset$ for $i = 1 \ldots k$.

In this case a run π is accepted if and only if at least one state of each set F_i for $i = 1 \ldots k$ occurs infinitely often during π.

Co-Büchi-Automata

The co-Büchi acceptance is dual to the Büchi acceptance.

Definition 2.14. (Co-Büchi Acceptance Condition)
The co-Büchi acceptance condition Acc is a finite set of states: $Acc = F \subseteq S$. A run π satisfies the acceptance condition, if $\mathsf{inf}(\pi) \cap Acc = \emptyset$.

Here, F defines a set of rejecting states. This condition requires that no state from F is visited infinitely often during π.

Muller-Automata

Muller-Automata were named after its inventor David Eugene Muller [Mul63].

Definition 2.15. (Muller Acceptance Condition)
The Muller acceptance condition Acc is given as a set of sets of states $Acc = F_1, F_2, \ldots, F_k$ with $F_i \subseteq S$ for $i = 1 \ldots k$. A run π satisfies the acceptance condition, if $\mathsf{inf}(\pi) \in Acc$.

In Muller-automata the sets F_i form accepting sets. A run π is accepting, if exactly the set of all states which occur infinitely often in π is specified as an accepting set.

Rabin-Automata

In 1969, Michael Oser Rabin introduced another acceptance condition, and the corresponding ω-automaton was named Rabin automaton [Rab69].

Definition 2.16. (Rabin Acceptance Condition)
The Rabin acceptance condition Acc is given as a set of pairs of sets of states $Acc = (E_1, F_1), (E_n, F_n), \ldots, (E_n, F_k)$ with $E_i, F_i \subseteq S$ for $i = 1 \ldots k$. A run π satisfies the acceptance condition, iff $\exists (E_i, F_i) \in Acc : \mathsf{inf}(\pi) \cap E_i = \emptyset \wedge \mathsf{inf}(\pi) \cap F_i \neq \emptyset$.

Here, a run π is accepting, if there exists a pair (E_i, F_i) where E_i is visited finitely often and F_i infinitely often.

Streett-Automata

Robert Streett introduced Streett automata [Str82]. The dual automaton to a Streett automaton is a Rabin automaton and vice versa. The Streett acceptance condition also consists of pairs of sets of states but the pairs are interpreted complementary.

Definition 2.17. (Streett Acceptance Condition)
The Streett acceptance condition Acc is given as a set of pairs of sets of states $Acc = (E_1, F_1), (E_n, F_n), \ldots, (E_n, F_k)$ with $E_i, F_i \subseteq S$ for $i = 1 \ldots k$. A run π satisfies the acceptance condition, iff $\forall (E_i, F_i) \in Acc : \mathsf{inf}(\pi) \cap E_i \neq \emptyset \rightarrow \mathsf{inf}(\pi) \cap F_i \neq \emptyset$.

This means that a run π is accepting, if for all pairs (E_i, F_i) holds that if states from E_i are visited infinitely often, than states from F_i must be visited infinitely often as well.

Parity-Automata

The Parity acceptance condition is also a very frequently used acceptance condition [Mos84], [EJS93].

Definition 2.18. (Parity Acceptance Condition)
The Parity acceptance condition *Acc* is given as a coloring function $c : S \to \{0, \dots k\}$, where the numbers $\{0, \dots k\}$ represent colors. A run $\pi = s_0, s_1, s_2 \dots$ generates a sequence of colors $c(\pi) = c(s_0), c(s_1), c(s_2) \dots$. A run π satisfies the acceptance condition, if $max(\mathsf{inf}(c(\pi)))$ is even.

In Parity automata, a run π is accepting, if the maximal color occurring infinitely often is even.

2.3.4 Symbolic Representation of ω-Automata

For reasons of efficiency, ω-automata are often represented in a symbolic setting, for instance by using Binary Decision Diagrams (BDDs)[Lee59]. The symbolic representation of an automaton can be exponentially more compact than its explicit representation, especially when the state space is large. In the explicit representation, each state and each transition is represented individually. The goal of symbolic encoding is, to represent sets of states and transitions more efficiently. This is done, by using formulas to define sets of states and transitions. This formulas are called *Characteristic Formulas*. A Characteristic Formula of a set evaluates to true, if and only if the element is in the set.

For the symbolic representation of ω-automata, we encode the input alphabet Σ with a set of Boolean Variables $v_\Sigma = v_0, v_1, \dots, v_n$ with $n = \lceil \log_2(|\Sigma|) \rceil$. In order to write a Characteristic Formula for the transition function, we introduce two additional sets of propositional variables. The first set of Boolean variables $s = s_0, s_1, \dots, s_m$ with $m = \lceil \log_2(|S|) \rceil$ represents the current state and the second set $s' = s'_0, s'_1, \dots, s'_m$ represents the next state. Using the sets of variables v_Σ, s and s', we are able to write Characteristic Formulas for the initial state, the transition relation and the acceptance condition of the automaton [MSL08].

2.4 Mealy and Moore Machines

2.4.1 Reactive Systems

Our goal is to synthesize reactive systems [MP92]. Reactive systems are systems that do not terminate. As illustrated in Figure 2.4, a reactive system is in constant interaction with its environment. At any time, these systems receive input from some environment (via sensors,

machines or humans), perform some internal calculation to process the current input and send a corresponding output to the environment. A specification of a reactive system defines the allowed interaction between environment and system. Popular examples of reactive systems include cell phones, control systems like traffic-light controlers and elevator controlers or even nuclear reactors.

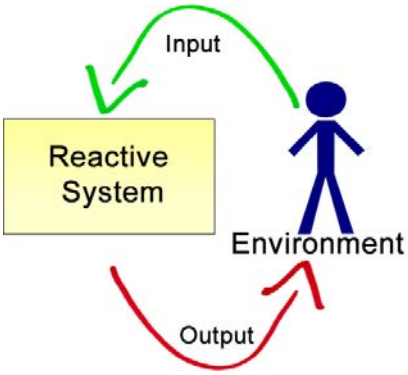


Figure 2.4: Reactive System.

2.4.2 Mealy and Moore Machines

We differentiate between two different types of reactive systems: *Mealy machines* and *Moore machines*. A Moore machine is a specialization of the Mealy machine. For every Mealy machine there exists an equivalent Moore machine.

Definition 2.19. (Moore and Mealy Machines)
Given a set of input variables X and a set of output variables Y. Mealy machines and Moore machines are defined over a 6-tuple $M = (S, D_X, D_Y, s_0, \delta, \lambda)$, when

- S is a finite set of states,
- $D_X = 2^X$ denotes the input alphabet, and $D_Y = 2^Y$ the output alphabet,
- $s_0 \in S$ defines the initial state,
- $\delta : S \times D_X \to S$ is the transition function, and
- $\lambda : S \times D_X \to D_Y$ is the output function for a Mealy Machine,
  $\lambda : S \to D_Y$ is the output function for a Moore Machine.

Moore and Mealy machines differ only in their output function. Mealy machines react instantaneously on input, therefore the next output is decided by the current state and the current input. On the other hand, Moore machines react one time step later. Their output is a function of the present state only.

To simplify further definitions, we extend the definition of the transition function:

$$\delta : S \times D_X^\star \to S, \text{ with } \delta(s, \epsilon) = s, \text{ and } \delta(s, aw) = \delta(\delta(s, a), w) \text{ for } a \in D_X, w \in D_X^\star. \quad (2.5)$$

A **path** π of a Moore or Mealy machine is defined as an infinite sequence of states: $\pi = s_0, s_1, s_2 \ldots$ such that for $\forall i \geq 0$ we have $s_{i+1} = \delta(s_i, x_i)$, $s_i \in S$, $x_i \in D_X$ and s_0 is the initial state. A **word** is an infinite sequence of tuples: $\bar{\sigma} = (x_0, y_0), (x_1, y_1), \ldots,$ where $\forall i : (x_i, y_i) \in D_X \times D_Y$.

- For a *Mealy machine* a word $\bar{\sigma}$ is defined as: $\bar{\sigma} = (x_0, \lambda(s_0, x_0))(x_1, \lambda(s_1, x_1)) \ldots$ with $s_i = \delta(s_{i-1}, x_{i-1})$ and $x_{i-1} \in D_X$ for $i > 0$.
- For a *Moore machine* a word σ is defined as: $\bar{\sigma} = (x_0, \lambda(s_0))(x_1, \lambda(s_1)) \ldots$ with $s_i = \delta(s_{i-1}, x_{i-1})$ and $x_{i-1} \in D_X$ for $i > 0$.

Example 2.3. Let's have a look at an example of a Mealy machine $M_1 = (S, D_X, D_Y, s_0, \delta, \lambda_1)$ with:

- $S = \{s_0, s_1, s_2\}$.
- $X = \{r\}$, $Y = \{a\}$, $D_X = \{0, 1\}$, $D_Y = \{0, 1\}$.
- The transition function δ is defined by Table 2.1, e.g., $\delta(s_0, 0) = s_1$, $\delta(s_0, 1) = s_2, \ldots$.
- The output function λ_1 is defined by the adjacent Table 2.2, e.g., $\lambda_1(s_0, 0) = 1$, $\lambda_1(s_0, 1) = 0, \ldots$.

Both transition relation δ and output function λ_1 of M_1 are represented graphically in Figure 2.5. The transitions are expressed as arrows and their captions contain the current input and output values.

Table 2.1: Mealy Machine M_1. Transition relation δ.

state	input	nextstate
s_0	0	s_1
s_0	1	s_2
s_1	0	s_1
s_1	1	s_2
s_2	0	s_1
s_2	1	s_2

Table 2.2: Mealy Machine M_1. Output function λ_1.

state	input	output
s_0	0	1
s_0	1	0
s_1	*	0
s_2	*	1

Example 2.4. Now, let us consider a Moore machine M_2, which differs from the Mealy machine M_1 in 2.3 only by its output function. The Moore machine $M_2 = (S, D_X, D_Y, s_0, \delta, \lambda_2)$ is defined by:

- $S = \{s_0, s_1, s_2\}$.
- $X = \{r\}$, $Y = \{a\}$, $D_X = \{0, 1\}$, $D_Y = \{0, 1\}$.
- The transition function δ is defined by Table 2.3.

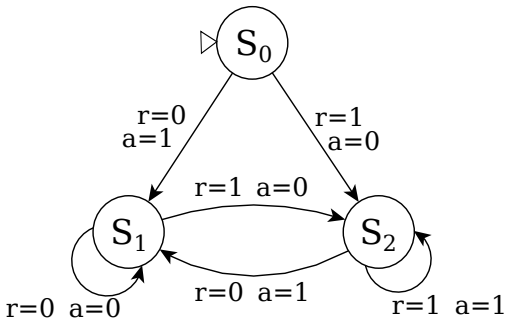


Figure 2.5: Graph of Mealy Machine M_1.

- The output function λ_2 is defined by the adjacent Table 2.4.

Both transition relation δ and output function λ_2 of M_2 are represented graphically in Figure 2.6.

Table 2.3: Moore Machine M_2.
Transition relation δ.

state	input	nextstate
s_0	0	s_1
s_0	1	s_2
s_1	0	s_1
s_1	1	s_2
s_2	0	s_1
s_2	1	s_2

Table 2.4: Moore Machine M_2.
Output function λ_2.

state	output
s_0	0
s_1	0
s_2	1

Note that the two machines are not equivalent, because they accept different sets of words. E.g. the first output of Moore Machine M_2 is always 0. On the other hand, the output function λ_1 of the Mealy Machine M_1 reacts instantaneously on the first input, and the first output can either be 0 or 1 depending on the first input.

2.5 Temporal Logics

We want to specify the behavior of reactive systems. Such behavior is always defined over paths (also called computation paths) or over computation trees of Kripke structures. A path π is an infinite sequence of states $\pi = s_0, s_1, s_2 \ldots$ such that s_0 is an initial State and we have for $\forall i \geq 0 : (s_i, s_{i+1}) \in R$ The computation tree of a Kripke structure contains all possible computation paths. With propositional logic or first-order logic it's is only possible to express properties of states, not properties of paths. Hence the need arose for temporal logics. Temporal logics allow the specification of properties of reactive systems over time, such as safety properties or liveness properties, see Section 2.5.2.

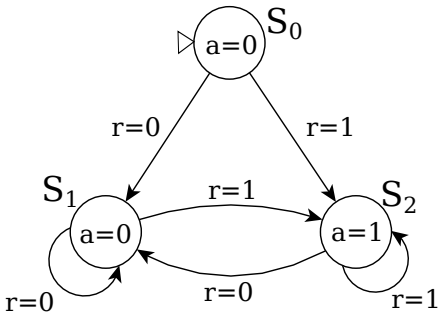


Figure 2.6: Graph of Moore Machine M_2.

In 1977, Pnueli proposed the idea of using temporal logics for the specification of non-terminating programs and introduced *Linear Temporal Logic* (LTL) [Pnu77]. LTL is currently one of the most popular temporal logics and was further studied by Manna and Pnueli in ([MP79], [MP81], [MP92], [MP95]). In [Pnu86], Pnueli introduced the application of temporal logic for the specification of reactive systems. Nowadays, LTL is used in many different applications. It is used as a specification language in model checking tools such as SPIN [Hol97] or SMV [McM93] and as a basis for other specification languages like PSL [EF06].

The second major family of temporal logic forms *branching-time logics*. Clarke and Emerson [CE81] and Queille and Sifakis [QS82] each independently introduced the principal of *model checking*. Through model checking, temporal logics became even more important and popular. Each of the papers mentioned used a different branching time logic. Clarke and Emerson introduced *Computation Tree Logic* (CTL), which became very famous. Queille and Sifakis used in their paper a branching time logic introduced by Lamport [Lam80]. In the 1980s, Wolper [Wol83] proposed efficient methods for model checking of LTL formulas, and Emerson and Halpern [EH85] for CTL formulas.

Because of the restricted syntax of CTL, writing specifications in LTL is easier than in CTL. However, LTL model checking is in theory PSPACE-complete [SC85] and takes in praxis exponential time and space in the size of the LTL formula, while we can efficiently model check CTL specifications.

Finally, it was shown that LTL and CTL have different expressive powers. This fact led to the introduction of the branching-time logic $CTL^\star$ [EH83], which is a superset of both CTL and LTL.

This Section gives an overview of commonly used temporal logics. Subsection 2.5.1 discusses LTL and Subsection 2.5.2 gives some examples of how properties of reactive systems can be expressed in LTL. In this work the specifications of reactive systems are written in *Generalized Reactivity of rank 1* called GR(1), which is a commonly used subset of LTL. An introduction of GR(1) can be found in Subsection 2.5.3. Subsection 2.5.4 gives a short introduction on CTL* and CTL. Another very important branching time temporal logic is called *modal μ-calculus*, which we define in Subsection 2.5.5.

2.5.1 LTL

Linear temporal logic (LTL) extends propositional logic by temporal operators [Pnu77]. and has attained an important role in the formal specification of reactive systems. LTL formulas are able to express properties over individual execution paths of systems, and thus it is very intuitive for describing the behavior of reactive systems with LTL formulas.

Before we define the syntax and the semantic of LTL, we are going to explain the meaning of the temporal operators in an informal fashion. A LTL formula is interpreted over an infinite sequence of states (called an execution path) $\pi = s_0, s_1, s_2, \ldots$ of Kripke structures. The two basic temporal operators of LTL are the *Next*-operator X and the *Until*-operator U. From these basic operators we can derive additional temporal operators like the *Global*-operator G, the *Eventually*-operator F and the *Weak-Until*-operator W. The meaning of these operators is explained below:

- The *Until* Operator U:
  $\phi \,\mathsf{U}\, \psi$ holds for a path π if ϕ holds in all states along π from s_0 until ψ holds.
- The *Next* Operator X:
  $\mathsf{X}\,\phi$ holds in a state s_i of path π if ϕ holds in the next state s_{i+1}.
- The *Globally* Operator G:
  $\mathsf{G}\,\phi$ holds for a path π if ϕ holds at **all** states on π.
- The *Eventually* Operator F:
  $\mathsf{F}\,\phi$ holds for a path π if ϕ holds at **some** states on π.
- The *Weak-Until* Operator W:
  $\varphi \,\mathsf{W}\, \psi$ holds for a path π if either ϕ holds until ψ holds or ϕ holds for all states along π.

Now we can define the syntax and the semantics of LTL [MP92].

Definition 2.20. (Syntax of LTL)

Let AP be a set of atomic propositions, M be a Kripke structure and $\pi = s_0, s_1, s_2, \ldots$ a path in M. The **syntax** of a LTL formula is defined as follows:

1. **Atomic Proposition:** $\forall p \in AP \cup \{\mathsf{true}, \mathsf{false}\}$ are LTL formula.
2. **Boolean Operators:** If φ and ψ are LTL formulas, then so are $(\neg\varphi)$, $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$.
3. **Temporal Operators:** If φ and ψ are LTL formulas, then so are $\mathsf{X}\,\varphi$ and $\varphi \,\mathsf{U}\, \psi$.

By using these elementary operators it is possible to derive the remaining operators.

- $\phi \rightarrow \psi = \neg\phi \vee \psi$
- $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$
- $\mathsf{F}\,\phi = \mathsf{true}\,\mathsf{U}\,\phi$

- $\mathsf{G}\,\varphi = \neg\,\mathsf{F}\,\neg\phi$
- $\phi\,\mathsf{W}\,\psi = (\phi\,\mathsf{U}\,\psi) \vee \mathsf{G}\,\phi$

Definition 2.21. (Semantic of LTL)

Let AP be a set of atomic propositions, M be a Kripke structure and $\pi = s_0 s_1 s_2 \ldots$ a path in M. ϕ and ψ are LTL formulas. We define $(\pi, i) \models \phi$ for the path π satisfies the formula ϕ at position i in state s_i. The satisfaction relation $\models$ is defined as follows:

1. **Atomic Propositions:**
   - $\phi \in AP \cup \{\mathsf{true}, \mathsf{false}\} : (\pi, i) \models \phi$ iff $s_i \models \phi$
2. **Boolean Operators:**
   - $(\pi, i) \models \neg\phi$ iff $(\pi, i) \not\models \phi$.
   - $(\pi, i) \models \phi \wedge \psi$ iff $(\pi, i) \models \phi$ and $(\pi, i) \models \psi$.
   - $(\pi, i) \models \phi \vee \psi$ iff $(\pi, i) \models \phi$ or $(\pi, i) \models \psi$.
3. **Temporal Operators:**
   - *Next* Operator:
     $(\pi, i) \models \mathsf{X}\,\phi$ iff $(\pi, i+1) \models \phi$.
   - *Until* Operator:
     $(\pi, i) \models \phi\,\mathsf{U}\,\psi$ iff $\exists j \geq i : (\pi, j) \models \psi$ and $\forall k, i \leq k < j : (\pi, k) \models \phi$

A path π of a Kripke structure M *satisfies* a LTL formula ϕ, if ϕ holds in the state s_0. We define, $\pi \models \phi$ iff $(\pi, 0) \models \phi$. A LTL formula ϕ holds for a Kripke structure M, if all paths π of M satisfy ϕ, so $M \models \phi$ iff for all paths π in M: $\pi \models \phi$.

The semantic of the remaining operators G, F and W derives from the semantics of the primary operators:

- *Globally* Operator:
  $(\pi, i) \models \mathsf{G}\,\phi$ iff $\forall j \geq i : (\pi, j) \models \phi$
- *Eventually* Operator:
  $(\pi, i) \models \mathsf{F}\,\phi$ iff $\exists j \geq i : (\pi, j) \models \varphi$
- *Weak-Until* Operator:
  $(\pi, i) \models \phi\,\mathsf{W}\,\psi$ iff $(\pi, i) \models \phi\,\mathsf{U}\,\psi$ or $(\pi, i) \models \mathsf{G}\,\phi$

2.5.2 LTL-Properties

LTL is widely used to specify properties of reactive hardware and software. The following are some very often used LTL formulas:

- $\mathsf{G}(\phi \rightarrow \mathsf{G}\,\phi)$
  This formula says that if some state s_i in the path π satisfies ϕ, than all following states s_j with $j > i$ also has to satisfy ϕ. So, *if once ϕ, than always ϕ.*
- $\mathsf{G}\,\mathsf{F}\,\phi$
  A path π satisfies this formula, if for all states in π, ϕ holds immediately or in some further state. So this formula expresses the property: *ϕ is* true *at infinitely many states on π.*
- $\mathsf{F}\,\mathsf{G}\,\phi$
  This formula states that at a certain point, the formula ϕ holds at all future states of the path. In other words, *there are only finitely many states, were ϕ does not hold.*
- $\mathsf{G}\,\mathsf{F}\,\phi \rightarrow \mathsf{G}\,\mathsf{F}\,\psi$
  Such formulas are so called *reactivity properties* or *fairness properties.* Fairness properties claim that *if something is requested infinitely often, then it will be granted infinitely often.*
  In many applications such properties are used. E.g. for an arbiter the specification could look like: $\mathsf{G}\,\mathsf{F}\,r \rightarrow \mathsf{G}\,\mathsf{F}\,g$ where again r is the input request variable and g is the output acknowledgment variable.

We can classify LTL formulas in *safety formulas* and in *liveness formulas.* Manna and Pnuelli defined safety and liveness properties in [MP92] as follows:

Definition 2.22. (Safety and Liveness Properties)
Let the path $\pi = s_0, s_1, \ldots$ be an infinite sequence of states $s_i : i \geq 0$. We define $\pi[0 \ldots k]$ for $k \geq 0$ to be the finite prefix of π.

- ϕ is a safety formula iff any path π violating ϕ contains a prefix $\pi[0 \ldots k]$ all of whose infinite extensions violate ϕ.
- ϕ is a liveness formula iff any finite sequence of states $s_0, s_1, \ldots, s_k$ can be extended to an infinite sequence satisfying ϕ.

Safety Properties:

Safety formulas are able to express properties which have to hold at all times. Basically they say that *"something bad will not happen"*. Examples of "bad things" in a program could be that two processes enter a critical Section at the same time, a deadlock occurs, ...

If ϕ is only a propositional formula, the safety condition $\mathsf{G}\,\phi$ defines that ϕ has to hold in all states of the path. This safety condition often occurs in specifications of arbiters. The specification could say that there are never two requests at the same time. This property can be expressed by the safety formula $\mathsf{G}\,\neg(r_0 \wedge r_1)$, where r_0 and r_1 are input variables denoting a request from the first and the second user, respectively.

Another property of the arbiter could be that each request has to be granted in the next tick. The safety formula for this property is $\mathsf{G}(r_0 \rightarrow \mathsf{X}\, g_0)$ and g_0 denotes a grant for the first user.

The violation of any safety property $\mathsf{G}\,\phi$ can be detected in finite time. If ϕ is a propositional formula, we can immediately notice if any state violates ϕ and know that something bad happened. If the safety property is of the form $\mathsf{G}(\phi \rightarrow \mathsf{X}\,\psi)$, and ϕ holds in some state s_i, but ψ does not in s_{i+1}, than we can only detect the violation at state s_{i+1}.

Liveness Properties:

Liveness formula claim that *"something good will happen eventually"*. Examples of "good things" in programs could be starvation freedom, meaning each process is making progress, termination or every packet sent must be received at its destination. Very often they define properties, which have to hold infinitely often.

Let ϕ and ψ be propositional formulas. Then some examples of liveness formulas would be: $\mathsf{F}\,\phi$, $\mathsf{G}\,\mathsf{F}\,\phi$, $\mathsf{F}\,\mathsf{G}\,\phi$, $\mathsf{G}\,\mathsf{F}\,\phi \rightarrow \mathsf{G}\,\mathsf{F}\,\psi$.

Violations of safety properties can be detected immediately. However, in the case of liveness properties, a violation cannot be detected at any point in time [AS85]. It is always possible that a good thing can occur in the future, but we cannot know this at any current state.

2.5.3 GR(1)

Generalized Reactivity of rank 1 (GR(1)) was first introduced by Piterman, Pnueli and Sa'ar in [PPS06] and defines a subset of LTL. In practice, most specifications of reactive systems can be rewritten in GR(1) [PPS06]. A GR(1) specification defines the interaction allowed between environment and system. Let X be a set of Boolean input variables controlled by the environment and Y be a set of Boolean output variables controlled by the system.

Definition 2.23. (GR(1))
A GR(1) formula φ over input variables X and output variables Y can be written in the following form [PPS06]:

$$\varphi = \varphi^e \rightarrow \varphi^s = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_l^e \rightarrow \varphi_i^s \wedge \varphi_t^s \wedge \varphi_l^s, \tag{2.6}$$

for each single part holds the following:

- φ_i^e is a Boolean formula over the input variables X and φ_i^s over the output variables Y.
- φ_t^e and φ_t^s are Boolean formulas. The formulas have the form $\bigwedge_j \mathsf{G}\, q_j$ where q_j is a Boolean combination of input and output variables and expressions of the form $\mathsf{X}\, v$. For φ_t^e we have $v \in X$ and for φ_t^s we have $v \in X \cup Y$.
- φ_l^e and φ_l^s are formulas of the form $\bigwedge_j \mathsf{G}\,\mathsf{F}\, p_j$, where each p_j is a Boolean formula over $X \cup Y$.

Any GR(1) specification $\varphi = \varphi^e \rightarrow \varphi^s$ consists of two parts. The first part φ^e defines the allowed behavior of the environment and is called *environment assumptions*. The second part φ^s defines the expected behavior from the system, and is called *system guarantees*. The implication in the specification requires that if all assumptions are fulfilled, also all guarantees have to be satisfied. For both, assumptions and guarantees we distinguish between safety properties (φ_i and φ_t) and liveness properties (φ_l). A definition of safety and liveness properties can be found in 2.5.2.

The formulas φ_i^e and φ_i^s define the initial state of the environment and the system, respectively. The Boolean formulas φ_t^e and φ_t^s represent the transition relation of the environment and the system. φ_t^e defines the set of possible next inputs in respect to the present input and output. φ_t^s defines the next outputs, also in respect to the current values of input and output. The liveness properties φ_l^e and φ_l^s define properties that have to hold infinitely often.

Example 2.5. We take a look at the GR(1) specification of a simple arbiter [PPS06]. The input variable r denotes a request and the output variable a denotes an acknowledgement. The properties of the arbiter are the following:

- At the begin, both signals are *low*.
  $\varphi_i^e = \neg r$ and $\varphi_i^s = \neg a$.
- The environment is not allowed to send a new request, if the grant signal is still *high*. Furthermore, the environment has to hold a request until it is granted by the system.
  $\varphi_t^e = G(\neg r \wedge a \rightarrow X\neg r) \wedge G(r \wedge \neg a \rightarrow Xr)$.
- The system isn't allowed to give a grant without a request, and it also has to hold the grant for a request, until the request is *low* again.
  $\varphi_t^s = G(\neg r \wedge \neg a \rightarrow X\neg a) \wedge G(r \wedge a \rightarrow Xa)$.
- The liveness conditions for environment and system is defined as follows
  $\varphi_l^e = GF(r \wedge \neg a) \wedge GF(\neg r \wedge a)$ and $\varphi_l^s = GF(r \wedge a) \wedge GF(\neg r \wedge \neg a)$.

The resulting GR(1) formula $\varphi = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_l^e \rightarrow \varphi_i^s \wedge \varphi_t^s \wedge \varphi_l^s$ describes the behavior of the arbiter. Figure 2.7 shows a Moore machine that realizes this specification.

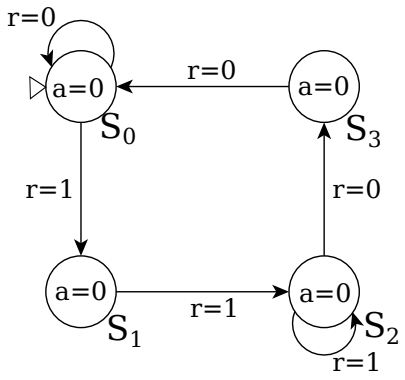


Figure 2.7: Moore Machine for simple Arbiter.

2.5.4 CTL* and CTL

This Section is intended to give only a brief overview of branching time logic, for more details and references please see [Eme90].

Linear-time logics are only able to express properties over individual computation paths. This follows from the fact that in linear-time models there is only one possible path in the future which is realized. Instead, a branching-time model says that there are many different paths in the future, and which one will be realized is not fixed. So *linear-time temporal formulas express properties of computation paths of Kripke structures*, and *branching-time temporal formulas express properties of computation trees of Kripke structures.* A computation tree is an infinite tree, which arises from unwinding a Kripke structure M, see Section 2.1.1.

In 1983, the branching-time logic CTL* was introduced by Emerson and Halpern [EH83], [EH85].

Lemma 2.1. *Let M be a Kripke structure.*

- *a LTL formula ϕ holds for M, if all paths π of M satisfy ϕ.*
- *a CTL* formula ϕ holds for M, if ϕ holds for the computation tree of M.*

CTL* has the same logical operators and temporal operators as LTL. Additionally, CTL* has two *path operators*, the *All*-operator A and the *Exists*-operator E. Therefor, CTL* has more expressive power than LTL, and LTL is a subset of CTL*. The temporal operators describe the properties of a path of the computation tree, and the path operators describe properties on the branching structure of the tree. The meaning of the path operators is the following:

- $\mathsf{A}\,\phi$ holds in a state $s \in S$ of M, iff for **all** paths π starting from s the formula ϕ holds.
- $\mathsf{E}\,\phi$ holds in a state $s \in S$ of M, if **there exists** at least one path π starting from s for which the formula ϕ holds.

A Kripke structure M *satisfies* a CTL* formula ϕ, if ϕ holds in the state s_0.

In 1981, the branching-time logic called Computation Tree Logic (CTL) was introduced by Clarke and Emerson [CE81]. CTL is also a subset CTL*, which is easer to verify. In CTL* all operators can be combined arbitrarily. In contrast, in CTL, there must always be one path operator followed by one temporal operator. So the path operators A and E must immediately be followed by a temporal operator X, F, U or W. Examples for CTL formulas would be $\mathsf{A}\,\mathsf{X}\,\phi$, $\mathsf{E}\,\mathsf{X}\,\phi$, $\mathsf{A}\,\mathsf{F}\,\phi$, ...

CTL and LTL have different expressive powers. This means that there are properties which can be expressed in CTL, but not in LTL, vice versa. CTL can specify the existence of paths like $\mathsf{E}\,\mathsf{X}\,\phi$, which isn't possible in LTL. In return LTL can specify fairness properties like $\mathsf{G}\,\mathsf{F}\,\phi \rightarrow \mathsf{G}\,\mathsf{F}\,\psi$, which isn't possible in CTL.

Since it is more intuitive for most people to write the specifications of systems in LTL than in CTL, LTL is more commonly used [Var01], although verifying LTL properties is harder than verifying CTL properties.

2.5.5 Modal μ-Calculus

Fixed-point theory

First we give a short introduction to fixed-point theory [DG08].

Let S be a set and $F : 2^S \to 2^S$ be a *monotone* function with respect to set inclusion. A function F is monotone if $\forall X, Y \in S : X \subseteq Y \to F(X) \subseteq F(Y)$.

Definition 2.24. (Fixed Point)
Given a set S and a monotone function $F : 2^S \to 2^S$. A *fixed point* is defined as any set $X \subseteq S$ with $F(X) = X$.

Each monotone function $F : 2^S \to 2^S$ has a least and a greatest fixed point. Their existence is a consequence of the Knaster-Tarski-Theorem [BL69].

Definition 2.25. Let S be a set and F: $2^S \to 2^S$ a monotone function. Then F has a least fixed point $\mu(F)$ and a greatest fixed point $\nu(F)$, defined as:

$$\mu(F) := \bigcap \{X \subseteq S : F(X) = X\} = \bigcap \{X \subseteq S : F(X) \subseteq X\} \tag{2.7}$$

$$\nu(F) := \bigcup \{X \subseteq S : F(X) = X\} = \bigcup \{X \subseteq S : F(X) \supseteq X\} \tag{2.8}$$

Least and greatest fixed points can also be calculated inductively. For the least fixpoint we get:

$$X^0 := \emptyset, X^{i+1} := F(X^i) \tag{2.9}$$

with $X^i \subseteq S$ for $i \geq 0$. Since F is a monotone function, we have $\forall i, j$ with $i \leq j : X^i \subseteq X^j$. At some point, this sequence of sets reaches a fixed point, with $X^i = X^{i+1}$ for some i. The least i for which this condition holds defines the so called *least inductive fixed point of* F: X^∞, with $X^\infty := X^i = X^{i+1} = \bigcup_k X^k$ for $k = 0 \ldots i$. Similarly, the greatest fixed point can be calculated recursively by computing the following sequence of sets:

$$X^0 := S, X^{i+1} := F(X^i) \tag{2.10}$$

Again, we get the *greatest inductive fixed point* X^∞, with $X^\infty := X^i = X^{i+1} = \bigcap_i X^k$ for $k = 0 \ldots i$ for some i.

A further consequence of the Knaster-Tarski-Theorem is that the least and greatest fixed point and the least and least and greatest inductive fixed point of every monotone function $F : 2^S \to 2^S$ coincide [KL08].

By definition, each fixed point X is a solution for the equation $F(X) = X$. Besides least fixed point $\mu(F)$ and greatest fixed point $\nu(F)$, there may also exist other fixed points Z of F. For all fixed points Z of F holds that $\mu(F) \subseteq Z \subseteq \nu(F)$.

The modal μ-Calculus

Another branching-time temporal logic is called *modal μ-calculus* (μ-calculus for short). μ-calculus is used as a specification language to formulate properties of reactive systems. Almost all other temporal logics, like CTL*, CTL or LTL an be seen as fragments of μ-calculus. Its currently used form was introduced by Kozen in [Koz83]. For further information about modal μ-calculus see [BBW06] and [Eme96].

In addition to the Boolean operators, the μ-calculus consists of the *nexttime operators* AX and EX and the fixpoint operators μ and ν. The operators AX and EX are already known from CTL, and have the same meaning. The operator $\mathsf{EX}\,\varphi$ means that φ holds in at least one of the successor states of the current state. The formula $\mathsf{AX}\,\varphi$ holds in a state, if φ holds in all successor states. Fixpoint operators bind free variables. E.g. consider the formula $\mu Y\varphi$. The least fixpoint operator μ binds each occurrence of the propositional variable Y in the μ-calculus formula φ. A *closed μ-calculus formula* contains no free variables.

Definition 2.26. (Syntax of μ-calculus)

Let AP be the set of atomic propositions, and V be the set of relational variables. The syntax of a μ-calculus is inductively defined as follows:

1. **Atomic Proposition:** p is a μ-calculus formulas for all $p \in AP$.
2. **Propositional Variables:** $\forall X \in V$ are μ-calculus formulas.
3. **Boolean Operators:** If φ and ψ are μ-calculus formulas, then so are $(\neg\varphi)$, $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$.
4. **Nexttime Operator:** If φ is a μ-calculus formula, than so are $\mathsf{EX}\,\varphi$ and $\mathsf{AX}\,\varphi$.
5. **Fixpoint Operators:** If φ is a μ-calculus formula and $X \in V$, than $\mu X\varphi$ and $\nu X\varphi$ are also μ-calculus formulas.

Usually, μ-calculus formula describe properties of Kripke structures (see 2.1). A μ-calculus formula φ is interpreted over states $s \in S$ of a Kripke structure M in which φ holds.

Definition 2.27. (Semantic of μ-calculus)
Let AP be de set of atomic propositions, V the set of relational Variables, and $M = \{S, S_0, R, L\}$ be a Kripke structure. We define $\|\varphi\|^i_M$ as the set of states of the Kripke structure M, for which the μ-calculus formula φ holds. The interpretation function $i : V \to 2^S$ assigns a set of states of S to each free variable. The formula $i[X \leftarrow Z]$ denotes that $i[X \leftarrow Z](X) = Z$ and $i[X \leftarrow Z](Y) = i(Y)$ for $Y \neq Z$.

The sets $\|\varphi\|^i_M$ and $\|\psi\|^i_M$ are inductively defined as follows:

- $p \in AP : \|p\|^i_M = \{s \in S : s \models p\}$,
- $p \in AP : \|\neg p\|^i_M = \{s \in S : s \not\models p\}$,
- $X \in V : \|X\|^i_M = i(X)$,

- $\|\varphi \wedge \psi\|_M^i = \|\varphi\|_M^i \cap \|\psi\|_M^i$,
- $\|\varphi \vee \psi\|_M^i = \|\varphi\|_M^i \cup \|\psi\|_M^i$,
- $\|\,\mathsf{EX}\,\varphi\|_M^i = \{s \in S : \exists s' \text{ with } (s, s') \in E \text{ and } s' \in \|\varphi\|_M^i\}$,
- $\|\,\mathsf{AX}\,\varphi\|_M^i = \{s \in S : \forall s' \text{ with } (s, s') \in E \text{ and } s' \in \|\varphi\|_M^i\}$,
- $\|\mu X \varphi\|_M^i = \bigcup_j X_j$ where $X_0 = \emptyset$ and $X_{j+1} = \|\varphi\|_M^{i[X \leftarrow X_j]}$, and
- $\|\nu X \varphi\|_M^i = \bigcap_j X_j$ where $X_0 = S$ and $X_{j+1} = \|\varphi\|_M^{i[X \leftarrow X_j]}$,

If all variables are bound by a least or a greatest fixpoint operator, than there are no free variables, and the interpretation function i does not matter any more, and we write $\|\varphi\|^M$ instead of $\|\varphi\|_M^i$. For simplicity, if the Kripke structure M is clear, we only write $\|\varphi\|$.

Let φ be a μ-calculus formula, and $Y \in V$. Than the relation between the operators fulfills the following equations:

$$\mathsf{EX}\,\varphi = \neg\,\mathsf{AX}\,\neg\varphi, \text{ and} \tag{2.11}$$

$$\mu Y \varphi = \neg \nu Y \neg \varphi, \tag{2.12}$$

2.6 Games

To solve the synthesis problem we translate the GR(1) specification into an infinite two-player game, played between an environment and a system. The system is winning, if it is able to satisfy the specification, regardless of the actions of the environment. Our notation of games is based on the notation introduced in [PPS06].

Definition 2.28. (Symbolic Game Structure)
A game structure G is defined as a 7-tuple $G = (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$, when

- $V = \{v_0 \dots v_n\}$ is a finite set of Boolean variables. A state is an interpretation of V, i.e. in a state s, each variable $v \in V$ has an assigned value. $\Sigma = 2^V$ defines the set of all states.
- $X \subseteq V$ is the set of environment input variables. The input domain is defined by $D_X = 2^X$ and defines all possible valuations to the variables in X.
- $Y = V \setminus X$ is the set of system output variables. The output domain is defined by $D_Y = 2^Y$ and defines all possible valuations to the variables in Y.
- The formula Θ defines all initial states. A state s_0 is an initial state if it satisfies Θ.
- $\rho_e(X, Y, X')$ is a Boolean formula and is called an environment transition relation. It relates each state to possible next input values. Let X' be a primed copy from X. For a state $s \in \Sigma$, $x' \in D_X$ is a possible next input value, if $(s, x') \models \rho_e$.

- $\rho_s(X, Y, X', Y')$ is a Boolean formula and is called a system transition relation. It relates each state and the next input value to possible next output values. Let X' and Y' be primed copies from X and Y, respectively. For a state $s \in \Sigma$ and next input value $x' \in D_X$, $y' \in D_Y$ is a possible next output value, if $(s, x', y') \models \rho_s$. x' and y' define the next state $s' = (x', y')$.
- φ is the winning condition defined by a LTL formula.

Game Structures are often represented symbolically, for instance by using BDDs. The inputs and the outputs are defined by sets of Boolean variables: $X = x_0, x_1, \ldots, x_n$ and $Y = y_0, y_1, \ldots, y_m$. The initial states, the environment transition relation, the system transition relation and the winning condition are represented by Characteristic Formulas. In order to write a Characteristic Formula for the transition relations, we introduce two additional Sets of Boolean variables. The set $X' = x'_0, x'_1, \ldots, x'_n$ represents the next input variables and the set $Y' = y'_0, y'_1, \ldots, y'_m$ represents the next output variables. The symbolic representation of game structures is very similar to the symbolic representation of $\omega - automaton$, see Section 2.3.4.

In a game for a reactive system, the environment is Player 0 and controls the input variables X and the system is Player 1 and controls the output variables Y. A **play** is an infinite sequence of states. In each step, the environment chooses a next input $x' \in D_X$ and the system responds by choosing a next output $y' \in D_Y$. By doing so, they are defining the next state $s' = (x', y')$. This results in an infinite sequence of states. This corresponds to passing a token from one state to the next state.

Definition 2.29. (Play $\bar{\sigma}$)
A **play** $\bar{\sigma}$ of a game structure G is an infinite sequence of states $\bar{\sigma} = s_0, s_1, s_2, \ldots$, with $s_0 \models \Theta$ and $\forall i \geq 0 : (s_i, s_{i+1}) \models \rho_e \wedge \rho_s$. s_{i+1} is called the successor of s_i.

The goal of the system is to fulfill the specification, and the environment seeks to prevent this. The *system wins a play* $\bar{\sigma}$, if the play $\bar{\sigma}$ fulfills the winning condition φ. Otherwise the environment wins the play. The set **Win** contains all plays, which are winning plays for the system.

$$Win = \{\bar{\sigma} : \bar{\sigma} \models \varphi\} \tag{2.13}$$

2.6.1 Strategies

A *strategy for the environment* is a function $f_e : \Sigma^+ \to D_X$, which defines a next input value for each step in the play. For a sequence of states $\bar{\sigma} = s_0, s_1, \ldots s_n$, we get the next input value $x' \in D_X$ with $x' = f_e(\bar{\sigma})$ and $(s_n, x') \models \rho_e$.
A *strategy for the system* is a function $f_s : \Sigma^+ \times D_X \to D_Y$, which defines for each current state and next input value a next output value. For a sequence of states $\bar{\sigma} = s_0, s_1, \ldots s_n$ and next input value $x' \in D_X$, we get the next output value $y' \in D_Y$ with $y' = f_s(\bar{\sigma}, x')$ and $(s_n, x', y') \models \rho_s$. A strategy f_s is called a ***winning strategy*** for a given state s, if every play $\bar{\sigma}$ with initial state s and played according to the strategy f_s, is a win for the system. The

winning region W_s is defined as the set of all states from which the system has a winning strategy. Winning strategy and winning region W_e are defined dually for the environment.

Different types of **winning conditions** φ exist, which define different types of games. Winning conditions are defined over states, which are visited finitely or infinitely during a play.

- $\mathsf{occ}(\bar{\sigma})$ defines the set of states occurring often during a play $\bar{\sigma}$.
- $\mathsf{inf}(\bar{\sigma})$ defines the set of states occurring *infinitely* often in a play $\bar{\sigma}$.

All winning conditions are defined by using $\mathsf{inf}(\bar{\sigma})$ or $\mathsf{occ}(\bar{\sigma})$. In our setting, we need two types of games for the synthesis of reactive systems: *Generalized Reactivity Games* and *Streett Games*. In the following, we define the most common used winning conditions, with respect to Section 2.3.3.

2.6.2 Safety Games

Definition 2.30. (Safety Winning Condition)
Let G be a game structure, F a finite set of states and $\bar{\sigma} = s_0, s_1, \ldots$ be a play. The safety winning condition φ is defined by:

$$\varphi(\bar{\sigma}) \Leftrightarrow \mathsf{occ}(\bar{\sigma}) \subseteq F. \tag{2.14}$$

F defines a set of *safe states*. A play $\bar{\sigma}$ is winning for player 0, if the set of visited states corresponds to the set of safe states: $\bar{\sigma} \in Win \Leftrightarrow \forall i : s_i \in F$.

2.6.3 Reachability Games

Definition 2.31. (Reachability Winning Condition)
Let G be a game structure, F a finite set of states and $\bar{\sigma} = s_0, s_1, \ldots$ be a play. The reachability winning condition φ is defined by:

$$\varphi(\bar{\sigma}) \Leftrightarrow \mathsf{occ}(\bar{\sigma}) \cap F \neq \emptyset. \tag{2.15}$$

In this case, F defines a set of *target states*. Player 0 wins the play, if a state in F is visited: $\bar{\sigma} \in Win \Leftrightarrow \exists i : s_i \in F$.

2.6.4 Büchi Games

Definition 2.32. (Büchi Winning Condition)
Let G be a game structure, F a finite set of states and $\bar{\sigma} = s_0, s_1, \ldots$ be a play. The Büchi winning condition φ is defined by:

$$\varphi(\bar{\sigma}) \Leftrightarrow \mathsf{inf}(\bar{\sigma}) \cap F \neq \emptyset. \tag{2.16}$$

2.6.5 co-Büchi Games

Definition 2.33. (co-Büchi Winning Condition)
Let G be a game structure, F a finite set of states and $\bar{\sigma} = s_0, s_1, \ldots$ be a play. The co-Büchi winning condition φ is defined by:

$$\varphi(\bar{\sigma}) \Leftrightarrow \mathsf{inf}(\bar{\sigma}) \cap F = \emptyset. \tag{2.17}$$

2.6.6 Muller Games

Definition 2.34. (Muller Winning Condition)
Let G be a game structure, $\bar{\sigma} = s_0, s_1, \ldots$ be a play and F a set of sets of states $F = \{F_1, \ldots, F_k\}$ with $F_i \subseteq \Sigma$ for $i = 1 \ldots k$. The Muller winning condition φ is defined as follows:

$$\varphi(\bar{\sigma}) \Leftrightarrow \mathsf{inf}(\bar{\sigma}) \in F. \tag{2.18}$$

2.6.7 Rabin Games

Definition 2.35. (Rabin Winning Condition)
Let G be a game structure and $\bar{\sigma} = s_0, s_1, \ldots$ be a play. The Rabin winning condition φ is given as a set of pairs of sets of states $\varphi = (E_1, F_1), (E_n, F_n), \ldots, (E_n, F_k)$ with $E_i, F_i \subseteq \Sigma$ for $i = 1 \ldots k$. φ is defined as follows:

$$\varphi(\bar{\sigma}) \Leftrightarrow \bigvee_{i=1}^{k} (\mathsf{inf}(\bar{\sigma}) \cap E_i = \emptyset \wedge \mathsf{inf}(\bar{\sigma}) \cap F_i \neq \emptyset) \tag{2.19}$$

2.6.8 Streett Games

Definition 2.36. (Streett Winning Condition) The Streett winning condition φ is given as a set of pairs of sets of states $\varphi = (E_1, F_1), (E_n, F_n), \ldots, (E_n, F_k)$ with $E_i, F_i \subseteq \Sigma$ for $i = 1 \ldots k$. φ is defined as follows:

$$\varphi(\bar{\sigma}) \Leftrightarrow \bigwedge_{i=1}^{k} (\mathsf{inf}(\bar{\sigma}) \cap E_i \neq \emptyset \rightarrow \mathsf{inf}(\bar{\sigma}) \cap F_i \neq \emptyset) \tag{2.20}$$

2.6.9 Parity Games

Definition 2.37. (Parity Winning Condition)
Let G be a game structure, $c : \Sigma \rightarrow \{1, \ldots, k\}$ with k as even and $\bar{\sigma} = s_0, s_1, \ldots$ being a play. The parity winning condition φ is defined by:

$$\varphi(\bar{\sigma}) \Leftrightarrow max(\mathsf{inf}(c(\bar{\sigma}))) \; is \; even. \tag{2.21}$$

2.6.10 GR(1) Games

Definition 2.38. (GR(1) Condition)
Let G be a game structure and $E = E_1, \ldots, E_k$ and $F = F_1, \ldots, F_l$ with $E_i \subseteq \Sigma$ for $i = 1 \ldots k$ and $F_j \subseteq \Sigma$ for $j = 1 \ldots l$. The GR(1) winning condition φ is defined by:

$$\varphi(\bar{\sigma}) \Leftrightarrow \bigwedge_{i=1}^{k} (\mathsf{inf}(\bar{\sigma}) \cap E_i \neq \emptyset) \rightarrow \bigwedge_{j=1}^{l} (\mathsf{inf}(\bar{\sigma}) \cap F_j \neq \emptyset) \tag{2.22}$$

2.6.11 The modal μ-Calculus over Game Structures

In order to solve game, we use a slightly different definition of modal μ-calculus. μ-calculus over game structures is defined in [PP06] and in [PPS06]. The definition of game structures can be found in Section 2.6 and the definition of μ-calculus in Section 2.5.5.

Definition 2.39. (Syntax μ-calculus over game structures)
Let $G = (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$ be a *game structure*, and V the set of *relational variables*. The **syntax** of a μ-calculus formulas is inductively defined as follows:

- $\forall v \in V$: v and $\neg v$ are μ-calculus formulas (*atomic formulas*).
- $\forall X \in V$ are μ-calculus formulas.
- If φ and ψ are μ-calculus formulas, then so are $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$.
- If φ is a μ-calculus formula, then so are $\diamondsuit\varphi$ and $\boxdot\phi$.
- If φ is a μ-calculus formula and $X \in V$, than $\mu X\varphi$ and $\nu X\varphi$ are also μ-calculus formulas.

Before defining the semantic, let's again look at the operators in an informal way. Here, the μ-calculus formula φ is interpreted over states $s \in \Sigma$ of the game structure G, in which φ holds. We define $\|\varphi\|_G^i$ as the set of states of the game structure G, for which φ holds. The interpretation function $i : Var \rightarrow 2^\Sigma$ assigns to each free variable a subset of Σ. The meaning of the attractor-operators $\diamondsuit$ and $\boxdot$ differ from nexttime-operators used before, and are defined as follows:

- The **Attractor-Operator** $\diamondsuit$.
  The attractor $\|\diamondsuit\varphi\|_G^i$ defines the set of all states from which the system can force the game into a state in which φ holds in one step. No matter what input the environment chooses, the system can always choose an output, so that the successor state is in $\|\varphi\|_G^i$.
- The **Attractor-Operator** $\boxdot$.
  Has the same effect, only for the other environment. $\|\boxdot\varphi\|_G^i$ defines the set of all states, from which the environment can force the game in a state in which φ holds in one step, regardless of the output of the system.

Definition 2.40. (Semantic μ-calculus over game structures)
Let φ and ψ be μ-calculus formulas. The set $\|\varphi\|_G^i$ and $\|\psi\|_G^i$ is inductively defined as follows:

- $v \in V : \|v\|_G^i = \{s \in \Sigma : s[v] = 1\}$.
- $v \in V : \|\neg v\|_G^i = \{s \in S : s[v] = 0\}$.
- $X \in Var : \|X\|_G^i = i(X)$.
- $\|\varphi \wedge \psi\|_G^i = \|\varphi\|_G^i \cap \|\psi\|_G^i$.
- $\|\varphi \vee \psi\|_G^i = \|\varphi\|_G^i \cup \|\psi\|_G^i$.
- $\|\langle\!\rangle\varphi\|_G^i = \{s \in \Sigma : \forall x' \in D_X$ with $(s, x') \models \rho_e : \exists y' \in D_Y$ with $s' = (x', y')$ and $(s, s') \models \rho_s$ and $s' \in \|\varphi\|_G^i\}$.
- $\|[\!]\varphi\|_G^i = \{s \in \Sigma : \exists x' \in D_X$ with $(s, x') \models \rho_e : \forall y' \in D_Y$ with $s' = (x', y')$ and $(s, s') \models \rho_s$ and $s' \in \|\varphi\|_G^i\}$.
- $\|\mu X\varphi\|_G^i = \bigcup_j S_j$ where $S_0 = \emptyset$ and $S_{j+1} = \|\varphi\|_G^{i[X \leftarrow S_j]}$
- $\|\nu X\varphi\|_G^i = \bigcap_j S_i$ where $S_0 = S$ and $S_{j+1} = \|\varphi\|_G^{i[X \leftarrow S_j]}$

If there are no free variables and the game structure G is clear, we write $\|\varphi\|$ instead of $\|\varphi\|_G^i$.

Example 2.6. For a better understanding, let's have a look at some examples of μ-calculus formulas from [PP06]:

- $\varphi = \nu X(v \wedge \langle\!\rangle X)$ with $v \in V, X \in Var$
  $s \in \|\varphi\|$, if player 0 can force the game to visit only states where v holds.
- $\varphi = \mu X(v \vee [\!]X)$ with $v \in V, X \in Var$
  $s \in \|\varphi\|$, if player 1 can force a visit of a state where v holds.

2.7 Implementation

This Section gives a short introduction on how RATSY [BCG+10b] works, its components and how it all fits together.

2.7.1 RAT

In 2007, Bloem et al. [BCP+07] presented the *Requirement Analysis Tool*, RAT. Writing formal specifications for circuits is a tricky task, in which errors can easily be overlooked. RAT supports the user in developing and analyzing his specifications by providing three main functionalities: *Property Assurance*, *Property Simulation* and *Property Realizability*. RAT supports specifications written in PSL [EF06]. All functionality of RAT can be accessed via its graphical user interface.

Property Simulation

Property Simulation allows the user to interactively check his specification. The behavior of properties can be explored, by looking at possible time traces of the input and output signals. Here, the user has several possibilities. He can either ask RAT to provide a possible time trace. Then the user can fix certain values of any signal in any time step and then ask RAT, if the new trace still fulfills the specification. This helps the user to understand, what impacts the different requirements of his specification have, and whether the specification does what the user had in mind. Also, the user can provide RAT with input values, and RAT answers with possible output values. It's also possible the other way around. The user can provide the output values, and RAT delivers possible input values. Property simulation is very similar to classical hardware simulation.

Property Assurance

Property Assurance provides the user with two functionalities. First, it enables a user to check properties automatically against contradiction and consistency. Second, the user can define two sets of properties, *assertions* and *possibilities*. The designer can use assertions to make sure that undesired behavior is forbidden by the specification. Possibilities are used to check, if a certain described behavior is still allowed by the specification. So, these two sets supply the designer with a tool to check whether his specification is strict enough to exclude bad behavior, and at the same time is not too strict and still allows good behavior. This way, property assurance helps the user to understand, whether he has written the right specification for his desired circuit.

Property Realizability

Property Realizability is the third functionality of RAT. Realizability is the problem of checking, whether there exists a system that fulfills the specification. We deviate between two types of properties in the specification. *Environment assumptions* restrict the set of possible input sequences, and *system guarantees* define the allowed behavior of the system. For any allowed input sequence, a correct system must be able to provide an output sequence, which satisfies all system guarantees. If the system is not able to do so, the specification is not realizable. In this case, the user can perform property simulation, in order to find out the reason for un-realizability.

With these three functionalities; Property Assurance, Property Simulation and Property Realizability, a designer can iteratively develop his specification. First, he can write a specification with assertions and possibilities. Then he can check if the specification is consistent, if no assertions are violated, if all possibilities are fulfilled and if the specification is realizable. If there are any problems, the designer can use the simulation functionality and the diagnosis information provide to find the problem. Afterwards, he can refine the specification and test it again.

Technical Aspects

Property Assurance and Property Simulation both use Bounded Model Checking methods. Property Realizability is solved via a two player game between the environment and the system. If a winning strategy exists for the system, the specification is realizable. RAT uses the model checker NuSMV [CCGR00] and VIS [ea96].

RAT has been used in practical projects [Aue06] and was found to be very helpful.

2.7.2 RATSY

In 2010, Bloem et al. [BCG+10b] presented the successor tool of RAT, called RATSY (Requirements Analysis Tool with Synthesis). RATSY extends RAT's functionality with three new features. First, the tool provides an improved user interface, in which the user can specify properties in the form of Büchi word automaton. Second, it implements a game-based debugging approach. Third, the tool is now able to automatically synthesize given specifications.

Typical Design Workflow

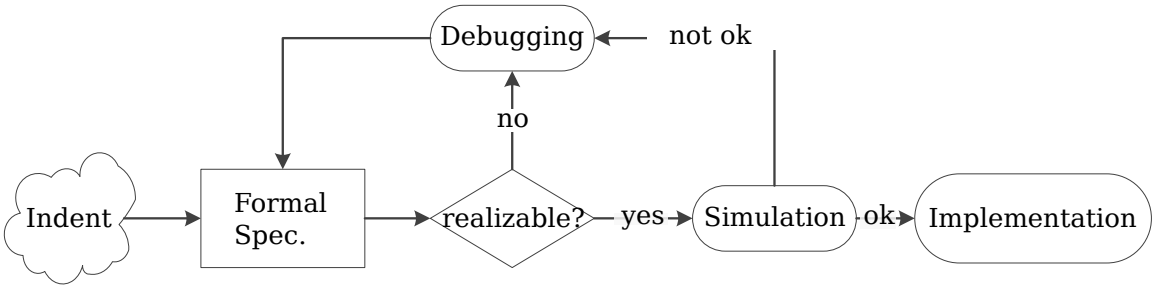


Figure 2.8: Typical Design Workflow.

Figure 2.8 shows the typical process of writing a correct specification. First, the user tries to formulate his idea of the design. To make this task easier and more intuitive, RATSY provides an automata editor, which automatically translates Büchi automata into PSL properties. Then the resulting specification is tested for realizability and is then simulated. If something goes wrong, the user has to debug his specification. RATSY provides a special new modus just for debugging. When the specification finally satisfies the intention of the user, RATSY can automatically synthesize a correct implementation. With the new features, RATSY overcomes all the shortcomings of RAT and is able to optimally support the user in the design process.

Automaton Editor

With the graphical automaton editor, the user can draw a complete deterministic Büchi automaton. Automata are much easier to understand than PSL formulas, and hence it is easier to express design intents by using automata. The tool automatically maintains completeness, by defining a default *dead state*. Edges, which are not defined by the user, point automatically to this state. By only allowing deterministic complete automata, the synthesis' problem gets easier. RATSY automatically transforms the Büchi automata into PSL properties for further processing.

Game Based Debugging

RATSY implements a game-based debugging approach presented in [KHB09]. If a specification is unrealizable, the user has to first identify the problem, before he can refine the specification. A specification is unrealizable, if the environment has a winning strategy in the two player game. This strategy is called counter strategy. If the environment plays according to this strategy, the system can't do anything to fulfill the specification.

For the debugging of unrealizable specifications, the user can play a counter game against RATSY to understand the problem. In every time step, RATSY chooses an input value, and the user chooses an output value. The tool chooses the input values according to the counter strategy, to win the game. At some point, the user is not able to choose a proper output anymore and loses. Although he lost the game, understanding why he lost, helps him to understand the problem. Either, the user has to restrict some of the environment's behavior which led to this situation, or he has to relax the guarantees for the system.

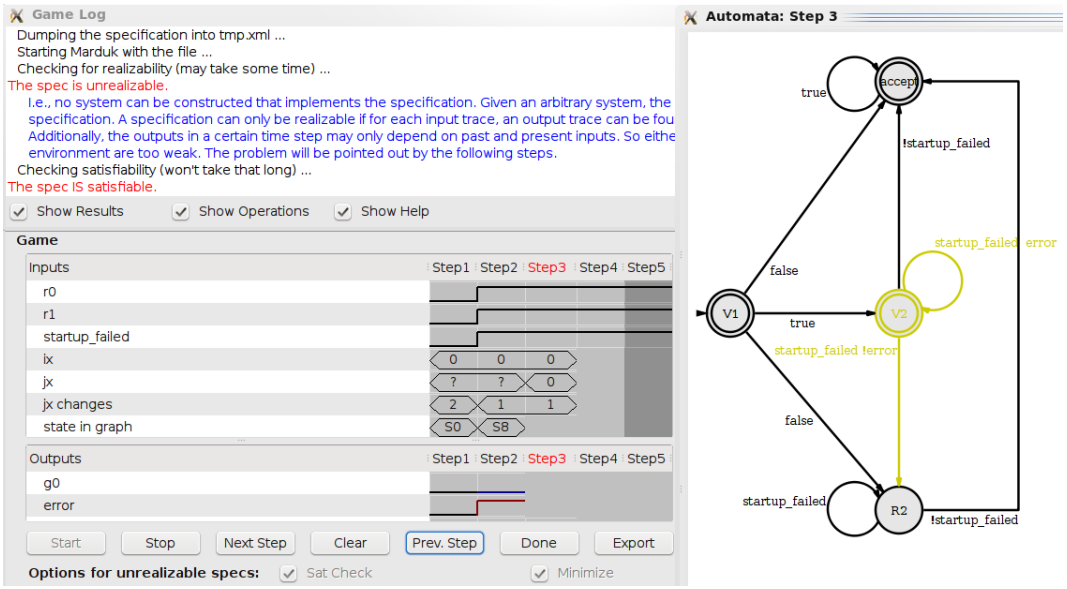


Figure 2.9: RATSY's User Interface while playing a Counter Game.

Figure 2.9 shows a part of RATSY's GUI while playing a counter game. The tool has already chosen the input values. The user selects the output values via the game window or via the automaton window. In the automaton window, possible next states and transitions are marked. By the second time step, the user ends up in an error state and loses.

Synthesis

RATSY is able to synthesize correct implementations from realizeable specifications. The user can choose between three different hardware description languages: BLIF , Verilog and HIF. For synthesis, RATSY uses the external tool MARDUK, which is a reimplementation of the tool Anzu[JGWB07] in Python. The tool works with specifications written in GR(1). By using the NUSMV library, RATSY can translate many specifications into GR(1). Since synthesis can be rather time-consuming, it is possible to run MARDUK in the background, independently from the GUI of RATSY. More information on MARDUK follows in Section 2.7.3.

Technical Aspects

RATSY is written in Python. It utilizes the symbolic model checker NUSMV [CCG+02]. NUSMV is also used to apply syntactical transformations to specifications before handing them to MARDUK. It also handles the conversion of multi-valued variables into Boolean signals. Ratsy and NUSMV heavily use CUDD [Som98]. CUDD is a Decision Diagram Package which provides data-structures and basic BDD [Bry95] operations. NUSMV and CUDD are written in C/C++, Ratsy is not. Therefore SWIG [Bea96] is used to generate a wrapper around them, giving Python and Ratsy the ability to access their functionality.

2.7.3 Marduk

MARDUK is a tool which is able to generate hardware descriptions from GR(1) specifications. It implements an efficient symbolic algorithm [PPS06], which needs N^3 time, where N is the size of the state space. MARDUK is the successor of ANZU, which had nearly the same functionality. The greatest difference between ANZU and MARDUK is the programming language used for the implementation. ANZU was written in Perl, whereas MARDUK is written in Python. The reason for rewriting ANZU into Python, may be the easier interaction with RATSY, since RATSY is also written in Python.

MARDUK takes as input a set of input variables, a set of output variables and a GR(1) specification, which consists of assumptions and guarantees. It transforms the specification into a transition system. Safety properties define the initial states and the transitions, and liveness properties define accepting states, which have to be visited infinitely often. If the specification is realizable, MARDUK creates a BDD representing all possible implementations. In the final step, MARDUK constructs a circuit from the BDD.

2.8 Related Work on Robust Synthesis

In 2009, Bloem, Greimel, Henzinger and Jobstmann [BGHJ09] introduced a notation of robustness and solved the verification and the synthesis problem of robust systems for safety specifications. They defined *a system to be robust if a finite number of environment failures induces only a finite number of system failures.*

The user provides an error specification, which consists of a pair of error functions (d_e, d_s). An error function measures the number of specification violations in some appropriate sense and assigns to each run σ a natural number or infinite. If a run satisfies the specification, the value of the error function is 0. Other values indicate a violation of the specification. The higher the value, the more serious the violation. The error function maps to infinite, if there is an infinite number of safety violations during the run. d_e is the error function for the environment, and d_s is the error function for the system. *A system is robust with respect to an error specification* (d_e, d_s)*, if for each run* σ *a finite number of* $d_e(\sigma)$ *implies a finite number of* $d_s(\sigma)$*.* Note that this condition can be encoded as a Streett pair.

The error specification (d_e, d_s) can be defined by a double cost automaton with two weights w_e and w_s. This double cost automaton is the product of two single cost automaton, one for the environment with cost function w_e and one for the system with cost function w_s. Thus, $d_e(\sigma)$ and $d_s(\sigma)$ correspond to the sum of w_e and w_s of a run over σ, respectively. Each single cost automaton can be constructed from a set of cost automata. So the user can simply define a cost automaton for each safety assumption and and each safety guarantee, and together the cost automatons form a double cost automaton, which defines the error specification.

To solve the synthesis problem, the error specification is transformed into a Streett game with one pair $\langle F_1, F_2 \rangle$. F_1 is the set of states with incoming transitions with systems costs and F_2 is the set of states with incoming transitions with environment costs. The solution of the game corresponds to a robust implementation of the specification. The system can be synthesized in polynomial time.

Additionally, they defined another measure of robustness. *A system is k-robust if the ratio of the system error to the environment error is smaller than or equal to k for every word of the system.* The synthesis problem for k-robustness is solved with a novel game type; the *ratio game*. Based on ratio games they are able to synthesize k-robust systems with minimal k in pseudopolynomial time. So the most robust system for a given specification is the system with the smallest possible k.

The big disadvantages of the previous discussed method from Bloem et al. are that the method only deals with safety specifications and that the user has to specify the proper reaction to each environment failure, in addition to the normal behavior. Writing error specifications is time consuming, leads to bigger specifications and is error prone. The robust synthesis method presented in this work requires no additional effort from the user and handles specifications consisting of safety and liveness properties.

Bloem et al. continued their work on robust synthesis. One year later, in 2010, they presented an algorithm that was able to deal with liveness properties [BCG+10a]. So instead of pure safety specification, the new paper supported the important class of GR(1) specifications. In

order to deal with liveness properties, they used a different notation of robustness. They used another definition of robustness, because it is not possible to notice a violation of a liveness property at any point in time, and therefor it is not possible to count the liveness violations. Bloem et al. defined *a system to be robust, if for any number of environment assumptions that is violated, there is a minimal number of system guarantees that must still be fulfilled.*

Any GR(1) specification can be turned into a robustness specification, which has the form of a General Generalized Reactivity formula of rank k. Suppose that the GR(1) specification consists of m assumptions $\mathcal{A} = \{A_1, \ldots, A_m\}$ and n guarantees $\mathcal{G} = G_1, \{\ldots, G_n\}$. Let A_k and G_k a be the set of all subsets of $\mathcal{A}$ and $\mathcal{G}$ of size k, respectively. The robustness specification has the form:

$$(\bigvee_{A \in A_k} \bigwedge_{A_i \in A} A_i) \rightarrow (\bigvee_{G \in G_l} \bigwedge_{G_i \in G} G_i). \tag{2.23}$$

Any system satisfying this specification has to fulfill l guarantees when k assumptions are fulfilled. It is also possible to compare systems according to their robustness. The more guarantees a system satisfies with the same number of fulfilled assumptions, the more robust the system is. The synthesis algorithm proposed in this paper always generates the *most* robust system.

To synthesize systems from Generalized Reactivity specifications, a Generalized Reactivity game has to be solved. In order to do that, Bloem et al. introduced a new algorithm which reduced the game with Generalized Reactivity objective into a game with Streett objective. A special case of this reduction is the reduction from a GR(1) game into a one-pair Streett game via counting construction. We used this idea for our robust synthesis method, which is presented in this thesis, for more detail on this step see Chapter 3.5. For games with Generalized Reactivity objective of rank k, this step is repeated k times. Each GR(1) objective is turned into Streett pair via counting construction. This results in a Streett game with k-pairs, which is solved with the algorithm of [PP06]. Both counting construction and solving the Streett game can be done symbolically.

The disadvantage of the method in [BCG$^+$10a] is that the system does not recover from safety assumption violations. In our method, if the environment makes a failure, eventually, the system stops making failures. This is not the case according to the definition of robustness in [BCG$^+$10a].

Robustness of sequential circuits was addressed by Doyen, Henzinger, Legay and Nickovic in [DHLN10]. Input values of digital systems often come from analog devices or from a physical environment. Since the environment cannot be fully captured, the input values are not precise. A system is robust if small changes in the input values do not result in huge changes in the output values. Doyen et al. studied a notation of robustness for sequential circuites and presented an algorithm to decide, whether a circuit is robust or not.

In 2011, Majumdar, Render and Tabuada presented a paper about robust discrete synthesis against unspecified disturbances [DHLN10]. Here robustness is not defined in terms of assumption and guarantee violations, but using metrics on the state of a system. Synthesis is performed via special automata incorporating these metrics.

Robust systems are closely related to *fault-tolerant* systems. Fault-tolerance is the property that enables a system to continue its operation even in the case of unexpected inputs or in the case of an error in hardware or software of some of its components. In [GÖ99], Gärtner studied fault tolerance for distributed systems. This paper formalizes important concepts of this area, and gives the reader a good introduction to this topic. In order to design a fault-tolerant system, the first thing to do is to specify a *fault class* that should be tolerated. Usually this means that the program still has to satisfy all of its safety and liveness guarantees even in the presence of a failure of this fault class. In order to make systems fault-tolerant, one must add some form of redundance to the system, via providing multiple identical instances of the same hardware which run in parallel or implementing an algorithm in software in different ways and run them in parallel.

In [KE05], Kulkarni and Ebnenasir focused on the synthesis of failsafe fault-tolerant systems, where fault-tolerance is added to an existing program. A failsafe system satisfies its safety properties in the presence of faults from the fault class, but its liveness properties may be violated. In [EKA08], Kulkarni at al presented a tool to add fault-tolerance to existing finite-state programs. This was the first automated tool to synthesize fault-tolerant distributed programs. More recently, [CRKB11] and [GR09] also presented tools to synthesize fault-tolerant systems, using controller synthesis.

3 Robust Synthesis from GR(1) Specifications

3.1 Idea

Synthesized systems are guaranteed to be correct in respect to a given specification, but there is no guarantee that these systems are also robust against environment assumption violations. Systems that are not robust do not behave reasonably in unexpected situations, i.e., when environment assumptions are violated.

Many specifications consist of environment assumptions and system guarantees. Guarantees must be fulfilled only if all assumptions are satisfied. If assumptions are violated, than the system can behave arbitrary. For both assumptions and guarantees, we may distinguish between safety and liveness properties. In this work, we build systems that are robust to safety assumption violations.

In order to define robustness, we define a system failure to be a violation of a safety guarantee, and an environment failure to be a violation of a safety assumption. *We define a system to be robust if a finite number of environment failures induces a finite number of system failures* [BGHJ09]. Let's assume the environment produces an environment failure for one tick. After some time, a robust system should recover and shouldn't produce system failures any more. Even if there is a finite number of environment failures, a robust system should still fulfill all liveness guarantees. Liveness properties state that some property will hold eventually. If there are finitely many environment failures, then the system works correctly for an infinitely long time, and should be able to fulfill all liveness guarantees.

We implemented our robust synthesis algorithm in the requirements analysis and synthesis tool RATSY [BCG$^+$10b]. With this extension, RATSY is now able to synthesize robust systems from realizable GR(1) specifications. Our algorithm first turns a GR(1) specification into a one-pair Streett game such that a winning strategy corresponds to a correct implementation. Furthermore, we add a second pair such that the winning strategy corresponds to a robust system. In this Chapter, we explain the single steps of the algorithm in detail.

3.2 Definition of Robustness

> A system should not only be correct, it should also behave reasonably even in circumstances that were not anticipated in the requirement specification[...][GJM91].

This means that a system should be correct and robust.

In RATSY we consider GR(1) specifications. In case of reactive systems, a GR(1) specification φ over input variables X and output variables Y consists of environment assumptions and system guarantees:

$$\varphi = A \rightarrow G = A^i \wedge A^t \wedge A^l \rightarrow G^i \wedge G^t \wedge G^l. \tag{3.1}$$

The formulas A^i and G^i define the initial condition for environment and system, respectively. The formulas $A^t = A_1^t \wedge \cdots \wedge A_n^t$ and $G_t = G_1^t \wedge \cdots \wedge G_m^t$ define the transition relation for the environment and the system, respectively. The environment transition relation formulas define for all time steps the next possible input values and the system transition relation formulas define the next possible outputs, in respect to the next input. The initial formulas and the transition formulas form the safety component of the GR(1) specification. The formulas $A^l = A_1^l \wedge \cdots \wedge A_k^l$ and $G_l = G_1^l \wedge \cdots \wedge G_l^l$ define liveness assumptions and guarantees, respectively. For more details to GR(1), see Section 2.5.3.

According to the definition of GR(1), if the environment chooses at any point of time a single invalid input value, that violates a transition assumption, a non robust system can behave arbitrarily. Even if the environment works correctly for the rest of the time, the system do not have to fulfill any guarantees. We want to synthesize systems that are robust against safety assumption violations. This means that a robust system should be able to recover from safety assumption violations. If eventually, the environment stops making failures, the system should eventually work correctly again.

In order to define robustness, we introduce two kinds of errors, environment failures and system failures. The environment causes an **environment failure**, if it chooses an input values that violates at least one safety assumption. The system causes a **system failure**, if it chooses an output values that violates at least one safety guarantee.

We can formulate our robustness definition in LTL, by using env_{err} for a safety assumption violation, and sys_{err} for a safety guarantee violation.

Definition 3.1. (Robustness)
We define a system to be robust if a finite number of environment failures only induce a finite number of system failures [BGHJ09].

$$\mathsf{F}\,\mathsf{G}\,\neg env_{err} \rightarrow \mathsf{F}\,\mathsf{G}\,\neg sys_{err}. \tag{3.2}$$

Systems synthesized with our robust synthesis method satisfy this robustness criterion. This property can be rewritten by using the LTL equivalence $\mathsf{G}\,\neg\varphi = \neg\,\mathsf{F}\,\varphi$:

$$\neg\,\mathsf{F}\,\mathsf{G}\,\neg env_{err} \vee \mathsf{F}\,\mathsf{G}\,\neg sys_{err}. \tag{3.3}$$

$$\neg\neg(\mathsf{G}\,\mathsf{F}\,env_{err} \vee \mathsf{F}\,\mathsf{G}\,\neg sys_{err}). \tag{3.4}$$

$$\neg(\neg\,\mathsf{G}\,\mathsf{F}\,env_{err} \wedge \mathsf{G}\,\mathsf{F}\,sys_{err}). \tag{3.5}$$

$$(\mathsf{G}\,\mathsf{F}\,env_{err} \vee \neg\,\mathsf{G}\,\mathsf{F}\,sys_{err}). \tag{3.6}$$

$$(\mathsf{G}\,\mathsf{F}\,sys_{err} \rightarrow \mathsf{G}\,\mathsf{F}\,env_{err}). \tag{3.7}$$

We end up in the last equation, which *defines a system to be robust, if infinitely many environment failures imply infinitely many system failures.* The implication of the equation requires that the system is only allowed to make an infinite number of failures, if the environment also makes an infinite number of mistakes.

Clearly, this definition of robustness only considers the violation of safety properties, and not the violation of liveness properties. This approach does not work for liveness, because it is not possible to notice a violation of a liveness property at any point in time. On the other hand, we immediately notice a violation of a safety property.

3.3 Illustration of the problem

Consider the specification of a simple arbiter for a resource shared between two clients. The input signals r_1 and r_2 are used by the clients to request access to the resource. The arbiter grants access via the output signals g_1 and g_2.

The specification states that initially r_1 and g_1 are false and r_2 and g_2 are true. This results in the initial assumption $A^i = (\neg r_1 \wedge r_2)$ and the initial guarantee $G^i = (\neg g_1 \wedge g_2)$. Also, it is assumed that the environment never raises both request signals at the same time. We get $A^t = \mathsf{G}\,\neg(r_1 \wedge r_2)$. According to the specification, the system must fulfill the following requirements. First, the system is never allowed to raise both grant signals at the same time. In LTL syntax, this can be written as $G_1^t = \mathsf{G}\,\neg(g_1 \wedge g_2)$. Second, a request has to be followed immediately by a grant, which can be formalized by the guarantees $G_2^t = \mathsf{G}(r_1 \rightarrow \mathsf{X}\,g_1)$ and $G_3^t = \mathsf{G}(r_2 \rightarrow \mathsf{X}\,g_2)$. Combining the four guarantees and the two assumption results in the specification $\varphi = A^i \wedge A^t \rightarrow G^i \wedge G_1^t \wedge G_2^t \wedge G_3^t$. The specification requires the arbiter to satisfy all guarantees, if the assumptions are fulfilled.

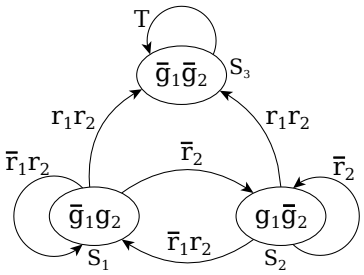


Figure 3.1: Non-robust Moore Machine.

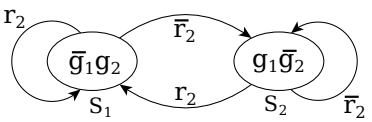


Figure 3.2: Robust Moore Machine.

One possible implementation of φ (in the form of a Moore Machine) is shown in Figure 3.1. The environment violates the transition assumption, if it raises r_1 and r_2 at the same time. This environment failure forces the machine to enter state S_3, and the machine will remain there forever. Irrespective of future inputs, both grant signals stay low. From this point on, whenever there is a request, there will be no grant in the next step and the system will make a system failure by violating G_2^t or G_3^t in the next step. This is not robust: a finite number of environment errors leads to an infinite number of system errors, i.e., the system does not

recover. Our new synthesis algorithm guarantees that this cannot happen. Instead, our approach may lead to an implementation as shown in Figure 3.2, which does not exhibit the aforementioned weakness. Now, if two requests occur simultaneously, one will be discarded while the other one will be granted. Once the environment resumes correct behavior, the system will also fulfill its guarantees again.

3.4 Example of Environment Failures and System Failures

To demonstrate our approach, we will use the GR(1) specification of a simple arbiter with a full-handshake protocol [PPS06].

The arbiter has a single Boolean request input signal r and a Boolean grant output signal g. Initially, the specification states that both request and input signal are low. We get, $A^i = (\neg r)$ and $G^i = (\neg g)$. For the environment, the safety assumptions $A_1^t = \mathsf{G}(r \wedge \neg g \rightarrow \mathsf{X}\, r)$ and $A_2^t = \mathsf{G}(\neg r \wedge g \rightarrow \mathsf{X}\, \neg r)$ are defined. The system also has two additional safety guarantees, $G_1^t = \mathsf{G}(\neg r \wedge \neg g \rightarrow \mathsf{X}\, \neg g)$ and $G_2^t = \mathsf{G}(r \wedge g \rightarrow \mathsf{X}\, g)$. The liveness assumption and guarantee are defined by the formulas $A^l = \mathsf{G}\,\mathsf{F}(\neg r \vee \neg g)$ and $G^l = \mathsf{G}\,\mathsf{F}((r \wedge g) \vee (\neg r \wedge \neg g))$. Combining the assumptions and the guarantees results in the specification $\varphi = A^i \wedge A_1^t \wedge A_2^t \wedge A^l \rightarrow G^i \wedge G_1^t \wedge G_2^t \wedge G^l$.

Figure 3.3 illustrates a cutout of a possible signal trace, without any violations of safety properties. First, both signals are low. At some point, there is a request, and r is set to true. One time step later, the system grants the request and rises g. After some time, the environment lowers the request, and the system responds by lowering the grant.

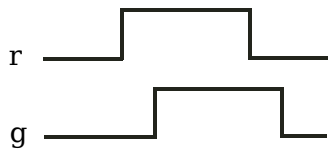


Figure 3.3: Possible Signal Trace, without Violation of Safety Properties.

In order to synthesize a robust arbiter, we first transform the specification into a one-pair Streett game.

Figure 3.4 illustrates the encoding of the safety assumptions $A^t = A_1^t \wedge A_2^t$ in the environment transition relation ρ_e of the game. The formula ρ_e defines the set of possible next inputs in respect to the present input and output. At all points in time holds that if the environment chooses a next input such that the formula ρ_e evaluates to true, then the environment did not make an *environment failure* in the current time step and the next environment error bit env'_{err} is set to false. If ρ_e evaluates to false, the environment made a failure and env'_{err} is

set to true. Let's assume, that the game is in the state where the request is low and the grant is high and the environment chooses $r = \mathsf{true}$ as next input value. This violates the formula ρ_e and env'_{err} is set to true. This can also be observed in Figure 3.4. There is no transition leading from the state with $r = \mathsf{false}$ and $g = \mathsf{true}$ to a state where $r = \mathsf{true}$.

Figure 3.5 illustrates the encoding of the safety guarantees $G^t = G^t_1 \wedge G^t_2$ in the system transition relation ρ_s of the game. The formula ρ_s defines the next outputs, also in respect to the current values of input and output. If the system chooses a next output such that ρ_s is violated, the system made a *system failure* and the next system error bit sys'_{err} is set to true. This fact can also be observed in Figure 3.5. If there is no transition from the current state to the next state specified by the next input and output values, then the system made a system error.

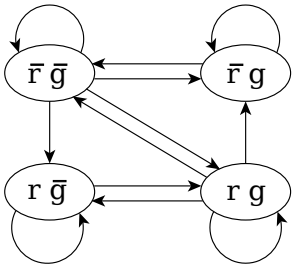


Figure 3.4: Environment Transitions Relation ρ_e of one-Pair Streett Game.

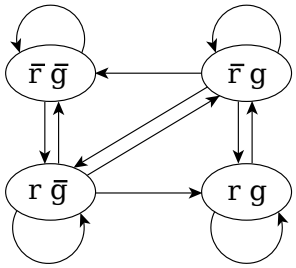


Figure 3.5: System Transitions Relation ρ_s of one-Pair Streett Game.

Figure 3.6 shows the combined transition relation $\rho_e \wedge \rho_s$ of the one-pair Streett game. This Figure contains only transitions, where neither system or environment have made a failure.

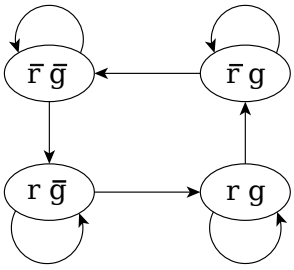


Figure 3.6: Transitions Relation $\rho_e \wedge \rho_s$ of one-Pair Streett Game.

To encode robustness into the game, we have to extend the state space by the two error bits env_{err} and sys_{err}. This results in a new state space $V^{\sim} = V \cup env_err \cup sys_err$ and in a new environment transition relation $\rho_e^{\sim} = \mathsf{true}$ and system transition relation $\rho_s^{\sim}$. The new system transition relation is defined as follows:

$\rho_1^{\sim} = \rho_e \wedge \rho_s \wedge \neg env'_{err} \wedge \neg sys'_{err}$
$\rho_2^{\sim} = \rho_e \wedge \neg\rho_s \wedge \neg env'_{err} \wedge sys'_{err}$
$\rho_3^{\sim} = \neg\rho_e \wedge \rho_s \wedge env'_{err} \wedge \neg sys'_{err}$
$\rho_4^{\sim} = \neg\rho_e \wedge \neg\rho_s \wedge env'_{err} \wedge sys'_{err}$
$\rho_s^{\sim} = \rho_1^{\sim} \vee \rho_2^{\sim} \vee \rho_3^{\sim} \vee \rho_4^{\sim}$

The formula $\rho_s^{\sim}$ keeps track of all environment failures and system failures by setting the next values of the environment error bit env_{err} and the system error bit sys_{err}.

3.5 Robust Synthesis Algorithm

Given are:

- a GR(1) specification: $\varphi = A^i \wedge A^t \wedge A^l \rightarrow G^i \wedge G^t \wedge G^l$ and
- the robustness criterion: $\mathsf{G}\,\mathsf{F}\, sys_{err} \rightarrow \mathsf{G}\,\mathsf{F}\, env_{err}$.

The algorithm for synthesizing robust system consists of several steps.

First, *the GR(1) specification is transformed into a GR(1) game*, see Section 3.5.1. The initial properties (A^i, G^i) define the initial states of the game structure (defined by Θ). The transition properties (A^t, G^t) are encoded directly into the transition relation of the GR(1) game (ρ_e,ρ_s), and the liveness properties (A^l, G^l) form the winning condition ($\varphi = A^l \rightarrow G^l$).

In the second step, *the GR(1) game is transformed into a one-pair Streett game*, by applying a *counting construction*, see Section 3.5.2. The winning condition of the GR(1) game is transformed into a Streett pair, called *Correctness-Streett Pair*. The Correctness Pair ensures that the liveness part of the specification is encoded properly in the game. An implementation of this Streett game would be a correct implementation of the specification.

Next, the robustness criterium has to be encoded into the game, see Section 3.5.3. This is done by *adding a second-Streett pair*, called the *Robustness Pair*. The state space is extended by two additional bits: env_{err} encodes an environment failure and sys_{err} encodes a system failure. Using these variables, the robustness criterion $\mathsf{G}\,\mathsf{F}\, sys_{err} \rightarrow \mathsf{G}\,\mathsf{F}\, env_{err}$ can be encoded by the second-Streett pair: $\langle (sys_{err}), (env_{err}) \rangle$.

Finally, we compute the *winning region*, see Section 3.5.4 and the winning strategy, see Section 3.5.5. The winning strategy for the two-pair Streett game corresponds to a correct and robust implementation of the original GR(1) specification.

Figure 3.7 summarizes the steps necessary to obtain a robust system.

3.5.1 GR(1) Specification to GR(1) Game

The first step of synthesizing a robust reactive system, is to transform the GR(1) specification into a GR(1) Game [PPS06].

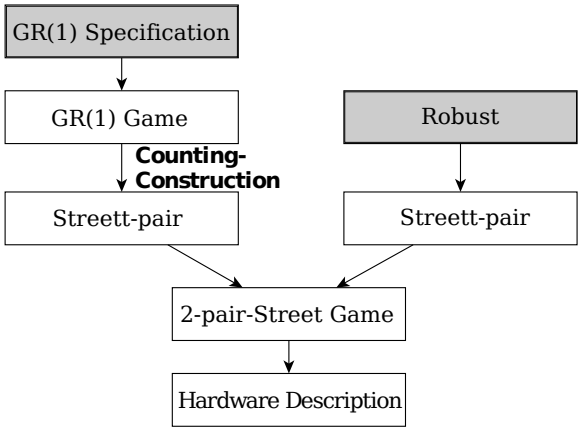


Figure 3.7: Robust System Generation Process.

Let $\varphi = \varphi_e \rightarrow \varphi_s$ be a GR(1) formula with $\varphi_\alpha = \varphi_i^\alpha \wedge \varphi_t^\alpha \wedge \varphi_l^\alpha$ for $\alpha \in \{e, s\}$. X denotes the set of input variables, and Y the set of output variables.

A game structure $G : (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$ can be constructed as follows:

- $V = X \cup Y$, X be the set of input variables, Y be the set of output variables.
- $\Theta = \varphi_i^e \wedge \varphi_i^s$.
- Let $\varphi_t^\alpha = \bigwedge_j \mathsf{G}\, q_j$, where q_j is a Boolean combination of input and output variables and expressions of the form $\mathsf{X}\, v$ with $v \in X$ if $\alpha = e$ and $v \in X \cup Y$ if $\alpha = s$.
  Then we get $\rho_\alpha = \bigwedge_j \tau(q_j)$, where τ replaces each occurrence of $\mathsf{X}\, v$ by v'.
- We get the winning condition $\varphi = \varphi_l^e \rightarrow \varphi_l^s$.

3.5.2 GR(1) Game to one-pair Streett Game

For the second step, we use the algorithm from Bloem et al. [BCG+10a], to transform a GR(1) game into a one-pair Streett Game. In our setting, we call the Streett pair resulting from this step the *Correctness Pair*.

The algorithm to reduce games with Generalized Reactivity objectives to games with Streett objectives, is called *counting construction*.

Reduction 3.1. (Counting Construction)
Let $G : (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$ be a game structure with the winning condition $\varphi = \bigwedge_{k=1}^{m} \mathsf{G}\,\mathsf{F}\, A_k \rightarrow \bigwedge_{l=1}^{n} \mathsf{G}\,\mathsf{F}\, G_l$. We define i to be a variable which is able to count from 0 to m, and the counter j is able to count from 0 to n. We construct an equivalent one-pair Streett game $G^\sim : (V^\sim, X^\sim, Y^\sim, \Theta^\sim, \rho_e^\sim, \rho_s^\sim, \varphi^\sim)$:

1. The state space:
   $V^{\sim} = V \cup i \cup j$,
   $X^{\sim} = X$,
   $Y^{\sim} = Y \cup i \cup j$,
   $\Sigma^{\sim} = 2^{X^{\sim} \cup Y^{\sim}}$.

2. The initial states Θ':
   $\Theta^{\sim} = \Theta \wedge i = 0 \wedge j = 0$.

3. The environment transition relation $\rho_e^{\sim}$:
   $((s, i, j), x') \models \rho_e^{\sim}$ if $s \in \Sigma, x' \in D_X, (s, x') \models \rho_e, 0 \leq i \leq m, 0 \leq j \leq n$

4. The system transition relation: ρ_s' :
   $((s, i, j), (s', i', j')) \models \rho_s^{\sim}$ if $s, s' \in \Sigma, (s, s') \models \rho_e \wedge \rho_s$ and if $s' \in A_{i+1}$ then $i' = i + 1$ else $i' = i$, if $s' \in G_{j+1}$ then $j' = j + 1$ else $j' = j$
   $((s, m, j), (s', 0, j')) \models \rho_s^{\sim}$ if $s, s' \in \Sigma, (s, s') \models \rho_e \wedge \rho_s, j \neq n$, if $s' \in G_{j+1}$ then $j' = j + 1$ else $j' = j$
   $((s, i, n), (s', 0, 0)) \models \rho_s^{\sim}$ if $s, s' \in \Sigma, (s, s') \models \rho_e \wedge \rho_s$ and for $0 \leq i \leq m$

5. The winning condition φ:
   $\varphi = \mathsf{GF}\, A^{\sim} \rightarrow \mathsf{GF}\, G^{\sim}$, with $A^{\sim} = \{(s, m, j) \in \Sigma^{\sim} : j \in \{0, \ldots, n\}\}$ and $G^{\sim} = \{(s, i, n) \in \Sigma^{\sim} : i \in \{0, \ldots, m\}\}$

We extend the state space by two counters i and j. For m liveness assumptions $\mathsf{GF}\, A_k$ (with $1 \leq k \leq m$) and n liveness guarantees $\mathsf{GF}\, G_l$ (with $1 \leq l \leq n$), the state-space is extended with two counters $i \in \{0, \ldots m\}$ and $j \in \{0, \ldots n\}$, which can be encoded with $\lceil \log_2(m+1) \rceil + \lceil \log_2(n+1) \rceil$ additional bits.

The counter i indicates which assumption was visited last and j which guarantee was visited last. i is incremented modulo $m + 1$ whenever the actual assumption is satisfied. Similarly for j, G_j is incremented modulo $n + 1$, whenever the actual guarantee is visited. It also takes the sequential arrangement of the traversal of the assumptions and guarantees into account. At the beginning both counters are set to 0. Counter i is incremented the first time, when A_1 is visited. The next increment of i is performed when A_2 is visited, etc. If all assumptions were visited in the right order, $i = m$ and i is going to be reset in the next step. The same principle is applied to the guarantees. If counter j reaches the value of n, all guarantees have been visited in a row and both counter i and j are going to be reset in the next step.

The set $A^{\sim} = \{(s, m, j) \in \Sigma^{\sim}\}$ is the set of states which have fulfilled all assumptions in a row and the set $G^{\sim} = \{(s, i, n) \in \Sigma^{\sim}\}$ defines all states which have fulfilled all guarantees in a row. This results in the Correctness-Streett Pair: $\langle A^{\sim}, G^{\sim} \rangle$, which ensures that the liveness part of the specification is encoded properly in the game. A winning strategy for this game corresponds to a correct implementation.

For simplicity, we write $\varphi = (\mathsf{GF}\, i = m) \rightarrow (\mathsf{GF}\, j = n)$ as the winning strategy. We get the *Correctness-Pair* by:

$$\varphi = \langle (i = m), (j = n) \rangle \tag{3.8}$$

Example 3.1. (Counting Construction)
This example demonstrates, how the counting construction works.
Let $G : (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$ be a game structure, see Figure 3.8. The game structure consists of four reachable states: $S_1 \dots S_4$. The winning condition is a Generalized Reactivity winning condition:

$$\mathsf{G}\,\mathsf{F}\,A_1 \wedge \mathsf{G}\,\mathsf{F}\,A_2 \rightarrow \mathsf{G}\,\mathsf{F}\,G_1 \wedge \mathsf{G}\,\mathsf{F}\,G_2 \tag{3.9}$$

with $A_1 = \{S_2\}$, $A_2 = \{S_3\}$, $G_1 = \{S_2\}$ and $G_2 = \{S_1\}$.
Applying the counting construction results in a new game graph, as shown in Figure 3.9.

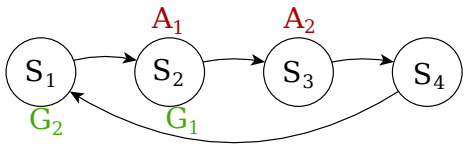


Figure 3.8: Game Graph before Applying Counting Construction.

This graph again shows only the states that are reachable. Each state is extended by two counters. The first counter is the assumption counter, and the second one is the guarantee counter. Since we have two assumptions and two guarantees, each counter has to be able to count from 0 to 2, and needs 2 bits.

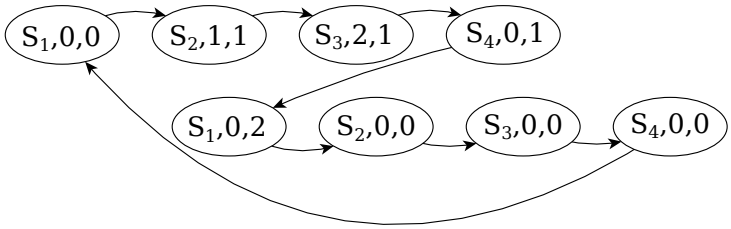


Figure 3.9: Game Graph after applying Counting Construction.

3.5.3 Robustness Streett pair

So far, we transformed the GR(1) specification into a one-pair Streett game. What's left is to encode the robustness property into the game.

In order to obtain a system which is also robust, we extend the state-space of the Streett game by two additional Boolean variables env_{err} and sys_{err}. These variables keep track of the current error state of the environment and the system. Initially, both env_{err} and sys_{err} are set to false, so no error has happened so far. The variable env_{err} is set to true whenever the environment makes an environment failure, sys_{err} is set to true if a system error occurs. If the environment or the system stops making errors, the corresponding variable is flipped to false again.

Formally, we have the following:
Let $G : (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$ be a game structure and let the game $\bar{\sigma}$ be in a state $s \in \Sigma$, the environment chooses the next input value $x' \in D_X$ and the system chooses the next output value $y' \in D_Y$. env'_{err} and sys'_{err} denote the next values of the error variables.

- If $(s, x') \not\models \rho_e$ and $(s, (x', y')) \not\models \rho_s$ than $env'_{err} = \mathsf{true}$ and $sys'_{err} = \mathsf{true}$.
- If $(s, x') \models \rho_e$ and $(s, (x', y')) \not\models \rho_s$ than $env'_{err} = \mathsf{false}$ and $sys'_{err} = \mathsf{true}$.
- If $(s, x') \not\models \rho_e$ and $(s, (x', y')) \models \rho_s$ than $env'_{err} = \mathsf{true}$ and $sys'_{err} = \mathsf{false}$.
- If $(s, x') \models \rho_e$ and $(s, (x', y')) \models \rho_s$ than $env'_{err} = \mathsf{false}$ and $sys'_{err} = \mathsf{false}$.

Our definition of robustness can be formulated using these variables. A system is robust if

$$\mathsf{G}\,\mathsf{F}\, sys_{err} \rightarrow \mathsf{G}\,\mathsf{F}\, env_{err}. \tag{3.10}$$

This can be expressed by the *Robustness Pair*:

$$\varphi = \langle (sys_{err}), (env_{err}) \rangle. \tag{3.11}$$

Reduction 3.2. (Translation of one-pair Streett game to robust two-pair Streett game)
Given is a one-pair Streett game $G : (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$ with $\varphi = \{\langle A, G \rangle\}$. Let sys_{err} and env_{err} be two Boolean variables and sys'_{err} and env'_{err} the next value of the variables. We construct a robust two-pair Streett game $G^{\sim} : (V^{\sim}, X^{\sim}, Y^{\sim}, \Theta^{\sim}, \rho_e^{\sim}, \rho_s^{\sim}, \varphi^{\sim})$:

1. The state space:
   $V^{\sim} = V \cup env_{err} \cup sys_{err}$,
   $X^{\sim} = X$,
   $Y^{\sim} = Y \cup env_{err} \cup sys_{err}$,
   $\Sigma^{\sim} = 2^{X^{\sim} \cup Y^{\sim}}$.

2. The initial states $\Theta^{\sim}$:
   $\Theta^{\sim} = \Theta \wedge \neg env_{err} \wedge \neg sys_{err}$.

3. The environment transition relation $\rho_e^{\sim}$:
   $\rho_e^{\sim} = \mathsf{true}$

4. The system transition relation: ρ'_s :
   $\rho_1^{\sim} = \rho_e \wedge \rho_s \wedge \neg env'_{err} \wedge \neg sys'_{err}$
   $\rho_2^{\sim} = \rho_e \wedge \neg \rho_s \wedge \neg env'_{err} \wedge sys'_{err}$
   $\rho_3^{\sim} = \neg \rho_e \wedge \rho_s \wedge env'_{err} \wedge \neg sys'_{err}$
   $\rho_4^{\sim} = \neg \rho_e \wedge \neg \rho_s \wedge env'_{err} \wedge sys'_{err}$
   $\rho_s^{\sim} = \rho_1^{\sim} \vee \rho_2^{\sim} \vee \rho_3^{\sim} \vee \rho_4^{\sim}$

5. The winning condition $\varphi^{\sim}$:
   $\varphi^{\sim} = (\mathsf{G}\,\mathsf{F}\, A \rightarrow \mathsf{G}\,\mathsf{F}\, G) \wedge (\mathsf{G}\,\mathsf{F}(sys_{err}) \rightarrow \mathsf{G}\,\mathsf{F}(env_{err}))$

According to the new environment transition relation $\rho_e^{\sim}$, the environment is allowed to make environment failures by choosing next input values that violate the initial transition relation ρ_e. The system keeps track of the violations of ρ_e, by changing the value of the environment error variable env'_{err} to true. Also the system is allowed to make system failures by violating the initial system transition relation ρ_s. In this case, sys'_{err} is set to true. In order to win the game, the system is not allowed to make an infinite number of errors, if the environment doesn't.

Another way to encode the system transition relation $\rho_s^{\sim}$ would be:
$\rho_1^{\sim} = \rho_e \wedge \rho_s \wedge \neg env'_{err} \wedge \neg sys'_{err}$
$\rho_2^{\sim} = \rho_e \wedge \neg\rho_s \wedge \neg env'_{err} \wedge sys'_{err}$
$\rho_3^{\sim} = \neg\rho_e \wedge \rho_s \wedge env'_{err} \wedge \neg sys'_{err}$
$\rho_4^{\sim} = \neg\rho_e \wedge \neg\rho_s \wedge \neg env'_{err} \wedge \neg sys'_{err}$
$\rho_s^{\sim} = \rho_1^{\sim} \vee \rho_2^{\sim} \vee \rho_3^{\sim} \vee \rho_4^{\sim}$

It differs only in the last formula $\rho_4^{\sim}$. If an environment error occurs, it is ok for the system to respond with a system error, according to the definition of robustness. Since this is ok, there is no need to set an error bit. This way, states with both $env_{err} = 1$ and $sys_{err} = 1$ are no longer reachable.

The first street pair $\langle A, G\rangle$ was obtained from the GR(1) specification. It implies that if all liveness assumptions A are fulfilled infinitely often, all liveness guarantees G also have to be fulfilled infinitely often. If this condition is fulfilled, the automatically synthesized system works correctly.

The robustness property is encoded in the second Streett pair $\langle(sys_{err}), (env_{err})\rangle$. The second pair implies that if a state with a system error is visited infinitely often, a state with an environment failure also has to be visited infinitely often.

Winning the game is only possible by fulfilling both pairs. Therefore the winning strategy of the two-pair Streett game corresponds to a robust and correct implementation.

3.5.4 Winning Region

The next step is to compute the winning region W_s for player 0 of the two-pair Streett game.

In [PP06], Piterman and Pnueli introduced a recursive fixpoint algorithm to compute the winning region for any number of Streett pairs. It is a symbolic algorithm and therefore we can use efficient symbolic fixpoint computations. By keeping intermediate values during the fixpoint calculation, Streett games can be solved symbolically in time $O(n^{k+1}k!)$ and space $O(n^{k+1}k!)$ where n ist the number of states of the game and k is the number of pairs in the winning condition. In our case, we want to solve Streett games with two pairs. In case of $k = 2$, we can solve such games in $O(n^3)$ time and $O(n^3)$ space.

Let $G = (V, X, Y, \Theta, \rho_e, \rho_s, \varphi)$ be a game structure and $\bar{\sigma}$ an infinite play. The winning condition φ is given by $\varphi = \{\langle E_1, F_1\rangle, \langle E_2, F_2\rangle, \ldots, \langle E_k, F_k\rangle\}$ with $E_i, F_i \subseteq \Sigma$. The winning

condition is as follow:

$$\varphi(\bar{\sigma}) \Leftrightarrow \bigwedge_{i=1}^{k} (\mathsf{inf}(\bar{\sigma}) \cap E_i \neq \emptyset \rightarrow \mathsf{inf}(\bar{\sigma}) \cap F_i \neq \emptyset) \tag{3.12}$$

First, the algorithm chooses the Streett pair $\langle E_1, F_1 \rangle$, and collects all states that can visit F_1 infinitely often. If F_1 is visited infinitely often, the Streett pair is satisfied, no matter if E_1 is visited infinitely often or not. Then the algorithm adds all states that visit E_1 only finitely often, while making sure recursively that all other pairs $\langle E_2, F_2 \rangle, \ldots, \langle E_k, F_k \rangle$ are satisfied. In the next step, the algorithm chooses the pair $\langle E_2, F_2 \rangle$, collects all states that satisfies this pair, and makes sure recursively that the others are satisfied as well....

We used this algorithm in our implementation, shown in Algorithm 1.

```
input  : A set Set of Streett pairs {⟨a_1, b_1⟩ ... ⟨a_k, b_k⟩}.
output : The winning region for Streett game with k pairs.
1 begin
2 |   if |Set|=0 then
3 |   |   return m_Streett(true,false);
4 |   end
5 |   return m_Streett(Set,true,false);
6 end
```

Algorithm 1: `main_Streett`: Main Function to compute Winning Region

The function $\diamondsuit(X)$ returns the set of states from which the system can force the play into X in one step. `LeastFix` and `GreatestFix` represent least and greatest fixpoint computations over sets of states. The loop `LeastFix(X)` initializes `X` by the empty set, and is repeated until the set of states computed for `X` does not change any more. The loop `GreatestFix(X)` initializes `X` by the set of all states, and is repeated until two rounds calculate the same set of states for `X`. The operators $\cap$ and $\cup$ perform intersection and union of sets of states. Let $\langle a, b \rangle$ be a Streett pair. Then a and b are sets of states, and $\bar{a}$ and $\bar{b}$ are their complements.

To compute the winning region for player 0 according to the Streett winning condition, the function `main_Streett` is called. The input Set is a set of Streett pairs: $Set = \langle a_1, b_1 \rangle, \ldots, \langle a_k, b_k \rangle$. The output is a set of states: the winning region for player 0. To compute the winning region, the function `main_Streett` calls the function `Streett`.

The function `Streett` is called recursively for all Streett pairs $\langle a_1, b_1 \rangle \ldots \langle a_k, b_k \rangle$. Suppose that the function `Streett` was already called i times with $1 \leq i \leq k$ and the parameter `Set` consists of the Streett pairs $\langle a_{i+1}, b_{i+1} \rangle, \ldots, \langle a_k, b_k \rangle$. In this case, the input parameter φ defines the set of states that eventually avoid states in $a_1 \cup \cdots \cup a_i$. The input parameter W defines states that can visit states in b_j for $j = 1 \ldots i$ infinitely often while satisfying all previous handled pairs.

If the function `Streett` reaches recursion depth k, it has processed all Streett pairs and the function `Streett` calls the function `m_Streett`. The input parameters φ and W of the function `m_Streett` have the same meaning as in the function `Streett`.

input : A set Set of Streett pairs $\{\langle a_1, b_1\rangle \ldots \langle a_k, b_k\rangle\}$, and sets of states φ and W.
output : The winning region Z for a game with winning condition
$(\varphi \,\mathsf{U}\, W) \bigwedge_{\langle a,b\rangle} [\mathsf{G}(\varphi \wedge \mathsf{F}\, b) \vee [\varphi \,\mathsf{U}\, (\mathsf{G}(\varphi \wedge \bar{a}) \wedge ltlStr(S - \langle a, b\rangle))]]$

```
1  begin
2      GreatestFix(Z)
3          foreach ⟨a, b⟩ ∈ Set do
4              nSet = Set − ⟨a, b⟩ ;
5              p1 = W ∪ φ ∩ r ∩ ⦻(Z);
6              LeastFix(Y)
7                  p2 = p1 ∪ φ ∩ ⦻(Y);
8                  if |nSet| = 0 then
9                      Y = m_Streett(φ ∩ ḡ, p2);
10                 else
11                     Y = m_Streett(nSet, φ ∩ ḡ, p2);
12                 end
13             end
14             Z = Y;
15         end
16     end
17     return Z ;
18 end
```

Algorithm 2: `Streett`: Recursive Function to compute Winning Region

input : Sets of states φ and W.
output : Winning region for game with winning condition: $(\varphi \,\mathsf{U}\, W) \vee \mathsf{G}\, \varphi$.

```
1  begin
2      GreatestFix(X)
3          X = W ∪ φ ∩ ⦻(X);
4      end
5      return X ;
6  end
```

Algorithm 3: `m_Streett`: Helper Function to compute Winning Region

We define $ltlStr(Set) = \bigwedge_{\langle a,b\rangle \in Set}(\mathsf{G}\,\mathsf{F}\,a \rightarrow \mathsf{G}\,\mathsf{F}\,b)$.

Lemma 3.1. *[PP06] The function* `m_Streett(`φ`,`W`)` *computes a set of state according to the formula:*

$$\nu X(W \vee \varphi \wedge \textcircled{\scriptsize }(X)). \tag{3.13}$$

This set corresponds to the winning region for player 0 in the game with winning condition:

$$(\varphi\,\mathsf{U}\,W) \vee \mathsf{G}\,\varphi. \tag{3.14}$$

The set of states which is calculated by the fixpoint-formula is exactly the same set of states satisfying the winning condition.

Lemma 3.2. *[PP06] The function* `Streett(`$\langle a,b\rangle$`,`φ`,`W`)` *computes a set of states according to the formula:*

$$\nu Z \mu Y(m\_Streett(\varphi \wedge \bar{a}, (W \vee (\varphi \wedge b \wedge \textcircled{\scriptsize }Z) \vee (\varphi \wedge \textcircled{\scriptsize }Y)))). \tag{3.15}$$

This set corresponds to the winning region for player 0 in the game with winning condition:

$$win(\langle a,b\rangle, \varphi, W) = (\varphi\,\mathsf{U}\,W) \vee \mathsf{G}(\varphi \wedge \mathsf{F}\,b) \vee (\varphi\,\mathsf{U}\,\mathsf{G}(\varphi \wedge \bar{a})). \tag{3.16}$$

A play $\bar{\sigma}$ is winning according the winning condition $win(\langle a,b\rangle, \varphi, W)$, if the play stays in φ-states, until it reaches a W-state, or it visits a-states finitely often, or in b-states infinitely often, both while staying in φ-states.

Following, we sketch the proof for soundness of 3.2. The proof for completeness can be found in [PP06]. Let $\hat{Z}$ be the set computed by the greatest fixpoint and let Y_i be the iterates of the least fixpoint using $\hat{Z}$. We define the **rank** r of a state v: $r(v) = \min\{i : v \in Y_i\}$. By using Lemma 3.1, the iterates for the fixpoint Y using $\hat{Z}$ are defined by:

$$Y_0 = \emptyset, \tag{3.17}$$

$$Y_1 = \mathsf{G}(\varphi \wedge \bar{a}) \vee [(\varphi \wedge \bar{a})\,\mathsf{U}\,(W \vee (\varphi \wedge b \wedge \textcircled{\scriptsize }\hat{Z}))], \text{ and} \tag{3.18}$$

$$Y_{i+1} = \mathsf{G}(\varphi \wedge \bar{a}) \vee [(\varphi \wedge \bar{a})\,\mathsf{U}\,(W \vee (\varphi \wedge b \wedge \textcircled{\scriptsize }\hat{Z}) \vee (\varphi \wedge \textcircled{\scriptsize }Y_i))]. \tag{3.19}$$

For any state with rank $r(v) = i$ with $i \geq 1$, there exists a sub-strategy for player 0. If there are n iterates of the fixpoint Y, than there are n sub-strategies for player 0. In a state v with $r(v) = i$, player 0 plays according to sub-strategy i. By combining these sub-strategies, we are able to prove the soundness of Lemma 3.2.

First we consider Equation 3.18. Suppose, the play $\bar{\sigma}$ is in state v in Y_1: $r(v) = 1$.
Sub-strategy 1 is defined as follows:
Player 0 can either stay in $(\varphi \wedge \bar{a})$-states forever, or he can stay in $\varphi \wedge \bar{a}$ states until he reaches a W-state, or he can stay in $\varphi \wedge \bar{a}$ states until he reaches a $(\varphi \wedge b \wedge \textcircled{\scriptsize }\hat{Z})$-state.

Now, we consider Equation 3.19. The play $\bar{\sigma}$ is in state v in Y_i, $i > 1$, i.e. $r(v) = i$.
Sub-strategy i for $i > 1$ is defined as follows:

Player 0 can either do the same things as defined in sub-strategy 1 or he can stay in $(\varphi \wedge \bar{a})$-states until he reaches a state with a lower rank.

If a play $\bar{\sigma}$ switches infinitely often between the sub-strategies, it visits states in $(\varphi \wedge b \wedge \bigcirc\hat{Z})$ infinitely often. While player 0 is playing according to a sub-strategy, the ranks of the states become lower and lower, and he gets closer and closer to states in $(\varphi \wedge b \wedge \bigcirc\hat{Z})$ with rank 0. If he reaches such a state, he can choose any successor state in $\hat{Z}$ with arbitrary rank. Afterwards, he again tries to reach a state with rank 0, etc. The play $\bar{\sigma}$ satisfies the formula $\mathsf{G}(\varphi \wedge \mathsf{F}\, b)$ and thereby is a winning play according to $win(\langle a, b\rangle, \varphi, W)$.

If a play $\bar{\sigma}$ switches only finitely often between the strategies, the play stays in some Y_i from this point on. From this point on, the play plays according to the i^{th} sub-strategy and stays in states in $(\varphi \wedge \bar{a})$ forever. The play satisfies the formula $(\varphi \,\mathsf{U}\, \mathsf{G}(\varphi \wedge \bar{a}))$ and thereby is winning according to $win(\langle a, b\rangle, \varphi, W)$. This proves the soundness of Lemma 3.2. □

To illustrate what we have just proofed we solve a Streett game with only one pair. In this "simple "we can make further simplifications. If there is only one Streett pair $\langle a, b\rangle$, then we are in the top-level call of the function `Streett` and we can set φ to true and W to false. This leads to the winning condition of the function `Streett`:

$$win(\langle a, b\rangle, \mathsf{true}, \mathsf{false}) = \mathsf{G}\,\mathsf{F}(b) \vee \mathsf{F}\,\mathsf{G}(\neg a) \tag{3.20}$$

This is exactly the winning condition for a one pair Streett game:

$$win(\langle a, b\rangle, \mathsf{true}, \mathsf{false}) = \neg\neg\,\mathsf{F}\,\mathsf{G}(\neg a) \vee \mathsf{G}\,\mathsf{F}(b) \tag{3.21}$$

$$win(\langle a, b\rangle, \mathsf{true}, \mathsf{false}) = \neg\,\mathsf{G}\,\mathsf{F}(a) \vee \mathsf{G}\,\mathsf{F}(b) \tag{3.22}$$

$$win(\langle a, b\rangle, \mathsf{true}, \mathsf{false}) = \mathsf{G}\,\mathsf{F}(a) \rightarrow \mathsf{G}\,\mathsf{F}(b) \tag{3.23}$$

So the function `Streett(`$\langle a, b\rangle$`,true,false)` computes the winning region for player 0 with a one-pair Streett winning condition.

This can easily be illustrated by simplification of the iterates. We get:

$$Y_0 = \emptyset. \tag{3.24}$$

$$Y_1 = \mathsf{G}(\bar{a}) \vee (\bar{a}\,\mathsf{U}\,(b \wedge \bigcirc\hat{Z})). \tag{3.25}$$

$$Y_{i+1} = \mathsf{G}\,\bar{a} \vee (\bar{a}\,\mathsf{U}\,(b \wedge \bigcirc\hat{Z} \vee \bigcirc Y_i)). \tag{3.26}$$

Suppose the play $\bar{\sigma}$ is in state v with $r(v) = 1$.
The play plays according to sub-strategy 1. Sub-strategy 1 defines that for each state in Y_1 the game either stays in $\bar{a}$-states forever, or the play stays in $\bar{a}$-states, until it reaches a $(b \wedge \bigcirc\hat{Z})$-state. Figure 3.10 illustrates sub-strategy 1.

Let's assume the play $\bar{\sigma}$ is in state v with $r(v) = i + 1$ for $i \geq 1$.
While playing according to sub-strategy $i + 1$, player 0 can either stay in $\bar{a}$-states, or he stays in $\bar{a}$-states, until it reaches a state in $(b \wedge \bigcirc\hat{Z})$ or a state in $\bigcirc Y_i$. Figure 3.11 illustrates the sub-strategy $i + 1$.

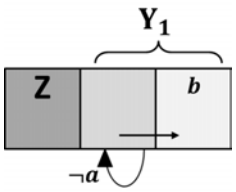


Figure 3.10: Strategy for States in Y_1 for one-Pair Streett Game.

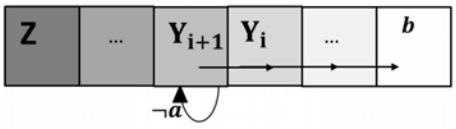


Figure 3.11: Strategy for States in Y_{i+1} for a one-Pair Streett Game.

This is shown in Figure 3.11.

If a play $\bar{\sigma}$ switches infinitely often between the strategies, then b-states are visited infinitely often. If a play $\bar{\sigma}$ switches only finitely often between the strategies, than a-states are also only visited finitely often. The play is winning according to the Streett-winning condition $\mathsf{G}\,\mathsf{F}\,a \rightarrow \mathsf{G}\,\mathsf{F}\,b$.

Lemma 3.3. *[PP06] Let Set be a set of Streett pairs $\{\langle a_1, b_1\rangle, \ldots \langle a_k, b_k\rangle\}$. The function* `Streett(Set,φ,W)` *computes a set of states according to the formula:*

$$\nu Z \mu Y [Streett(Set - \langle a_1, b_1\rangle, \varphi \wedge \bar{a_1}, (W \vee (\varphi \wedge b_1 \wedge \langle\!\langle\rangle\!\rangle Z) \vee (\varphi \wedge \langle\!\langle\rangle\!\rangle Y)))]. \tag{3.27}$$

This set corresponds to the winning region for player 0 in the game with winning condition:

$$win(\langle a, b\rangle, \varphi, W) = (\varphi \mathsf{U} W) \vee \bigwedge_{\langle a,b\rangle \in Set} [\mathsf{G}(\varphi \wedge \mathsf{F}\,b) \vee (\varphi \mathsf{U} (\mathsf{G}(\varphi \wedge \bar{a}) \wedge ltlStr(Set - \langle a, b\rangle)))]. \tag{3.28}$$

Let's assume Lemma 3.3 is proved for i Streett pairs. The induction step is done, by proving Lemma 3.3 for $i+1$ Streett pairs. Let $S' = Set - \langle a, b\rangle$. By induction, we get for every Streett pair $\langle a, b\rangle \in Set$ the iterates:

$$Y_0 = \emptyset. \tag{3.29}$$

$$Y_1 = \varphi \,\mathsf{U}\, (W \vee (\varphi \wedge b \wedge \langle\!\langle\rangle\!\rangle \hat{Z}))$$
$$\bigwedge_{\langle a',b'\rangle \in S'} [\mathsf{G}(\varphi \wedge \bar{a} \wedge \mathsf{F}\,b') \vee [(\varphi \wedge \bar{a}) \,\mathsf{U}\, (\mathsf{G}(\varphi \wedge \bar{a}) \wedge ltlStr(S' - \langle a', b'\rangle))]]. \tag{3.30}$$

$$Y_{i+1} = \varphi \,\mathsf{U}\, ((\varphi \wedge \langle\!\langle\rangle\!\rangle Y_i) \vee W \vee (\varphi \wedge b \wedge \langle\!\langle\rangle\!\rangle \hat{Z}))$$
$$\bigwedge_{\langle a',b'\rangle \in S'} [\mathsf{G}(\varphi \wedge \bar{a} \wedge \mathsf{F}\,b') \vee [(\varphi \wedge \bar{a}) \,\mathsf{U}\, (\mathsf{G}(\varphi \wedge \bar{a}) \wedge ltlStr(S' - \langle a', b'\rangle))]]. \tag{3.31}$$

First we consider Equation 3.30. Suppose, the play $\bar{\sigma}$ is in state v with $r(v) = 1$.
Sub-strategy 1 is defined as follows:
Player 0 can either stay in φ-states until he reaches a W-state or a state in $\varphi \wedge b \wedge \hat{\Diamond} \hat{Z}$ or the play is infinite and remains in $(\varphi \wedge \bar{a})$-states while satisfying the remaining Streett pairs.

Let's look at Equation 3.31. The play $\bar{\sigma}$ is in state v with $r(v) = i + 1$.
Sub-strategy $i + 1$ is defined as follows:
Player 0 can either do the same things as defined by sub-strategy 1 or he can stay in φ-states until he reaches a state with lower rank, while staying in φ-states.

If a play $\bar{\sigma}$ switches infinitely often between the strategies, it visits states in $(\varphi \wedge b \wedge \hat{\Diamond} \hat{Z})$ infinitely often. While player 0 is playing according to some sub-strategy, the ranks of the states decrease. If he reaches a state in $(\varphi \wedge b \wedge \hat{\Diamond} \hat{Z})$, he can choose any successor state in $\hat{Z}$ with arbitrary rank. The play $\bar{\sigma}$ satisfies the formula $\mathsf{G}(\varphi \wedge \mathsf{F}\, b)$ and thereby is a winning play according to $win(Set, \varphi, W)$.

If a play $\bar{\sigma}$ switches only finitely often between the strategies, the play stays in Y_i from this point on. From this point on, the play plays according to the sub-strategy i and satisfies the following formula $\varphi \,\mathsf{U}\, \bigwedge_{\langle a,b\rangle \in Set} [\mathsf{G}(\varphi \wedge \mathsf{F}\, b) \vee (\varphi \,\mathsf{U}\, (\mathsf{G}(\varphi \wedge \bar{a}) \wedge (ltlStr(Set - \langle a, b\rangle))))]$ which implies $\varphi \,\mathsf{U}\, (ltlStr(S') \wedge \mathsf{G}\, \varphi)$. This proves the soundness of Lemma 3.3.

3.5.5 Winning Strategy

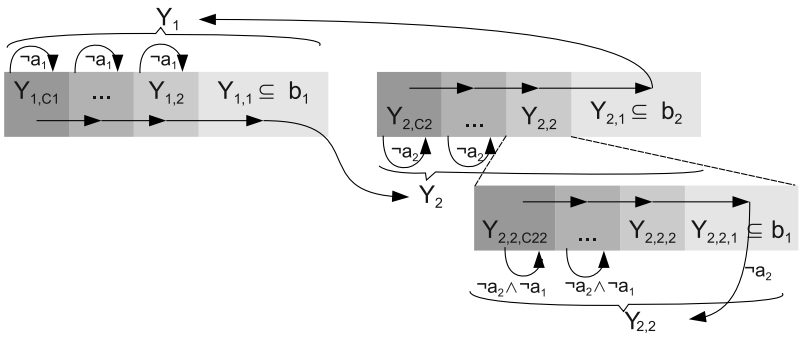


Figure 3.12: Illustration of the Iterates of the Fixpoint Computation.

The following discussion assumes that `Set`=$\{\langle a_1, b_1\rangle, \langle a_2, b_2\rangle\}$. Let Y_1 be the fixpoint in Y for the first Streett pair in the top-level call to `Str`. Y_2 is the result for the second pair. We denote the iterates of these fixpoint computations by $Y_{1,0} \ldots Y_{1,C_1}$ and $Y_{2,0} \ldots Y_{2,C_2}$. For both Streett pairs, the function `Str` is called recursively. The iterates of Y in the recursive call during the computation of $Y_{i,j}$ are denoted $Y_{i,j,0} \ldots Y_{i,j,C_{i,j}}$ for $i \in \{1, 2\}$ and $j \in \{0, \ldots C_i\}$.

Figure 3.12 illustrates the intuitive meaning of the iterates. As long as a_1 and a_2 hold, it is possible to proceed to the next lower iterate of Y_i. Y_2 is reachable from $Y_{1,1}$ and Y_1 is reachable from $Y_{2,1}$. The resulting cycle allows to visit b_1 and b_2 infinitely often during the play. If a_2 is not satisfied, the next lower iterate of Y_2 may not be reachable. Never reaching

b_2 again is fine if a_2 is also never satisfied again. However, the other Streett pair still has to be handled. This is ensured through the iterates from the recursive step. Figure 3.12 shows them for $Y_{2,2}$ only. If a_1 holds, it is possible to proceed to the next lower iterate of $Y_{2,2}$ and from $Y_{2,2,1}$ back to $Y_{2,2}$. This cycle ensures that b_1 is visited infinitely often if a_1 holds infinitely often but a_2 does not. Analogously for all other iterates $Y_{i,j}$.

To define a strategy, we introduce one bit m of memory. $m = 0$ means b_1 should be fulfilled next, $m = 1$ means b_2 should be fulfilled next. The strategy is composed of several parts, which we enumerate in the following Table. They are prioritized from top to bottom. If a particular sub-strategy cannot be applied (because of violated assumptions), the next one is tried.

Table 3.1: Winning Strategy for two-Pair Streett Game.

Nr.	present state in:	next state in:	informal description
1	$Y_{1,i} \setminus Y_{1,i-1}, \neg m$	$Y_{1,i-1}, \neg m$	step towards b_1
2	$Y_{2,i} \setminus Y_{2,i-1}, m$	$Y_{2,i-1}, m$	step towards b_2
3	$Y_{1,1}, \neg m$	Z, m	b_1 reached; switch towards b_2
4	$Y_{2,1}, m$	$Z, \neg m$	b_2 reached; switch towards b_1
5	$Y_{1,i,j} \setminus Y_{1,i,j-1}, \neg m$	$Y_{1,i,j-1}, \neg m$	$\neg a_1$; sub-game towards b_2
6	$Y_{2,i,j} \setminus Y_{2,i,j-1}, m$	$Y_{2,i,j-1}, m$	$\neg a_2$; sub-game towards b_1
7	$Y_{1,i,1}, \neg m$	$Y_{1,i}, \neg m$	b_2 reached in sub-game
8	$Y_{2,i,1}, m$	$Y_{2,i}, m$	b_1 reached in sub-game
9	$Y_{1,i,j} \setminus Y_{1,i,j-1}, \neg m$	$Y_{1,i,j}, \neg m$	$\neg a_1, \neg a_2$; stay
10	$Y_{2,i,j} \setminus Y_{2,i,j-1}, m$	$Y_{2,i,j}, m$	$\neg a_2, \neg a_1$; stay

3.6 Example of Robust Synthesis

To demonstrate how our algorithm works, we look again on the simple arbiter example specified in Section 3.4. The specification of the arbiter is defined by the formula $\varphi = A^i \wedge A^t_1 \wedge A^t_2 \wedge A^l \rightarrow G^i \wedge G^t_1 \wedge G^t_2 \wedge G^l$.

First, the specification is transformed into a **one-pair Streett game**. In this example there is no need for a **counting construction**, since there is only a single liveness assumption and guarantee. Figure 3.6 illustrates the encoding of the safety properties A^t_1, A^t_2, G^t_1 and G^t_2 in the transition relation of the Streett game. For example, the transition relation requires that if there is a request r has to stay true until the request is granted.

The next step is to transform the *one-pair Streett game to a robust two-pair Streett game*. In order to do that, we **extend the state space** with the variables env_{err} and sys_{err}. Before, the number of states was $4 = 2^2$. Now, through the two additional variables, we end up with $16 = 2^4$ states.

Further, we have to **adapt the transition relation**. Figure 3.13 shows a cutout of the new transition relations of the two-pair Streett game. The first bit of each state corresponds

to the request signal r and the second bit to the grant signal g. The third bit of each state corresponds to the signal env_{err}, which encodes an error caused by the environment. If this bit is false, no safety assumption violation occurred in the current step. The last bit is sys_{err}, which keeps track of the safety guarantee violations.

First, let's consider the blue dashed transition in Figure 3.13. Let ρ_e and ρ_s be the original transition relations of the one-pair Streett game. Let's suppose the play is in the initial state $s = (\neg r, \neg g)$, and all signals are low. The environment chooses the next input $x' = r$. Since $(s, r) \models \rho_e$, there is no safety assumption violation. Now, the system chooses the next output $y' = g$, although it is not allowed to raise the grant signal immediately, according to the safety guarantees. We get $(s, r, g) \not\models \rho_s$. So, in the two-pair Streett game, we get for $s = (\neg r, \neg g, \neg env_{err}, \neg sys_{err})$, $x' = r$ and $y' = g$ the next state $s' = (r, g, \neg env_{err}, sys_{err})$.

Now let's consider the dotted orange transitions in Figure 3.13. Suppose, the current state is $s = (r, g, \neg env_{err}, sys_{err})$. According to the safety assumptions, if both request and grand are true, than the environment is allowed to maintain the request, or to lower it. So in any case, $env'_{err} = 0$. However, the system can make safety guarantee violation. $sys'_{err} = 1$, if the system lowers the grant signal in the next step. In this case, the system error bit stays high. Otherwise, if the system keep the grand signal high, the system has recovered and works correctly again, and $sys'_{err} = 0$.

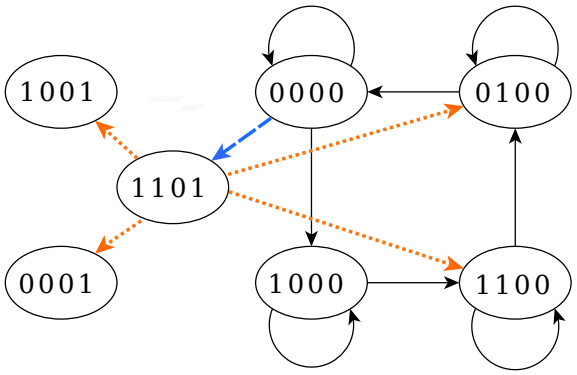


Figure 3.13: Cutout of the Transition Relation of the two-Pair Streett Game.

The complete transition relation graph consists of 16 states and 64 transitions. For a clear illustration, a more compact representation of the whole game graph has been chosen, see Figure 3.14. The first bit of each state represents r, the second bit g and the last bit env_{err}. The variable sys_{err} is encoded via the transitions. Black solid lines indicate that there is no system error ($sys_{err} = 0$) and red dashed-lines indicate that there is one ($sys_{err} = 1$). Colored states represent states where an environment error has occurred. E.g., assume we start in state $s = (100)$. In this state, a request has occurred which has not been granted yet, and no environment error occurred. The safety assumption prohibits the environment from lowering the request. If it does anyway, depending on the choice of the system, either the state $s' = (011)$ or $s' = (001)$ is entered, which are both colored states.

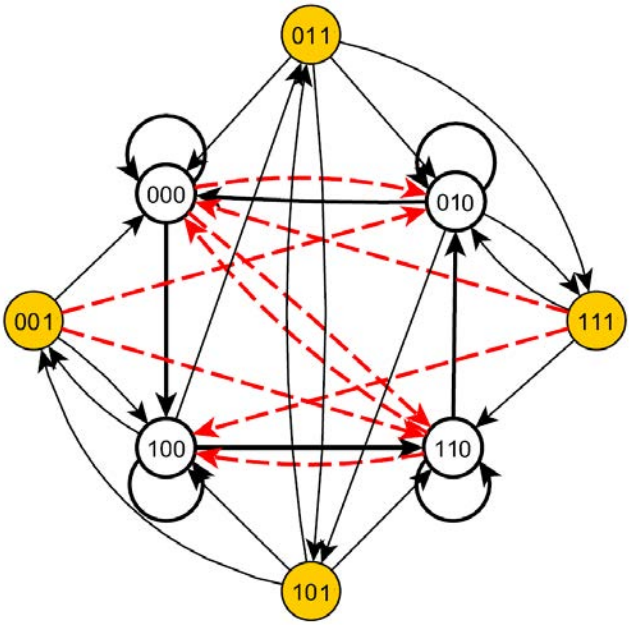


Figure 3.14: Transition Relation of the two-Pair Streett Game.

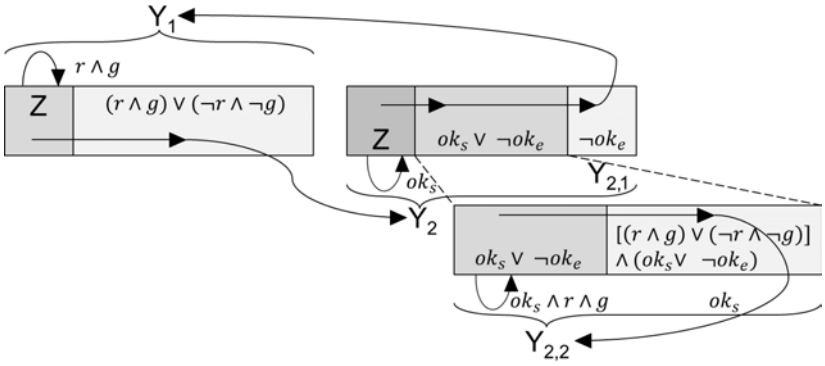


Figure 3.15: Arbiter Example: Illustration of the Iterates.

Next, the **winning region and the strategy** are computed. Figure 3.15 illustrates the iterates of the fixpoint computation. We have $a_1 = \neg(r \wedge g), b_1 = (r \wedge g) \vee (\neg r \wedge \neg g), a_2 = \neg ok_s, b_2 = \neg ok_e$. To illustrate strategy computation, we consider the following scenario. Assume that $m = 1$ and the arbiter is in a state out of $Y_{2,2} \backslash Y_{2,1}$. The value of $m = 1$ dictates that a state out of $Y_{2,1}$ is to be visited next, if possible. $Y_{2,1}$ contains all states with an environment error. If we assume that the environment always behaves correctly, the set $Y_{2,1}$ becomes unreachable. In order to win the game, the system is not allowed to make a mistake either, so the arbiter stays in $Y_{2,2}$. This way the second Streett pair $\langle(\neg ok_s), (\neg ok_e)\rangle$ is fulfilled, because both sets are only visited finitely often. To win the game, the first Streett

pair also has to be fulfilled. Therefore the subgame is entered, trying to reach states in b_1 while staying in $Y_{2,2}$. Through the loop in $Y_{2,2}$, it is possible to visit these states infinitely often, fulfilling the first Streett pair as well.

3.7 Recovery Time

Lemma 3.4. *(Recovery of a reactive System.)*
A robust system is able to recover from an environment failure with at most one system failure.

To prove Lemma 3.4, let's consider the Game Structure G of the one-pair Streett Game with transition relations ρ_e and ρ_s and the Game Structure $G^{\sim}$ of the robust two-pair Streett Game with transition relations $\rho_e^{\sim}$ and $\rho_s^{\sim}$. The transition relation ρ_e is complete. For all states, there exists at least one possible next input. Also the transition relation ρ_s is complete. For all states and for all possible next input values, there exists at least one possible next output value.

Suppose, we chose the following implementation of $\rho_s^{\sim}$:
$\rho_1 = \rho_e \wedge \rho_s \wedge \neg env'_{err} \wedge \neg sys'_{err}$
$\rho_2 = \rho_e \wedge \neg\rho_s \wedge \neg env'_{err} \wedge sys'_{err}$
$\rho_3 = \neg\rho_e \wedge \rho_s \wedge env'_{err} \wedge \neg sys'_{err}$
$\rho_4 = \neg\rho_e \wedge \neg\rho_s \wedge env'_{err} \wedge sys'_{err}$
$\rho_s^{\sim} = \rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4$

In the two-pair Streett game, the environment is allowed to choose any possible input combination, hence its transition relation is $\rho_e^{\sim} = \mathsf{true}$. If the environment violates ρ_e by doing so, the bit env_{err} is set to true. The system is allowed to choose its output values arbitrarily, only the values of env_{err} and sys_{err} result from the other choices of next input and output values.

Let's suppose. the environment violates ρ_e. The system can now choose a next output, such that such that the game is again in a "normal "state, where both environment and system can satisfy all safety properties in the next step. Such a state always exists, since the transition relations ρ_e and ρ_s are complete.

In some situations, when ρ_e is violated, the system may not be able to choose a next output without violating the transition function ρ_s. In most cases, the system has to violate ρ_s to reach a proper state. The only difference between violating and not violating ρ_s is that either the error bits are set or not. Since the system is also allowed to choose the output arbitrarily, it is always possible for the system to reach a valid state with at most one system error in the case of an environment error.

Suppose, we chose the following implementation for the system transition relation $\rho_s^{\sim}$:
$\rho_1 = \rho_e \wedge \rho_s \wedge \neg env'_{err} \wedge \neg sys'_{err}$
$\rho_2 = \rho_e \wedge \neg\rho_s \wedge \neg env'_{err} \wedge sys'_{err}$
$\rho_3 = \neg\rho_e \wedge \rho_s \wedge env'_{err} \wedge \neg sys'_{err}$
$\rho_4 = \neg\rho_e \wedge \neg\rho_s \wedge \neg env'_{err} \wedge \neg sys'_{err}$
$\rho_s^{\sim} = \rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4$

In this case, *the system has a winning strategy, where the system error bit will always stay low.* If there is an environment failure, the system may be able to recover without a system failure: $sys'_{err} = 0$. If there is an environment failure, and the system violates ρ_s, it also leads to $sys'_{err} = 0$ according to $\rho_s^{\sim}$.

There are at least two possible winning strategies. The first one is: in the case of an environment failure, do anything to get to a normal state. In this case the strategy could be that the system violates a guarantee even if it doesn't have to. We implemented an improved strategy: if the system is able to recover without a safety guarantee violation, it should do so.

4 Implementation of Robustness and Results

4.1 Implementation of Robustness in RATSY

To implement robustness into RATSY, most changes had to be made in Marduk. Only small cosmetic changes are made to RATSY itself, such as adding checkboxes to the user-interface. This section explains the changes.

4.1.1 RATSY

For the synthesis feature, RATSY's main task is to provide a comfortable user-interface. RATSY takes the user-settings and calls Marduk with the configured options, and Marduk creates the implementation. RATSY uses a Model-View-Controller (MVC) design pattern. It allows a clean separation of data-model, user-interface and control-logic. The library pygtkmvc is used in RATSY to implement this pattern, which provides an easy-to-use high-level MVC-framework for Python. Another important implementation aspect of RATSY is that it uses Glade to implement its graphical user interface. Glade is a rapid-development environment for creating gtk-windows, forms and panels via a comfortable and intuitive GUI-editor.

Changes to RATSY's forms. In order to allow robust synthesis, a new checkbox was added to the synthesis Glade-panel. This way, robust synthesis is still optional. For large specifications, it will sometimes not be possible to synthesize robust systems through to the higher complexity. Additionally, this checkbox was also added to the game panel. If the checkbox is enabled, realizability and a counterstrategy in case of non-realizability is computed for the robust setting. Most notable for the user is that input signals are now allowed to be chosen arbitrarily.

Changes to RATSY's models. The data-models for the modified panels were adapted, to keep track of the robustness checkbox.

Changes to RATSY's controllers. RATSY's controllers call Marduk and pass the user's configuration parameters to it. Therefore the controllers of the game and synthesis features were modified to pass the new robustness option correctly to Marduk. Originally, the control-logic did not allow input values during a game, which violates environment safety assumptions. This check was disabled, when the robust-synthesis option is active. In a counter game, the system is now allowed to violate safety assumptions, in order to win the game. During a normal game, the user can also choose arbitrary input vales, to see how the system recovers from safety assumptions violations. The system may respond by also violating safety guarantees, but will eventually recover.

4.1.2 Marduk

MARDUK is a straight forward implementation. The main module *(marduk.py)* is used for option parsing and management of the application's control-flow. After acquiring the options, MARDUK starts to calculate a winning-region and a winning-strategy and then creates the implementation of the circuit.

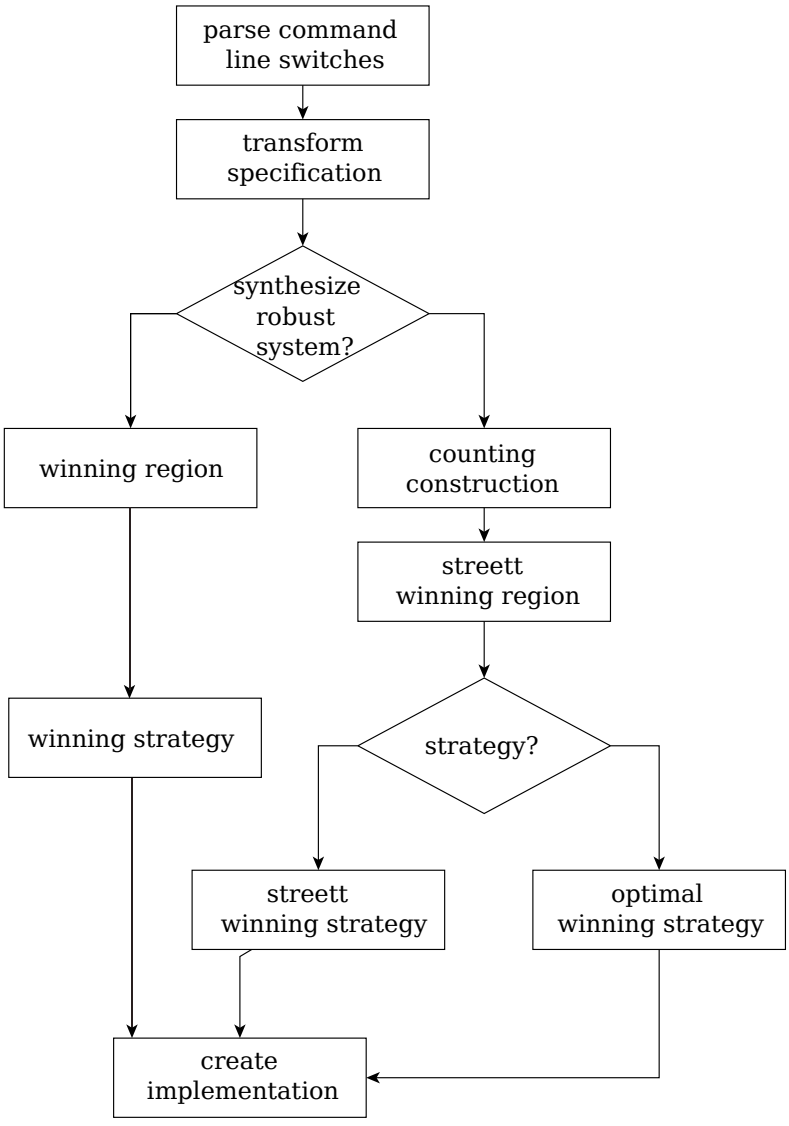


Figure 4.1: Control Flow of extended MARDUK.

The newly introduced robustness option is now used to alter the control flow of MARDUK, see Figure 4.1. If robustness is enabled, the counting construction is applied. The counting construction was implemented by Hans Jürgen Gamauf during his master thesis. Next the winning region and winning strategy for the two-pair Streett game are calculated. We

implemented two different winning strategies for robust synthesis: the first one is called *normal strategy* and the second one is called *best strategy*. The normal strategy works according to Table 3.5.5. In each step, the strategy tries to proceed to the next lower iterate, in order to fulfill it's guarantees. If playing according to the best strategy, the system chooses an iterate as low as possible, in order to fulfill the guarantees as soon as possible. MARDUK's simple design allowed this extension without any other modifications.

In order to compute a winning strategy for Streett games, we have to store the iterates of the fixpoint computation. Due to the recursive nature of the algorithm, storing these iterates requires large amounts of memory. Even simple examples, like arbiters with 10 request and acknowledgement lines, require an impracticable amount of memory.

4.2 Results

In this section, we tested our implementation in RATSY with an arbiter, with N request and acknowledge lines. The Tables 4.1 and 4.2 compare the results of the experiment. Three algorithms were compared. As baseline we used the already in RATSY implemented algorithm from Piterman et al.[PPS06]. The second algorithm extends the baseline algorithm by adding the counting construction. It shows the impact of the counting construction on the baseline algorithm. The third algorithm is our robust synthesis algorithm presented in this work. In the robustness algorithm, we used the *best strategy* as winning strategy.

Table 4.1 compares the synthesis time in seconds and Table 4.2 compares the implementation size in lines of Verilog code.

The counting construction results in a significant increase of the state space. This already leads to a huge increase in the synthesis time and in the size of the implementation compared with the original algorithm. As expected, the robust approach takes even more time and creates larger circuits than the second synthesis algorithm. This is due to the higher complexity of the new method.

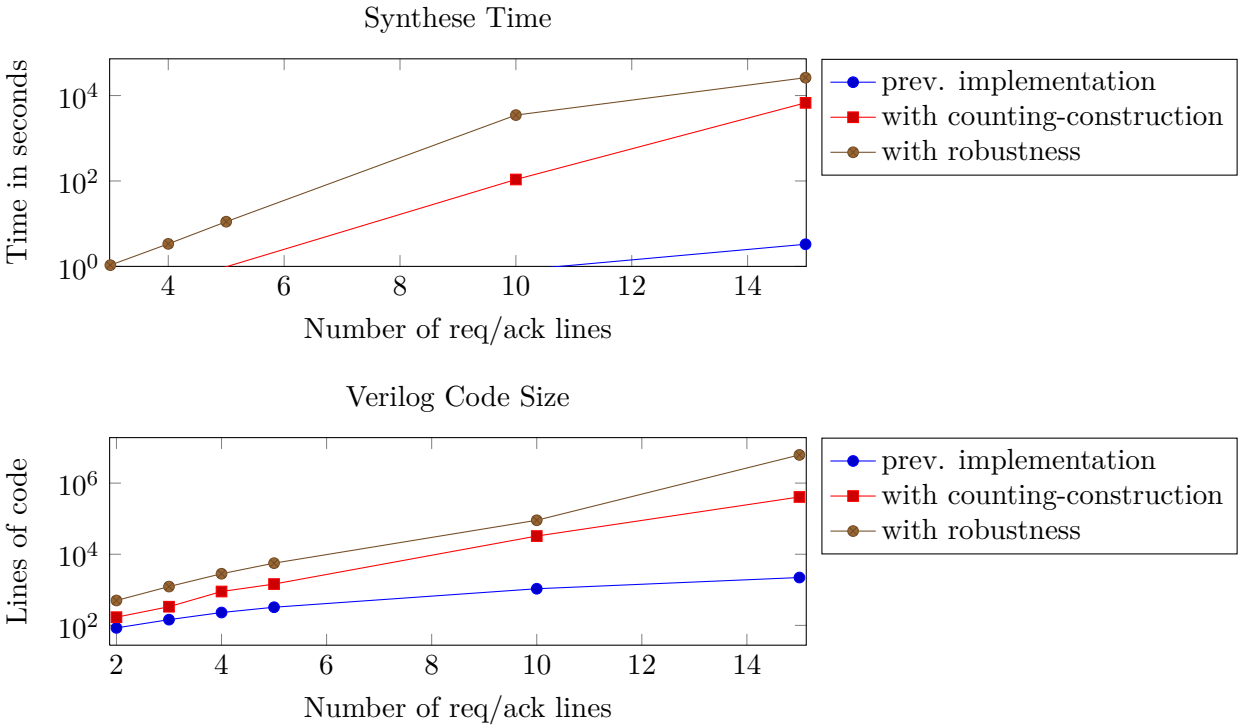
Table 4.1: Synthesis Time for Arbiter.

N	time prev. impl.	time with cc	time with robustness
2	0.04	0.06	0.15
3	0.08	0.16	1.07
4	0.14	0.4	3.37
5	0.18	0.98	11.13
10	0.81	108.37	3,485
15	3.30	6749	26,172

Figures 4.2 and 4.2 show a graphical representation of the data.

Table 4.2: Implementation Size for Arbiter.

N	size prev. impl.	size with cc	size with robustness
2	85	170	501
3	145	335	1,234
4	230	896	2,829
5	324	1,451	5,614
10	1,072	32,466	90,215
15	2,215	407,699	$6.2 \cdot 10^6$



We also compared the *normal winning strategy* and the *best winning strategy*. In most cases, systems synthesized according to the normal strategy end up in smaller circuites. The disadvantage is that they may react slower. For instance, the specification of an arbiter states that if there is a request, there will be a grant eventually. Usually, systems synthesized with the normal strategy need more time until the request is granted. The second strategy could lead to the reverse phenomenon. The circuits may be larger, therefore requests may be granted sooner.

5 Conclusions and Future Work

5.1 Conclusions

The original synthesis algorithm of RATSY gave no formal guarantees for robustness. The extension presented in this work guarantees that synthesized systems are *correct and robust by construction*. This comes at the cost of larger circuits and longer synthesis times, due to the increased computational complexity. The systems synthesized without the robustness method recover in zero or one steps. Experimental results show that in many practical cases, the ratio between system errors and environment errors is far below one. So the synthesized systems behave very reasonable in unexpected situations and for security relevant systems, it could be worth the overhead.

Since in practice, one has to be prepared for environment errors, guaranteed robustness is an important property that enhances the quality of a system.

5.2 Future Work

Possible future work would be to optimize the implementation to achieve better execution times. The implementation is a proof of concept implementation and has optimization potential. To compute the winning strategy, we use the iterates of the fixpoint algorithm of computing the winning region. Due to the recursive nature of the algorithm, storing these iterates requires large amounts of memory. Further optimization would be trying to reduce the needed memory.

Another way to handle unexpected situations would be to go to some reset states, if there are environment failures. This solution would result in much less overhead than our robust synthesis method and could be enough for many system requirements. It would be interesting to implement this method and to compare the resulting systems with the system of our robust synthesis method.

An interesting future work would be to expand our definition of robustness. In this work, we define a system to be robust if a finite number of environment failures only induce a finite number of system failures. This can be expressed via the formula $\mathsf{F}\,\mathsf{G}\,\neg env_{err} \rightarrow \mathsf{F}\,\mathsf{G}\,\neg sys_{err}$. An expansion would be to require additionally that if there is no environment failure, there should be no system failure. This can be expressed by the safety property

$$\varphi = \mathsf{G}\,\neg env_{err} \rightarrow \mathsf{G}\,\neg sys_{err}. \tag{5.1}$$

We could also require that either there is never a system error, or there is no system error until there is an environment error. This can be formulated by

$$\varphi = \neg sys_{err} \,\mathsf{W}\, env_{err}. \tag{5.2}$$

At the moment we assume that the environment does not violate the safety assumption A^i of the GR(1) specification. We can deal with violations of A^i by changing the initial state formula Θ of the game structure of the two-pair Streett game.

The formula Θ is a formula over the assumption A^i, the guarantee G^i, the counters i and j of the counting construction and the error bits env_{err} and sys_{err}. The formula Θ of the two-pair Streett game is currently defined by

$$\Theta = A^i \wedge G^i \wedge i = 0 \wedge j = 0 \wedge \neg env_{err} \wedge \neg sys_{err}. \tag{5.3}$$

In order to allow violations of A^i, we have to modify the formula Θ as follows:

$$\begin{aligned}
\Theta_1^{\sim} &= A^i \wedge G^i \wedge \neg env_{err} \wedge \neg sys_{err} \\
\Theta_2^{\sim} &= \neg A^i \wedge G^i \wedge env_{err} \wedge \neg sys_{err} \\
\Theta_3^{\sim} &= A^i \wedge \neg G^i \wedge \neg env_{err} \wedge sys_{err} \\
\Theta_4^{\sim} &= \neg A^i \wedge \neg G^i \wedge env_{err} \wedge sys_{err} \\
\Theta^{\sim} &= i = 0 \wedge j = 0 \wedge (\Theta_1^{\sim} \vee \Theta_2^{\sim} \vee \Theta_3^{\sim} \vee \Theta_4^{\sim})
\end{aligned}$$

According to our current definition of robustness, the system is allowed to violate the guarantee G^i regardless of the initial action of the environment. The additional requirements of Equation 5.1 and 5.2 would prevent this from happening.

List of Symbols

AP	Atomic propositions page 10
Σ	Alphabet page 13
$\Sigma^\star$	Set of finite words page 13
Σ^ω	Set of infinite words page 13
ϵ	Empty word page 13
$\bar{\sigma}$	Word over an Alphabet Σ page 13
A	Automaton page 13
π	Run on automaton A page 13
F	Temporal operator: Eventually page 23
X	Temporal operator: Next page 23
U	Temporal operator: Until page 23
W	Temporal operator: Weak Until page 23
E	Path operator: Exists page 28
A	Path operator: Forall page 28
μ	Least fixpoint operator page 29
ν	Greatest fixpoint operator page 29
G	Game Structure page 31
X	Set of input variables page 31
Y	Set of output variables page 31
D_X	Input domain page 31
D_Y	Output domain page 31
ρ_e	Transition function of environment in G page 31
ρ_s	Transition function of system in G page 31
$\Box$	Nexttime operator for the system page35
$\Diamond$	Nexttime operator for the environment page 35

Bibliography

[AMP94] Eugene Asarin, Oded Maler, and Amir Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems*, pages 1–20, 1994.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[AS10] Farhad Arbab and Marjan Sirjani, editors. *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers*, volume 5961 of *Lecture Notes in Computer Science*. Springer, 2010.

[AT04] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for ltl fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

[Aue06] Benalycherif L. Fedeli A. Fisman D. McIsaac A. Winkelmann K. Auerbach, G. Case studies in property-based requirements specification. *Prosyd Delivarable D1.4/1*, 2006.

[BBW06] Patrick Blackburn, Johan F. A. K. van Benthem, and Frank Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., New York, NY, USA, 2006.

[BCG+10a] Roderick Paul Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas Henzinger, and Barbara Jobstmann. Robustness in the presence of liveness. In Springer, editor, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 410 – 424. Springer, 2010.

[BCG+10b] Roderick Paul Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy - a new requirements analysis tool with synthesis. In Springer, editor, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 425 – 429, 2010.

[BCP+07] Roderick Bloem, Roberto Cavada, Ingo Pill, Marco Roveri, and Andrei Tchaltsev. Rat: A tool for the formal analysis of requirements. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.

[Bea96] David M Beazley. *SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C*, pages 129–139. USENIX Association, 1996.

[BGHJ09] Roderick Paul Bloem, Karin Greimel, Thomas Henzinger, and Barbara Jobstmann. Synthesizing robust systems. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009*, pages 85 – 92, 2009.

[BGJ+07a] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *DATE*, pages 1188–1193, 2007.

[BGJ+07b] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. volume 190, pages 3–16, 2007.

[BL69] J. Richard Buchi and Lawrence H. Landweber. Solving Sequential Conditions by Finite-State Strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[Bry95] Randal E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, ICCAD '95, pages 236–243, Washington, DC, USA, 1995. IEEE Computer Society.

[Büc62] Julius R. Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS'60)*, pages 1–11. Stanford University Press, June 1962.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. pages 359–364. Springer, 2002.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[Chu62] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[CRKB11] Chih-Hong Cheng, Harald Rueß, Alois Knoll, and Christian Buckl. Synthesis of fault-tolerant embedded systems using games: from theory to practice. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 118–133, Berlin, Heidelberg, 2011. Springer-Verlag.

[Dav90] A. Davis. *Software Requirements Analysis and Specification.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1990.

[DG08] Volker Diekert and Paul Gastin. First-order definable languages. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata: History and Perspectives*, volume 2 of *Texts in Logic and Games*, pages 261–306. Amsterdam University Press, 2008.

[DGV99] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In *CAV*, pages 249–260, 1999.

[DHLN10] Laurent Doyen, Thomas A. Henzinger, Axel Legay, and Dejan Nickovic. Robustness of sequential circuits. In *ACSD*, pages 77–84, 2010.

[Dow97] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, March 1997.

[ea96] Robert K. Brayton et al. Vis: A system for verification and synthesis. In *CAV*, pages 428–432, 1996.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[EF06] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time. In *POPL*, pages 127–140, 1983.

[EH85] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30(1):1–24, 1985.

[Ehl10] Rüdiger Ehlers. Symbolic bounded synthesis. In *CAV*, pages 365–379, 2010.

[Ehl11a] Rüdiger Ehlers. Experimental aspects of synthesis. In *iWIGP*, pages 1–16, 2011.

[Ehl11b] Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In *TACAS*, pages 272–275, 2011.

[EJS93] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model-checking for fragments of μ-calculus. In *CAV*, pages 385–396, 1993.

[EKA08] Ali Ebnenasir, Sandeep S. Kulkarni, and Anish Arora. Ftsyn: a framework for automatic synthesis of fault-tolerance. *Int. J. Softw. Tools Technol. Transf.*, 10(5):455–471, September 2008.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[Eme96] E. Allen Emerson. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models*, pages 185–214, 1996.

[ES84] E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, 1984.

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for ltl realizability. In *CAV*, pages 263–277, 2009.

[FJR10] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Compositional algorithms for ltl synthesis. In *ATVA*, pages 112–127, 2010.

[FJR11] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for ltl synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011.

[G̈99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, March 1999.

[Gar05] Simson Garfinkel. History's worst software bugs. In *Byte Magazine*, November 2005.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall, 1991.

[GKP11] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[GO01] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *CAV*, pages 53–65, 2001.

[GPV+95] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman and Hall, 1995.

[GR09] Alain Girault and Éric Rutten. Automating the addition of fault tolerance with discrete controller synthesis. *Form. Methods Syst. Des.*, 35(2):190–225, October 2009.

[HJK10] Jad Hamza, Barbara Jobstmann, and Viktor Kuncak. Synthesis for regular specifications over unbounded domains. In *FMCAD*, pages 101–109, 2010.

[Hol97] Gerard J. Holzmann. Designing bug-free protocols with spin. *Computer Communications*, 20(2):97–105, 1997.

[JB06] Barbara Jobstmann and Roderick Bloem. Optimizations for ltl synthesis. In *FMCAD*, pages 117–124, 2006.

[JGWB07] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In *CAV*, pages 258–262, 2007.

[KB05] Joachim Klein and Christel Baier. Experiments with deterministic omega-automata for formulas of linear temporal logic. In *CIAA*, pages 199–212, 2005.

[KE05] Sandeep S. Kulkarni and Ali Ebnenasir. Complexity issues in automated synthesis of failsafe fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2:201–215, 2005.

[Kel76] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, July 1976.

[KHB09] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD*, pages 152–159, 2009.

[KL08] Stephan Kreutzer and Martin Lange. Non-regular fixed-point logics and games. In *Logic and Automata*, pages 423–456, 2008.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[Kro99] Thomas Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[KV05] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542, 2005.

[Lam80] Leslie Lamport. "sometime" is sometimes "not never" - on the temporal logic of programs. In *POPL*, pages 174–185, 1980.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. 38(4):985–999, July 1959.

[LNP+12] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440, 2012.

[McM93] Kenneth L. McMillan. *Symbolic model checking.* Kluwer, 1993.

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

[Mor10] Andreas Morgenstern. *Symbolic controller synthesis for LTL specifications.* PhD thesis, 2010.

[Mos84] Andrzej Wlodzimierz Mostowski. Regular expressions for infinite trees and a standard form of automata. In *Symposium on Computation Theory*, pages 157–168, 1984.

[MP79] Zohar Manna and Amir Pnueli. The modal logic of programs. In *ICALP*, pages 385–409, 1979.

[MP81] Zohar Manna and Amir Pnueli. Verification of concurrent programs: Temporal proof principles. In *Logic of Programs*, pages 200–252, 1981.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[MS08] Andreas Morgenstern and Klaus Schneider. From ltl to symbolically represented deterministic automata. In *VMCAI*, pages 279–293, 2008.

[MSL08] Andreas Morgenstern, Klaus Schneider, and Sven Lamberti. Generating deterministic ω-automata for most ltl formulas by the breakpoint construction. In *MBMV*, pages 119–128, 2008.

[Mul63] David E. Muller. Infinite sequences and finite machines. In *SWCT (FOCS)*, pages 3–16, 1963.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.

[Nus97] Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14:15–16, 1997.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[Pnu86] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, pages 510–584. 1986.

[PP06] Nir Piterman and Amir Pnueli. Faster solutions of rabin and streett games. In *LICS*, pages 275–284, 2006.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.

[Pra95] Vaughan R. Pratt. Anatomy of the pentium bug. In *TAPSOFT*, pages 97–107, 1995.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

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

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

[Ros92] Roni Rosner. Modular Synthesis of Reactive Systems. PhD thesis. 1992.

[Saf88] Shmuel Safra. On the complexity of omega-automata. In *FOCS*, pages 319–327, 1988.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In *CAV*, pages 248–263, 2000.

[SC85] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[SF07] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *ATVA*, pages 474–488, 2007.

[SKK+02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, pages 389–398, 2002.

[SL09] Armando Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, pages 4–13, 2009.

[Som98] F. Somenzi. Cudd: Cu decision diagram package release, 1998.

[SS09] Saqib Sohail and Fabio Somenzi. Safety first: A two-stage algorithm for ltl games. In *FMCAD*, pages 77–84, 2009.

[Str82] Robert S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1/2):121–141, 1982.

[Tho96] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *TACAS*, pages 1–22, 2001.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

[VYY10] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.

[WHT03] Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *CIAA*, pages 11–22, 2003.

[Wol83] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.