Master's Thesis

Debugging Formal Specifications with Simplified Counterstrategies

Robert Könighofer[1]

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
A-8010 Graz, Austria

Advisor: Prof. Roderick Bloem

Graz, October 2009

[1] E-mail: robert.koenighofer@student.tugraz.at

Masterarbeit

Fehlerlokalisierung in formalen Spezifikationen mit vereinfachten Gegenstrategien

Robert Könighofer[1]

Institut für Angewandte Informationsverarbeitung und
Kommunikationstechnologie (IAIK)
Technische Universität Graz
A-8010 Graz, Österreich

Gutachter: Prof. Roderick Bloem

Graz, im Oktober 2009

Diese Arbeit ist in englischer Sprache verfasst.

[1] E-Mail: robert.koenighofer@student.tugraz.at

Abstract

A formal specification is typically derived manually from an informal design intent. Due to mistakes in this process, the resulting specification may be incomplete, unrealizable, or in conflict with the design intent. This work proposes debugging techniques for the latter two cases in the context of temporal specifications for reactive systems.

In order to debug conflicts between the formal specification and the informal design intent, the user has to understand the conflicts first. We show how the explanation of such conflicts can be reduced to the explanation of the unrealizability of a specification. Unrealizability is also interesting in its own right. Again, the user has to understand the problem before she can fix it. Our method for explaining unrealizability is based on the presentation of a counterstrategy, which illustrates the problem: If the input is chosen according to the counterstrategy, no behavior of the system can fulfill the specification. Counterstrategies may be complex and hard to understand for the user, especially for large specifications. Hence, we propose several ways to simplify them. First, we remove those requirements and signals from the specification which are not part of the problem. Second, we heuristically search for a *countertrace*. A countertrace is a single input trace for which no system can fulfill the specification. It can be thought of as a counterstrategy that is independent of the behavior of the system. Finally, we present the countertrace or the counterstrategy to the user in form of an interactive game and as a graph that summarizes all plays in this game. Our debugging method operates solely on the specification, i.e., it does not require an actual implementation of the design.

This work shows how simplified counterstrategies can be computed and presented to the user in order to debug unrealizable specifications, and specifications which are in conflict with the design intent. At first, this is done for a rather general setting. Then, the approach is further elaborated for the class of GR(1) specifications. For this class of specifications, experimental results are finally presented.

Keywords: Formal Specifications, Reactive Systems, Debugging, Unrealizability, Counterstrategies

Kurzfassung

Eine formale Spezifikation wird typischerweise manuell von einer informellen Design-Absicht abgeleitet. Aufgrund von Fehlern in diesem Prozess kann die so entstandene Spezifikation unvollständig sein, sie kann nicht realisierbar sein oder sie kann im Widerspruch mit der Design-Absicht stehen. Diese Arbeit präsentiert Konzepte zur Fehlerlokalisierung für die beiden letztgenannten Fälle im Kontext von temporalen Spezifikationen für reaktive Systeme.

Bevor ein Widerspruch zwischen einer formalen Spezifikation und einer informellen Design-Absicht aufgelöst werden kann, muss dieser Widerspruch zunächst verstanden werden. Wir zeigen, dass die Erklärung solcher Widersprüche auf die Erklärung der Unrealisierbarkeit einer Spezifikation zurückgeführt werden kann. Die Unrealisierbarkeit einer Spezifikation ist auch für sich genommen ein interessantes Problem. Wie vorhin muss das Problem zunächst verstanden werden bevor es gelöst werden kann. Unsere Methode um die Unrealisierbarkeit einer Spezifikation zu erklären basiert auf der Präsentation einer Gegenstrategie, die das Problem illustriert: Wenn der Input gemäß dieser Gegenstrategie gewählt wird, kann kein Verhalten des Systems die Spezifikation erfüllen. Solche Gegenstrategien können mitunter recht komplex und dadurch schwierig zu verstehen sein, insbesondere für große Spezifikationen. Deshalb stellen wir einige Konzepte zu ihrer Vereinfachung vor. Zunächst entfernen wir jene Anforderungen und Signale von der Spezifikation, die nicht Teil des Problems sind. Weiters suchen wir heuristisch nach einer *Gegensequenz*. Eine Gegensequenz ist eine einzelne Sequenz von Inputs, für die kein System die Spezifikation erfüllen kann. Sie kann als eine Gegenstrategie verstanden werden, die unabhängig vom Verhalten des Systems ist. Schließlich präsentieren wir die Gegensequenz oder die Gegenstrategie dem Benutzer in Form eines interaktiven Spiels und als Graph, der alle möglichen Abläufe in diesem Spiel zusammenfasst. Unsere Methode zur Fehlerlokalisierung arbeitet einzig und alleine mit der Spezifikation, das heißt sie benötigt keine Implementierung des Designs.

Diese Arbeit zeigt wie vereinfachte Gegenstrategien berechnet und präsentiert werden können um damit Fehler in nicht realisierbaren Spezifikationen oder in Spezifikationen, die der Design-Absicht widersprechen, zu finden. Der Ansatz wird zunächst für recht allgemeine Rahmenbedingungen erläutert. Danach erfolgt eine tiefergehende Ausarbeitung für GR(1)-Spezifikationen. Für diese Klasse von Spezifikationen werden schließlich auch experimentelle Ergebnisse präsentiert.

Schlagworte: Formale Spezifikationen, Reaktive Systeme, Unrealisierbarkeit, Gegenstrategien, Fehlerlokalisierung

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 erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

. . . . . .
Ort, Datum

. . . . . .
(Unterschrift)

Acknowledgements

I am indebted to numerous people who supported me in the creation of this thesis. First of all, I want to express my gratitude to my advisor Roderick Bloem. He sparked my interest in the field of formal methods for design and verification. Roderick always took time to discuss problems and half-baked ideas with a lot of patience. Without him giving theoretical background knowledge, pinpointing inconsistencies, and giving impulses to solutions, I would have been hopelessly lost in the topic.

Secondly, I would like to thank Georg Hofferek and Karin Greimel for their support and feedback concerning the implementation of this work. I also want to thank for the fruitful exchange of ideas in various project meetings and paper discussions. I furthermore thank Amir Pnueli for suggesting the use of counterstrategies and Viktor Schuppan for his valuable comments.

Special thanks is addressed to my girlfriend Elisabeth Jöbstl for proofreading this work, for motivating me, and for keeping me grounded. Last but not least, I thank my parents Rosa and Heinz Könighofer for giving me the opportunity to study as well as for their moral support and financial backing.

Robert Könighofer
Graz, Austria, October 2009

Danksagung

Vielen Menschen, die mich in der Erstellung dieser Diplomarbeit unterstützt haben, bin ich zu großem Dank verpflichtet. Zunächst will ich meinem Betreuer Roderick Bloem meine Dankbarkeit zum Ausdruck bringen. Er hat mein Interesse an formalen Methoden für Design und Verifikation geweckt. Roderick hat sich immer Zeit genommen um Probleme und unausgegorene Ideen zu diskutieren, und das mit viel Geduld. Ohne dass er theoretisches Hintergrundwissen gegeben hätte, Inkonsistenzen aufgezeigt hätte, und Impulse zur Lösung von Problemen gegeben hätte, hätte ich mich im Thema wohl hoffnungslos verloren.

Weiters möchte ich Georg Hofferek und Karin Greimel für ihre Unterstützung und ihr Feedback bezüglich der Umsetzung dieser Arbeit danken. Ich bedanke mich auch für den fruchtbaren Gedankenaustausch in zahlreichen Projekttreffen und Paperbesprechungen. Ich danke weiters Amir Pnueli für den Vorschlag Gegenstrategien zu verwenden und Viktor Schuppan für seine wertvollen Kommentare.

Ganz besonderer Dank gilt meiner Freundin Elisabeth Jöbstl für ihr Korrekturlesen, dafür dass sie mich motiviert, und für den Rückhalt, den sie mir gibt. Schließlich danke ich meinen Eltern Rosa und Heinz Könighofer für die Möglichkeit zu studieren sowie für ihre moralische und finanzielle Unterstützung.

Robert Könighofer
Graz, Österreich, im Oktober 2009

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Background and Motivation

More than ever, our daily life is dependent on the correct functioning of the computer systems surrounding us. Incorrect systems often cause high costs and sometimes even endanger lives. One example of an incorrect hardware system that has been shipped to customers is the Pentium processor of 1994. The famous FDIV bug let the floating point unit compute false results under certain circumstances [83]. Intel had to spend about 475 million US dollars to replace the faulty processors [65]. In 1996, an even more spectacular instance of a bug caused an Ariane 5 rocket to leave its intended path, so it had to be destructed. The reason was an incorrectly handled software exception caused by a data conversion operation [78]. The direct costs were estimated to about 370 million US dollars [36]. Further examples of hardware and software bugs, even some causing loss of life, can easily be found [48].

Ensuring the correctness of the developed systems is also one of the biggest challenges in today's hardware and software engineering processes. Various formal and informal methods can be utilized. Informal methods such as testing or simulation usually cannot be carried out exhaustively [77]. Formal methods, on the other hand, are able to guarantee the correctness regarding some formal specification. In the past, the applicability of formal methods for design and verification in industrial practice was very limited. The methods did not scale, the notation was complicated, and tools rarely existed [30]. Recent advances, especially in the field of model checking and automated property-based synthesis, have changed the situation.

1.1.1 Model Checking

Model checking, which is a technique for formal verification, can be used to prove that a system (a model thereof, to be more precise) has a certain property. The full automation of the check, together with the fact that efficient algorithms for powerful logics are available, makes model checking attractive [76].

The key inspiration was provided by modal and temporal logics [40]. Pnueli [81] suggested to use temporal logics to reason about non-terminating concurrent programs, so-called *reactive systems*. Temporal logic can be classified into *linear* and *branching time* [38]. The Linear Temporal Logic (LTL) [81] is a popular instance of the former class, whereas the Computation Tree Logic (CTL) [26] is a widely used representative of the latter. Both logics, LTL and CTL, are of incomparable expressive power [25].

Pnueli and others first advocated hand constructed proofs [24], but the manual proof construction was soon replaced by a model-theoretic approach that could be automated. The first model checker, named Extended Model Checker (EMC), was already created in 1982 [27; 28]. It could solve the model checking problem for CTL properties in polynomial time. McMillan [73] proposed the use of Binary Decision Diagrams (BDDs) [12] for the symbolic representations of the state transition graph, which made the verification of larger systems feasible [29]. For the more popular class of LTL specifications, the model checking problem could be proven to be PSPACE-complete [89]. Nevertheless, several new techniques and advances (e.g., as presented by Bloem et al. [9]) made LTL model checking competitive with CTL model checking regarding efficiency.

Nowadays, model checkers are already widely used in the industry, first of all in the development of systems with safety critical or economically vital applications [40]. Among the most popular [44] model checking tools are SPIN [54], SMV [73] with its variants NuSMV [20] and RuleBase [4], VIS [11], and COSPAN/FormalCheck [52].

1.1.2 Automated Synthesis

Another way to obtain a correct system is to use automated synthesis techniques. Given a formal specification of a system, an implementation thereof is constructed automatically. The resulting system is guaranteed to conform to the specification (*correct-by-construction*).

The applications and benefits that automated synthesis procedures can provide are numerous. The system engineer only has to specify *what* the system should do, but not *how*. Hence, the system engineer can operate on a higher abstraction level. The circumstance that the construction of a system that implements the specified behavior is automated excludes a potential source of defects and reduces the amount of manual work and hence the costs for the development of a system. Another application of automated synthesis is in rapid prototyping: Automatically synthesized implementations of some modules of a complex system can be used to test or simulate the entire system already before more efficient (manual) implementations of all system modules are available. Last but not least, automated synthesis can help in the task of understanding and debugging specifications. It makes simulations of the specification possible, since simulating the outcome of the synthesis process can be thought of as simulating the specification itself. In this thesis, we will make use of this idea.

Research on Automated Synthesis

The problem of finding an implementation to a given specification was already mentioned in 1962 by Church [19]. It is therefore often referred to as *Church's problem*. Church stated the synthesis problem in the context of the monadic second-order logic S1S (confer to the work of Thomas [94]). A few years later, a theoretical solution has been given by Büchi and Landweber [14] as well as by Rabin [85]. The given solutions are quite different: Büchi and Landweber use infinite games, while Rabin solves the problem over tree automata.

For the more popular class of LTL specifications, the synthesis problem was solved in the context of reactive systems by Pnueli and Rosner [82]. The idea was to transform the LTL specification into a Büchi automaton [13] which accepts exactly those words that fulfill the specification. Vardi et al. [98] as well as Emerson et al. [42] show how this can be done. In general, the resulting Büchi automaton is non-deterministic. Also, its size is exponentially larger than the size of the LTL specification. Next, the non-deterministic Büchi automaton is determinized by Safra's construction [88], yielding a deterministic Rabin automaton [84]. This operation causes another exponential blow-up of the state space. The resulting Rabin automaton is finally interpreted as a tree automaton. This tree automaton represents an implementation of the specification. If its language is empty, the specification is unrealizable, i.e., no system can implement the specification.

The high complexity of the synthesis procedure as defined by Pnueli and Rosner is no coincidence. It can even be proven that the complexity of synthesis from LTL specifications has a double exponential lower bound [86]. Another problem of this approach is that Safra's construction is very hard to implement [51]. Hence, there has been research focused on the avoidance of Safra's construction in the synthesis process. Kupferman and Vardi [68] propose a method to avoid Safra's construction by a transformation into universal co-Büchi tree automata and non-deterministic Büchi tree automata. This procedure has the same complexity as the approach over Safra's construction (which is asymptotically optimal), but is much simpler to implement. It furthermore provides various possibilities for optimizations and can be implemented symbolically (e.g., by using BDDs). In a subsequent publication [67], a compositional version of this construction has been presented. Jobstmann and Bloem [58] finally present several optimizations of the Safraless approach. They also implemented their ideas in the tool Lily, which is able to synthesize systems from full LTL specifications. Of course, the tool is limited to rather small specifications because of the difficulty of the problem.

The high complexity of LTL synthesis caused the research in this field to focus on subsets of LTL and on other specification languages for which more efficient synthesis procedures exist. Alur and La Torre [3] define different fragments of LTL for which synthesis is feasible with less than a double ex-

ponential complexity in the size of the specification. Wallmeier et al. [99] consider specifications that consist of safety conditions and so-called *request-response* conditions. Request-response conditions are given as tuples (p, q) with the meaning that after the condition p has been fulfilled, the condition q must be fulfilled. A synthesis algorithm, which has also been implemented, is presented. Other recent work includes synthesis from Live Sequence Charts (LSCs) [10; 66] and synthesis from Metric Temporal Logic (MTL) [72]. Live Sequence Charts [32] represent a scenario based specification notation, whereas MTL [63] is a real-time temporal logic.

An important contribution to automated synthesis has also been made by Piterman et al. [80]. They present an algorithm (see also Section 2.6) to synthesize systems from specifications belonging to the class of Generalized Reactivity of Rank 1 (GR(1) for short), a subset of LTL. Such a specification consists of two parts, a set of environment assumptions and a set of system guarantees (see also Section 2.2). If the environment of the system fulfills all assumptions, the specification requires the system to fulfill all guarantees. It has been demonstrated in several case studies [7; 8] that the class of GR(1) specifications is expressive enough to be used for modeling real-world systems. However, there are properties which cannot be expressed in this language. For example, one cannot embody *compassion* requirements (also called *strong fairness* requirements) within GR(1) [80]. Compassion requirements are given as a set of tuples $\{(p_1, q_1) \dots (p_n, q_n)\}$ such that for all $1 \leq i \leq n$ the condition q_i has to be fulfilled infinitely often if p_i is fulfilled infinitely often. The algorithm of Piterman et al. has been implemented in the tool Anzu [59] as well as inside RATSY, a successor of the requirement analysis tool RAT [79].

All in all, we conclude that automated synthesis is certainly a promising approach for the construction of correct systems in the future, although not yet mature enough to be widely accepted in industry.

1.1.3 Correct Formal Specifications

As already motivated, formal methods for design and verification are of great importance for the construction of correct systems. However, such methods can only guarantee the correctness of a system with respect to a formal specification thereof. If the formal specification itself is incorrect, then the constructed system is likely to be incorrect as well.

Formal specifications are also used in other engineering tasks. In software testing, for example, formal specifications can be used to (automatically) generate test cases. One way to do so is to utilize a model checker [46]. Formal specifications can furthermore serve as test oracles, i.e., to determine the correct outcome of a test case. For this purpose, the Java Modeling Language (JML) [69], an executable specification language, is already widely used in practice. Of course, the correctness of the utilized specifications is of outermost importance for such applications as well.

Typically, a formal specification is derived manually from some informal design intent. Ideally, the resulting formal specification is *sound* and *complete*. A specification is said to be sound [34] if all systems that correctly implement the informal design intent conform to the specification. It is said to be complete [34] if no system that is invalid with respect to the design intent conforms to the specification.

Yet, just like many other engineering tasks, creating a formal specification is an error-prone process. As a consequence of a mistake, the resulting formal specification might not express what it was intended to express, i.e., it might be incomplete or not sound. The detection of flaws in the specification, as well as the correction of the specification such that the design intent is expressed, are both challenging tasks.

1.2 Problems Addressed in this Thesis

Incomplete specifications and coverage metrics to detect incompleteness have already been addressed before in various contexts [17; 18; 23; 34; 45; 56; 60] (see Section 7 for a detailed discussion). This work is therefore focused on debugging techniques for formal specifications that are not sound. Incompleteness is only addressed peripherally. A shortened version of this work will soon be published [61].

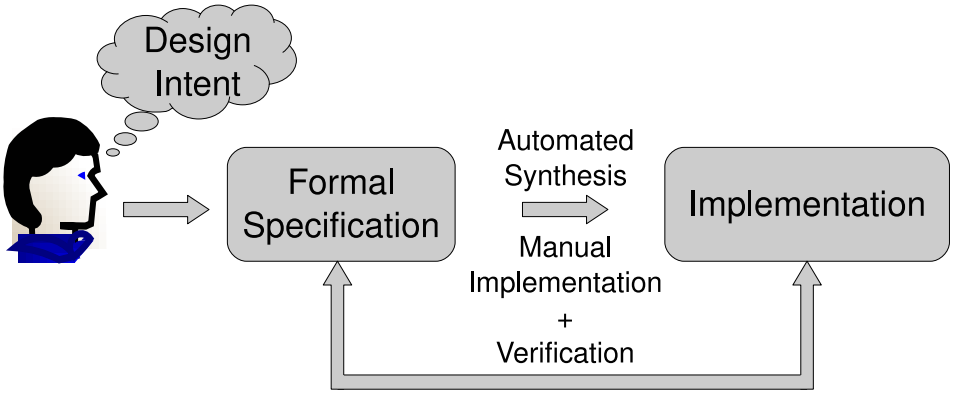


Figure 1.1: The assumed design flow for the construction of a correct system: A formal specification is derived manually from some informal design intent. This specification is then implemented either manually or automatically. Verification is used to ensure the correctness in case of a manual implementation.

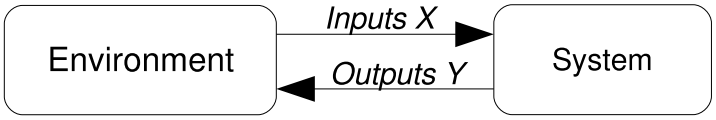


Figure 1.2: The setting of a reactive system: A system interacts with its environment via inputs and outputs. The interaction is maintained indefinitely, i.e, the system does not terminate. A specification for a reactive system defines what is allowed in this interaction.

1.2.1 The Assumed Setting

We assume a design flow as depicted in Figure 1.1. A formal specification is derived manually from some informal design intent. The specification is then used to create an implementation, either manually or by means of automated synthesis. In case of automated synthesis, the produced system is guaranteed to be correct by construction. In case of a manual implementation, formal verification techniques such as model checking can be used to ensure the correctness of the system. The design flow illustrated in Figure 1.1 assumes that the formal specification is created before an implementation of the design is available. Hence, our goal is a debugging method which operates solely on the specification, i.e., which does not rely on the availability of a corresponding implementation.

We restrict our investigation to specifications for reactive systems [53]. Reactive systems are systems that do not terminate. As illustrated in Figure 1.2, they continuously interact with their environment over a set X of input signals and a set Y of output signals. In every time step, the environment first provides input values, after which the system responds with output values. Hence, the environment behaves as a Moore machine and the system as a Mealy machine (see also Section 2.3.2). A specification for a reactive system defines what is allowed in this interaction.

The specifications considered in this work are furthermore supposed to be of the form $A \rightarrow G$, where A is a (possibly empty) set of environment assumptions and G is some set of system guarantees. The arrow in this notation indicates an implication of the following form: if the environment behavior is conform with the assumptions, then the specification requires the system behavior to fulfill all guarantees.

1.2.2 The Problem of Unrealizability

A special case of a specification that is not sound is a specification that is *unrealizable*. Intuitively, a specification is unrealizable if no system (no Mealy machine as defined in Section 2.3.2) can implement it [1; 82]. More formally, the realizability of a specification can be defined over an infinite game of perfect information that is constructed from the specification [1]: The game has two players, the system controlling the outputs and the environment controlling the inputs. The two players move alternately, with the environment starting. The system wins if it produces a correct behavior according to the specification. A winning strategy for the system in this game is a strategy to find output values in such a way that the system always wins, independent of the behavior of the environment, i.e., independent of the inputs chosen by the environment (see also Section 2.4). A specification is realizable iff such a winning strategy for the system exists in the game obtained from the specification [1].

A specification that is unrealizable can never be sound, no matter what the actual design intent was (as long as this design intent itself is realizable). Hence, unrealizable specifications are never correct and never desired. They can only result from mistakes in the specification development process. Our experience with the synthesis tool Anzu [59] shows that the problem of ending up with an unrealizable specification is quite common when creating a formal specification. In addition, it is often very difficult to localize the bug in the specification.

Realizability versus Satisfiability

Another similar yet different term is the term of *satisfiability*. A specification is said to be satisfiable if there exists at least one trace of signal values that conforms to the specification. For specifications of *closed systems*, satisfiability is equivalent to realizability [82]. A closed system is a system without any inputs, i.e., a system that is not able to react to actions of the environment. In contrast, an *open system* has inputs, which cannot be controlled by the system.

We consider reactive systems, which are open systems that do not terminate but continuously interact with their environment. For specifications of such systems, there is a difference between satisfiability and realizability [82]. It is sufficient for a specification to be satisfiable if one trace of inputs and outputs conforms to the specification. Realizability, however, requires that for each input trace there must be an output trace such that their combination fulfills the specification. Additionally, the output in any time step may only depend on past and present inputs. If a system can only comply with the specification when it is able to look into the future, then this specification is unrealizable. Clearly, every unsatisfiable specification is also unrealizable, but not vice versa.

The difference between satisfiability and realizability in the case of a reactive system can be stated in a more formal manner with the help of trees [49]. Let $\mathcal{X}$ be the input alphabet and let $\mathcal{Y}$ be the output alphabet of a reactive system. An *(output labeled, input branching) tree* t is a tuple $t = (N, L, n_0)$, where N is a set of nodes, $L : N \to \mathcal{Y}$ is a node labeling function, and $n_0 \in N$ is the root node of the tree. The set of nodes N forms a prefix closed subset of $\mathcal{X}^*$. Each node represents a trace of inputs, the node n_0 represents the empty trace ϵ. Let $n \cdot x$ denote the concatenation of the input trace in node $n \in N$ with the input $x \in \mathcal{X}$. A *path* $\overline{\pi}$ in t is a maximal trace $n_0 n_1 \ldots$ of nodes such that there exists a trace of inputs $x_1 x_2 \ldots$ so that $n_{i+1} = n_i \cdot x_{i+1}$ for all $i \geq 0$. Each path $\overline{\pi}$, built with the input trace $x_1 x_2 \ldots$, is assigned a word $\mathcal{L}(\overline{\pi}) = (x_1, y_1)(x_2, y_2) \ldots$ such that $y_i = L(n_i)$ for all $i \geq 1$. A specification is now *realizable* iff there exists a tree t such that for all path $\overline{\pi}$ in t the corresponding word $\mathcal{L}(\overline{\pi})$ fulfills the specification. In contrast, a specification is *satisfiable* iff there exists a tree t in which for one path $\overline{\pi}$ the corresponding word $\mathcal{L}(\overline{\pi})$ fulfills the specification.

Intuitively, in this definition, the function L serves as a strategy for the system to find output values, given the trace of input values observed so far. For realizability, we require that one such function exists which produces valid output traces for any possible input trace. In order to find an output value y_i in step i, this function is given only the inputs $x_1 \ldots x_i$, so the function is not able to look into the future. For satisfiability, on the other hand, we only require that a function L exists which produces a valid output

trace for one particular input trace. For satisfiability, this function could even be allowed to look into the future, i.e., to use inputs x_{i+k}, with $k > 0$, in order to determine the output value y_i in step i. However, this function need not be given the entire input trace explicitly. It can simply assume a certain input trace, because it has to produce valid outputs only for this input trace, anyway.

Our case study (cf. Section 6) will show that many unrealizable specifications are indeed satisfiable. For specifications of the form $A \rightarrow G$, where A is a set of environment assumptions and G is a set of system guarantees, this is not surprising. The existence of one trace that violates the assumptions is enough to make the specification satisfiable, no matter what the guarantees are.

Explaining Unrealizability

Before the user can fix an unrealizable specification, she has to understand the problem. Yet, explaining unrealizability is not easy. In case of an erroneous piece of software or hardware, one would simply execute or simulate it in order to track down the error. For an unrealizable specification, this is not possible. As already mentioned, unrealizable specifications are often still satisfiable, so existing techniques from SAT solvers cannot be used either. Tools like RAT [79] can be used to explain why a single trace does not fulfill the specification. Again, this does not help much in the task of explaining unrealizability. After all, the user cannot try out every possible input trace in order to check if it prevents a system from conforming to the specification.

Providing the user with simple explanations for unrealizability so that she can resolve the problem in the specification is one of the main issues in this thesis.

1.2.3 Soundness Problems in General

Problems in the specification often show up when an implementation of the specification is simulated or tested. Hence, we consider soundness problems other than the problem of unrealizability in the following scenario: The user simulates a system conforming to the formal specification and observes undesired behavior, i.e., behavior that does not conform to the informal design intent. Since the simulated system conforms to the formal specification, there must be a mismatch between this formal specification and the informal design intent, that is, the formal specification is clearly incorrect.

In such a case, the specification can either be incomplete or not sound. The first problem is to find out which of the two cases applies. If the specification is not sound, the second challenge faced in this thesis is the explanation of the problem so that the user can fix it.

1.3 Outline of the Solution

In this thesis, we will first present a rather generic approach for debugging soundness issues in formal specifications. It is then further elaborated for specifications belonging to the class of *Generalized Reactivity of Rank 1* [80] (we will write GR(1) as abbreviation throughout the rest of this document). This class of specifications was chosen because it is expressive enough to be used for modeling real-world systems [7; 8] while still offering efficient algorithms [80]. In order to evaluate our debugging approach, we have implemented it for the class of GR(1) specifications inside RATSY[1], a successor of the requirement analysis tool RAT [79], as well as inside the synthesis tool Anzu[2] [59].

1.3.1 Explaining Unrealizability

We explain unrealizability by presenting a counterstrategy. A counterstrategy is a strategy to find "problematic" inputs. With these inputs, no behavior of the system can fulfill the specification. For a speci-

[1] Available at `http://rat.fbk.eu/ratsy` (last visit in October of 2009)

[2] Available at `http://www.ist.tugraz.at/staff/jobstmann/anzu/` (last visit in October of 2009)

fication of the form $A \rightarrow G$, the inputs dictated by the counterstrategy will conform to all environment assumptions $a \in A$ and at the same time force any system to violate at least one of its guarantees $g \in G$. Thus, a counterstrategy demonstrates that the environment can force any system to violate the specification, i.e., that no system can implement the specification.

The use of counterstrategies in this context is not new. Counterstrategies have already been mentioned as aids for diagnostics in various settings [5; 10; 71; 91; 92; 93; 96] (see Section 7 for a detailed discussion). However, judging from experience, we claim that the sole presentation of a counterstrategy does not suffice to help the user debug unrealizability in larger specifications.

Keeping the Explanations Simple

In general, the inputs suggested by the counterstrategy depend on all previous values of the output signals of the system. Thus, a counterstrategy can be presented as a graph or an interactive game (cf. Section 3.2). For larger specifications, this graph or game can become so complex that the user has no chance to learn where the specification is too restrictive to be realizable. Hence, we propose several techniques to keep the counterstrategies simple.

First, we compute an unrealizable core as suggested by Cimatti et al. [21]. An unrealizable core is a simplified specification that is still unrealizable. The idea is that the problem is easier to locate in this simplified version of the specification. We combine this idea with counterstrategies. Furthermore, we improve the work of Cimatti et al. in three points: (1) we do not minimize environment assumptions as this step is computationally expensive, (2) we also remove output signals from the specification in order to achieve an additional simplification, and (3) we use Delta Debugging as a more advanced and often faster minimization algorithm.

Second, we propose to use a *countertrace* instead of a counterstrategy in order to explain unrealizability. A countertrace is a single trace of inputs for which no behavior of the system can fulfill the specification. A countertrace can therefore be seen as a counterstrategy that is independent of the behavior of the system. It is thus way easier to understand than a conventional counterstrategy. The problem is that a countertrace does not always exist and that its computation is expensive. We therefore present a heuristic algorithm for the computation of countertraces. Countertraces have not been mentioned before in the literature to the best of our knowledge.

The above mentioned simplification techniques for counterstrategies, enabling the communication of meaningful information to the user even for larger specifications, have to be seen as the core contribution of this work.

1.3.2 Resolving the General Case of a Soundness Problem

Suppose, as motivated in Section 1.2.3, that undesired behavior has been observed during the simulation of a system implementing some formal specification. The fact that the implementation conforms to the formal specification implies that this formal specification is incorrect.

In order to be able to distinguish between the case where the specification is incomplete and the case where the specification is not sound, we let the user specify the desired response to the given input scenario. We suggest that the user simply modifies the incorrect simulation trace to make it represent the desired behavior.

If the specification is so restrictive that no system implementing this desired behavior can comply with the specification, then the specification is not sound. There is a conflict between the design intent and the specification, since the specification disallows the desired behavior. Before the user can resolve this conflict, she has to understand it. We reduce the explanation of such conflicts to the explanation of the unrealizability of a specification. Thus, our approach for explaining unrealizability can be used to debug other kinds of soundness problems as well. We are not aware of any previous work that shows how

counterstrategies can be used in order to debug conflicts between a formal specification and the design intent.

If undesired behavior is observed during simulation, and if the formal specification of the system would allow the desired behavior as well, then the specification must have been incomplete. More or less as a side-product of our approach to debug conflicts with the design intent, our method is also able to automatically compute a fix in the case of an incomplete specification: A guarantee that enforces the desired behavior is added to the specification in order to eliminate the incompleteness regarding the input trace which uncovered the incompleteness in the first place.

Note that an implementation of the design is only assumed to be present since this is a common scenario where bugs are uncovered. Our debugging method itself does not require an implementation of the design to be available.

1.4 Structure of this Document

The rest of this document is structured in a top-down manner: We start by introducing a generic approach, concretize it for a certain class of specifications, discuss the implementation of this concretization, and finally present evaluation results obtained from this implementation. To be more precise, the subsequent chapters have the following contents.

Chapter 2 gives some definitions and establishes notation that is used throughout the rest of this document. It does not contain anything new.

In Chapter 3, we introduce our generic debugging approach. After a discussion of the prerequisites for its application, we illustrate how counterstrategies can be used to explain unrealizability. Furthermore, simplification of counterstrategies is addressed as well as their presentation. Finally, this chapter shows how counterstrategies can be used to explain other kinds of soundness problems as well.

Chapter 4 concretizes the generic debugging approach for the class of GR(1) specifications. Concrete algorithms for the different steps of the procedure to explain unrealizability are discussed. Furthermore, a definition of a counterstrategy for unrealizable GR(1) specifications is given. This definition is then used to construct a symbolic algorithm which computes such counterstrategies. Finally, a concrete construction that allows to explain conflicts between GR(1) specifications and the informal design intent is presented.

In Chapter 5, we introduce the implementation of the debugging concepts for GR(1) specifications. The most important features added to the tools Anzu [59] and RATSY are illustrated on an example. The example also shows which information the tools provide for the user, and how this information can be used for debugging. Furthermore, a brief overview of the software design is given.

We present an evaluation of our debugging concepts in Chapter 6. Since both Anzu and RATSY behave very similar, we only use RATSY for the evaluation. First, we analyze the performance of the different steps that are carried out by RATSY. Second, we investigate the explanations given by the tool for two examples. A discussion of the main outcomes is included as well.

Chapter 7 discusses related work and in which points our work differs therefrom. Chapter 8 concludes the document by summarizing and discussing the most important facts. Finally, proposals for future work are made.

2 Preliminaries

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [81] is one of the most popular logics used for the specification of reactive systems. It allows for the description of the time dependence of events. LTL formulas are constructed over a set P of atomic propositions. The syntax can be defined in the following way [35]:

- An atomic proposition $p \in P$ is a valid LTL formula.
- If φ and ψ are LTL formulas, then so are $\neg\varphi$, $\varphi \vee \psi$, $\mathsf{X}\,\varphi$ and $\varphi\,\mathsf{U}\,\psi$.

Let $\overline{\tau} = \tau_0\tau_1\tau_2\ldots \in \left(2^P\right)^\omega$ be an infinite trace over P. In this notation, ω denotes the set of non-negative integers as common for denoting infinite words (refer to the work of Farwer [43] for an introduction into this topic). Let $\overline{\tau}, i \models \varphi$ denote that the LTL formula φ holds at the point $i \in \mathbb{N}$ of $\overline{\tau}$. We further say that $\overline{\tau}$ satisfies a formula φ, denoted $\overline{\tau} \models \varphi$, iff $\overline{\tau}, 0 \models \varphi$. The semantics of an LTL formula can then be defined as follows [97]:

- $\overline{\tau}, i \models p$ for $p \in P$ iff $p \in \tau_i$,
- $\overline{\tau}, i \models \neg\varphi$ iff not $\overline{\tau}, i \models \varphi$,
- $\overline{\tau}, i \models \varphi \vee \psi$ iff $\overline{\tau}, i \models \varphi$ or $\overline{\tau}, i \models \psi$,
- $\overline{\tau}, i \models \mathsf{X}\,\varphi$ iff $\overline{\tau}, i+1 \models \varphi$, and
- $\overline{\tau}, i \models \varphi\,\mathsf{U}\,\psi$ iff for some $j \geq i$, the condition $\overline{\tau}, j \models \psi$ holds and for all k, $i \leq k < j$, the condition $\overline{\tau}, j \models \varphi$ holds.

The semantics of the operators $\neg$ and $\vee$ are defined as usual. Intuitively, a formula $\mathsf{X}\,\varphi$ is true iff φ is true in the next step. Hence, the operator X can be read as "next". A formula $\varphi\,\mathsf{U}\,\psi$ is true iff φ is true until ψ becomes true. Hence, U is read as "until". Other Boolean operators can be reduced to $\neg$ and $\vee$ in the usual way:

$$\begin{aligned}
\varphi \wedge \psi &= \neg(\neg\varphi \vee \neg\psi),\\
\varphi \Rightarrow \psi &= \neg\varphi \vee \psi,\\
\varphi \Leftrightarrow \psi &= (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi), \text{ and}\\
\varphi \oplus \psi &= (\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi).
\end{aligned}$$

Based on the operator U, some more temporal operators can be defined [97]:

- $\mathsf{F}\,\varphi = \mathsf{true}\,\mathsf{U}\,\varphi$, where the operator F is read as "eventually" or "finally". Intuitively, $\mathsf{F}\,\varphi$ is true iff φ will become true at some point in the future.
- $\mathsf{G}\,\varphi = \neg\,\mathsf{F}\,\neg\varphi$, where the operator G is read as "globally" or "henceforth". Intuitively, $\mathsf{G}\,\varphi$ is true iff φ will be true at any point in the future.

2.1.1 Quantified Propositional Temporal Logic

Quantified Propositional Temporal Logic (QPTL) [90] is an extension of LTL that allows to quantify over propositional variables. The syntax can be defined as for LTL with the additional rule:

- If φ is an LTL formula, then so is $\exists p\,.\,\varphi$ for a propositional variable $p \in P$.

The rules defining the semantics of LTL are extended by the additional rule:

- $\overline{\tau}, i \models \exists p \,.\, \varphi$ for $p \in P$ iff $\overline{\tau}', i \models \varphi$ for some p-variant $\overline{\tau}'$ of $\overline{\tau}$. A trace $\overline{\tau}' = \tau_0'\tau_1'\tau_2' \in \left(2^P\right)^\omega$ is a p-variant of $\overline{\tau} = \tau_0\tau_1\tau_2 \in \left(2^P\right)^\omega$ iff $\exists p_0 p_1 p_2 \in \left(2^{\{p\}}\right)^\omega \,.\, \forall i \geq 0 \,.\, \tau_i' = \tau_i^\exists$, where $\tau_i^\exists = (\tau_i \setminus \{p\}) \cup p_i$.

The universal quantification can be reduced to an existential quantification in the usual way, i.e, with the equality $\forall p \,.\, \varphi = \neg\exists p \,.\, \neg\varphi$. We will furthermore handle $\exists P' \,.\, \varphi$ with $P' \subseteq P$ as an abbreviation for the existential quantification of all propositional variables $p \in P'$ in φ.

2.2 Generalized Reactivity

The class of *Generalized Reactivity of Rank 1* [80] formulas, abbreviated as GR(1), forms a subset of LTL. A GR(1) formula φ defines the allowed interaction between a system and its environment. The system controls a set Y of Boolean output variables and the environment controls a set X of Boolean input variables. A GR(1) formula φ can be written as [80]

$$\varphi = \varphi^e \rightarrow \varphi^s = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \rightarrow \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s,$$

where the parts are defined below.

- φ_i^e and φ_i^s are Boolean formulas over the sets X and Y of variables.
- φ_t^e is a formula of the form $\bigwedge_{i \in I} \mathsf{G}\, B_i$ where each B_i is a Boolean combination of variables from $X \cup Y$ and expressions $\mathsf{X}\, v$ where $v \in X$.
- φ_t^s is a formula of the form $\bigwedge_{i \in I} \mathsf{G}\, B_i$ where each B_i is a Boolean combination of variables from $X \cup Y$ and expressions $\mathsf{X}\, v$ where $v \in X \cup Y$.
- φ_g^e and φ_g^s are formulas of the form $\bigwedge_{i \in I} \mathsf{G}\,\mathsf{F}\, B_i$ where each B_i is a Boolean formula.

Intuitively, the part φ^e defines assumptions about the environment and φ^s defines guarantees provided by the system. If the environment fulfills all the assumptions, the implication in the formula requires the system to fulfill all the guarantees. The formulas φ_i^e and φ_i^s characterize the initial state of the environment and the system, respectively. The term φ_t^e determines allowed next input values when given the present input and output values. With φ_t^s, allowed next output values are specified, given the present variable values and the next input values. The formulas φ_g^e and φ_g^s define fairness constraints for the environment and the system, respectively. These are conditions that have to be fulfilled infinitely often.

2.3 Automata

2.3.1 Finite ω-Automata

A *finite ω-automaton* $\mathcal{A}$ is a 5-tuple $\mathcal{A} = (Q, \Sigma, \Delta, q_0, \mathrm{Acc})$, where Q is a finite set of states, Σ is a finite input alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $q_0 \in Q$ is the initial state, and $\mathrm{Acc} : Q^\omega \rightarrow \{\mathsf{false}, \mathsf{true}\}$ is the acceptance condition [94]. A finite ω-automaton is called *complete* iff its transition relation is complete, i.e., iff [37]

$$\forall q \in Q, \sigma \in \Sigma \,.\, \left|\left\{q' \in Q \mid (q, \sigma, q') \in \Delta\right\}\right| \geq 1.$$

A finite ω-automaton is called *deterministic* iff its transition relation is deterministic, i.e., iff [37]

$$\forall q \in Q, \sigma \in \Sigma \,.\, \left|\left\{q' \in Q \mid (q, \sigma, q') \in \Delta\right\}\right| \leq 1.$$

In case of a deterministic automaton, the transition relation Δ is replaced by a transition function $\delta : Q \times \Sigma \to Q$ [94].

A *run* $\overline{r}$ of a deterministic automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathrm{Acc})$ on a given ω-word $\overline{\sigma} = \sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$ is an infinite sequence $\overline{r} = r_0r_1r_2 \ldots \in Q^\omega$ of states such that $r_0 = q_0$ and $r_{i+1} = \delta(r_i, \sigma_i)$ for all $i \geq 0$. The run is *accepting* iff $\mathrm{Acc}(\overline{r}) = \mathsf{true}$ [94].

A *deterministic and complete Büchi word automaton* (DBW) is a deterministic and complete automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathrm{Acc})$ in which the acceptance condition Acc is given by a set of accepting states $F \subseteq Q$. Let $\inf(\overline{r})$ denote the set of states which occur infinitely often in $\overline{r}$. Then Acc is defined as

$$\mathrm{Acc}(\overline{r}) \Leftrightarrow \inf(\overline{r}) \cap F \neq \emptyset.$$

That is, a run $\overline{r}$ of a DBW is accepting iff some accepting state $f \in F$ occurs infinitely often in that run $\overline{r}$ [94].

In the following, we assume that $Q = 2^V$ for a set V of state bits. We furthermore suppose that $\Sigma = 2^X \times 2^Y$, with X and Y being sets of Boolean signals. With these assumptions, automata can be represented symbolically as BDDs [12]. Symbolic representations are often more space efficient than explicit ones, especially if the state space is large. To simplify notation, we will use the abbreviations $\mathcal{X} = 2^X$ and $\mathcal{Y} = 2^Y$. For an input trace $\overline{x} = x_0x_1 \ldots \in \mathcal{X}^\omega$ and an output trace $\overline{y} = y_0y_1 \ldots \in \mathcal{Y}^\omega$, we define $\overline{x}||\overline{y} = (x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$ to denote their combination.

2.3.2 Finite ω-Automata with Output

A *Mealy machine* [55] is a six-tuple $M = (Q, \mathcal{X}, \mathcal{Y}, \delta, \lambda, q_0)$, where Q is a finite set of states, $\mathcal{X}$ is an input alphabet, $\mathcal{Y}$ is an output alphabet, $\delta : Q \times \mathcal{X} \to Q$ is a transition function, $\lambda : Q \times \mathcal{X} \to \mathcal{Y}$ is an output function, and $q_0 \in Q$ is the initial state. When given an input trace $x_0x_1x_2 \ldots \in \mathcal{X}^\omega$, the output trace produced by the Mealy machine is defined as $\lambda(q_0, x_0)\lambda(q_1, x_1)\lambda(q_2, x_2) \ldots \in \mathcal{Y}^\omega$, where $q_0q_1q_2 \ldots \in Q^\omega$ is a sequence of states such that $q_{i+1} = \delta(q_i, x_i)$ for all $i \geq 0$.

Likewise, a *Moore machine* [55] is a six-tuple $M = (Q, \mathcal{Y}, \mathcal{X}, \delta, \lambda, q_0)$, where Q is a finite set of states, $\mathcal{Y}$ is an input alphabet, $\mathcal{X}$ is an output alphabet, $\delta : Q \times \mathcal{Y} \to Q$ is a transition function, $\lambda : Q \to \mathcal{X}$ is an output function, and $q_0 \in Q$ is the initial state. Note that, in comparison to the definition of the Mealy machine, we have swapped the letters $\mathcal{Y}$ and $\mathcal{X}$. This makes further definitions less confusing, because Moore machines will be combined with Mealy machines so that the input of the one is the output of the other. When given an input trace $\overline{y} = y_0y_1y_2 \ldots \in \mathcal{Y}^\omega$, the output trace produced by the Moore machine M is defined as $M(\overline{y}) = \lambda(q_0)\lambda(q_1)\lambda(q_2) \ldots \in \mathcal{X}^\omega$, where $q_0q_1q_2 \ldots \in Q^\omega$ is a sequence of states such that $q_{i+1} = \delta(q_i, y_i)$ for all $i \geq 0$. Finally, we denote with $L(M) = \{\overline{x}||\overline{y} \in (\mathcal{X} \times \mathcal{Y})^\omega \mid M(\overline{y}) = \overline{x}\}$ the set of words that can be produced by the Mealy machine M.

2.4 Games

Similar to Piterman et al. [80], we define a *game* as a tuple $\mathcal{G} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$. The elements of this tuple are defined as for deterministic ω-automata. That is, Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $\mathrm{Win} : Q^\omega \to \{\mathsf{false}, \mathsf{true}\}$ is the winning condition. We further require that $\Sigma = \mathcal{X} \times \mathcal{Y}$, which means that each letter in the alphabet is composed of an input letter and an output letter.

The game is played by two players, Player 1 and Player 2. When we talk about games for reactive systems, we will also refer to Player 1 as the *environment* and to Player 2 as the *system*. A *play* $\overline{\pi}$ of $\mathcal{G}$ is defined as an infinite sequence of states $\overline{\pi} = q_0q_1q_2 \ldots \in Q^\omega$ such that $q_{i+1} = \delta(q_i, \sigma_i)$ for all $i \geq 0$. The letters $\sigma_i = (x_i, y_i)$ are chosen by the two players in cooperation. In each step, Player 1 first chooses an $x_i \in \mathcal{X}$ and then Player 2 chooses some $y_i \in \mathcal{Y}$. Hence, Player 1 behaves as a Moore machine

and Player 2 as a Mealy machine (see Section 2.3.2). A play is won by Player 2 iff $\mathrm{Win}(\overline{\pi}) = \mathsf{true}$. Otherwise, it is lost for Player 2 and won for Player 1.

2.4.1 Strategies

We further define a *(finite memory) strategy* for Player 1 on $\mathcal{G} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ to be a tuple $\varrho = (\Gamma, \gamma_0, \rho)$. The element Γ is some finite set representing the memory, and $\gamma_0 \in \Gamma$ is the initial memory content. The last element ρ of the tuple is a relation $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$ such that $\forall q \in Q, \gamma \in \Gamma \,.\, \exists x \in \mathcal{X}, \gamma' \in \Gamma \,.(q, \gamma, x, \gamma') \in \rho$. This relation contains tuples (q, γ, x, γ') that determine possible next inputs $x \in \mathcal{X}$ and next memory contents $\gamma' \in \Gamma$, given a certain state $q \in Q$ of the game and a certain memory content $\gamma \in \Gamma$. A strategy is *deterministic* iff $\forall q \in Q, \gamma \in \Gamma \,.\, |\{(q, \gamma, x, \gamma') \in \rho\}| \leq 1$. Otherwise it is *non-deterministic*. A play $\overline{\pi} = q_0 q_1 q_2 \ldots \in Q^\omega$ *conforms to a strategy* ϱ, denoted $\overline{\pi} \sqsubseteq \varrho$, iff in all time steps the input x_i is chosen such that $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$. More formally, conformance with a strategy is defined as

$$\overline{\pi} \sqsubseteq \varrho \Leftrightarrow \exists \gamma_0 \gamma_1 \gamma_2 \ldots \in \Gamma^\omega \,.\, \forall i \geq 0 \,.\, \exists (x_i, y_i) \in \Sigma \,.\, \delta(q_i, (x_i, y_i)) = q_{i+1} \wedge (q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho.$$

Similarly, $\overline{\pi} \sqsubseteq \overline{x}$ denotes that a play $\overline{\pi} = q_0 q_1 q_2 \ldots \in Q^\omega$ *conforms to an input trace* $\overline{x} = x_0 x_1 x_2 \ldots \in \mathcal{X}^\omega$. This conformance relation is defined as

$$\overline{\pi} \sqsubseteq \overline{x} \Leftrightarrow \forall i \geq 0 \,.\, \exists y_i \in \mathcal{Y} \,.\, \delta(q_i, (x_i, y_i)) = q_{i+1}.$$

A strategy ϱ is *winning from a state* $q \in Q$ for Player 1 iff all plays that start in q and conform to ϱ are won by Player 1. The *winning region* $W \subseteq Q$ of Player 1 is the set of all states for which a winning strategy for Player 1 exists. A strategy is *winning* for Player 1 iff it is winning from q_0. A winning strategy for Player 1 will also be called *counterstrategy* in the following, as it enforces the negated winning condition $\neg\,\mathrm{Win}$.

2.4.2 Implementation of a Strategy

Let $\mathcal{G} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ be a game with $\Sigma = \mathcal{X} \times \mathcal{Y}$, and let $\varrho = (\Gamma, \gamma_0, \rho)$ with $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$ be a deterministic strategy for Player 1 in this game. Then this strategy can be implemented as a Moore machine $M = (Q^M, \mathcal{Y}, \mathcal{X}, \delta^M, \lambda^M, q_0^M)$, where

- $Q^M = Q \times \Gamma$,
- $\delta^M : Q^M \times \mathcal{Y} \to Q^M$ such that $\delta^M((q, \gamma), y) = (\delta(q, (x, y)), \gamma')$, where x and γ' are chosen such that $(q, \gamma, x, \gamma') \in \rho$,
- $\lambda^M : Q^M \to \mathcal{X}$ such that $\lambda((q, \gamma)) = x$, where x is chosen such that $\exists \gamma' \in \Gamma \,.(q, \gamma, x, \gamma') \in \rho$ holds, and
- $q_0^M = (q_0, \gamma_0)$.

We will denote this construction by $M = \mathcal{G} \times \varrho$. In the case of a non-deterministic strategy, we use the same notation and assume that the strategy is determinized before the above construction is applied.

2.4.3 GR(1) Games

A GR(1) specification φ with m environment assumptions and n system guarantees can always be expressed with $m + n$ DBWs [80]. In the following, we assume that a GR(1) specification is represented in this form. If the GR(1) specification is given in terms of LTL formulas (cf. Section 2.2), a transformation into sets of DBWs is straight forward. The DBWs representing the environment assumptions will be denoted $\mathcal{A}_i^e = (Q_i^e, \Sigma, \delta_i^e, q_{0,i}^e, F_i^e)$, and the DBWs representing the system guarantees will be referred to as $\mathcal{A}_j^s = (Q_j^s, \Sigma, \delta_j^s, q_{0,j}^s, F_j^s)$ in the following. All these DBWs operate with the same alphabet $\Sigma = \mathcal{X} \times \mathcal{Y}$.

We define a *GR(1) game* $\mathcal{G}^{\mathrm{GR}(1)}$ to be the tuple $(Q, \Sigma, \delta, q_0, \mathrm{Win})$, which is determined as the product of all DBWs $\mathcal{A}_i^e$ and $\mathcal{A}_j^s$ representing the GR(1) specification. The set of states Q of this product is given by

$$Q = Q_1^e \times \cdots \times Q_m^e \times Q_1^s \times \cdots \times Q_n^s.$$

The alphabet Σ is again $\Sigma = \mathcal{X} \times \mathcal{Y}$. The transition function $\delta : Q \times \Sigma \to Q$ is defined as

$$\delta\left((q_1^e, \ldots, q_m^e, q_1^s, \ldots, q_n^s), \sigma\right) = \left(\delta_1^e\left(q_1^e, \sigma\right), \ldots, \delta_m^e\left(q_m^e, \sigma\right), \delta_1^s\left(q_1^s, \sigma\right), \ldots, \delta_n^s\left(q_n^s, \sigma\right)\right).$$

The initial state of the product is determined by the equation $q_0 = (q_{0,1}^e, \ldots, q_{0,m}^e, q_{0,1}^s, \ldots, q_{0,n}^s)$. Let $J_i^e = \{(q_1^e, \ldots, q_m^e, q_1^s, \ldots, q_n^s) \mid q_i^e \in F_i^e\}$ be the set of all states of the game $\mathcal{G}^{\mathrm{GR}(1)}$ that are accepting in $\mathcal{A}_i^e$. Analogously, let $J_j^s = \{(q_1^e, \ldots, q_m^e, q_1^s, \ldots, q_n^s) \mid q_j^s \in F_j^s\}$ be the set of all states of $\mathcal{G}^{\mathrm{GR}(1)}$ that are accepting in $\mathcal{A}_j^s$. Then, the winning condition Win for the GR(1) game $\mathcal{G}^{\mathrm{GR}(1)}$ is given by

$$\begin{aligned} \mathrm{Win}\left(\overline{\pi}\right) &\Leftrightarrow \left(\forall i \,.\, \inf(\overline{\pi}) \cap J_i^e \neq \emptyset\right) \Rightarrow \left(\forall j \,.\, \inf(\overline{\pi}) \cap J_j^s \neq \emptyset\right) \\ &\Leftrightarrow \neg\left(\forall i \,.\, \inf(\overline{\pi}) \cap J_i^e \neq \emptyset\right) \vee \left(\forall j \,.\, \inf(\overline{\pi}) \cap J_j^s \neq \emptyset\right). \end{aligned}$$

Intuitively, a play of the game $\mathcal{G}^{\mathrm{GR}(1)}$ is won by the system (by Player 2) iff all sets J_j^s of accepting states of the system are visited infinitely often, or some set J_i^e of accepting states of the environment is visited only finitely often. The sets J_i^e represent the environment assumptions and the sets J_j^s represent the system guarantees. The game $\mathcal{G}^{\mathrm{GR}(1)}$ hence represents the entire GR(1) specification φ from which it was constructed: A play resulting from an infinite word $\overline{\sigma} \in \Sigma^\omega$ is won by the system iff $\overline{\sigma} \models \varphi$, i.e., iff $\overline{\sigma}$ satisfies the GR(1) specification φ.

2.5 μ-Calculus

The (propositional) μ-calculus [64] can be seen as an extension of a temporal logic with a least fixpoint operator μ and a greatest fixpoint operator ν [39]. We will extend the propositional μ-calculus with some more operators and use it on games. Just like Piterman et al. [80], we additionally allow two different mixed preimage operators MX^{e} and MX^{s} in μ-calculus expressions. Furthermore, we will make use of an image operator IMG.

2.5.1 Syntax

Let $\mathcal{G} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ be a game with $\Sigma = \mathcal{X} \times \mathcal{Y}$, as defined in Section 2.4. Furthermore, let Var be a set of variables, each representing a specific subset of Q. The syntax of a propositional μ-calculus formula can then be defined as following [39]:

- A state $q \in Q$ is a valid μ-calculus formula.
- A variable $Y \in \mathrm{Var}$ is a valid μ-calculus formula.
- If p is a μ-calculus formula, then $\mathsf{EX}\, p$ is a μ-calculus formula.
- If p is a μ-calculus formula, then $\neg p$ is a μ-calculus formula.
- If p and q are μ-calculus formulas, then $p \wedge q$ is a μ-calculus formula.
- Let $Y \in \mathrm{Var}$ be some variable and let p be a μ-calculus formula that is syntactically monotone in Y, i.e., all occurrences of Y in p fall under an even number of negations. Then $\mu Y.p$ and $\nu Y.p$ are μ-calculus formulas.

We slightly extend this definition with two *mixed preimage* operators, one for the environment and one for the system [80]:

- If p is a μ-calculus formula, then $\mathsf{MX}^{\mathrm{e}}\, p$ and $\mathsf{MX}^{\mathrm{s}}\, p$ are μ-calculus formulas.

We furthermore allow an *image* operator IMG in μ-calculus formulas:

- If p is a μ-calculus formula, then so is $\mathsf{IMG}\,p$.

2.5.2 Semantics

Let $\mathcal{G} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ be a game with $\Sigma = \mathcal{X} \times \mathcal{Y}$, as defined in Section 2.4. A μ-calculus formula p represents a set of states in which p is true. We will use the same notation as Piterman et al. [80], where $[[p]]^e \subseteq Q$ denotes this set of states and $e : \mathrm{Var} \to 2^Q$ is an environment assigning subsets of Q to each variable $Y \in \mathrm{Var}$. Furthermore, the environment is denoted by $e[X \leftarrow S]$ in the way that $e[X \leftarrow S](X) = S$ and $e[X \leftarrow S](Y) = e(Y)$ for $X \neq Y$. Utilizing this notation, the set of states $[[p]]^e$ in which the μ-calculus formula p is true can be defined inductively as following [80]:

- $[[q]]^e = \{q\}$ for $q \in Q$.
- $[[Y]]^e = e(Y)$ for $Y \in \mathrm{Var}$.
- $[[\mathsf{EX}\,p]]^e = \{q \in Q \mid \exists \sigma \in \Sigma\,.\,\delta(q, \sigma) \in [[p]]^e\}$. Intuitively, a state $q \in Q$ is in $[[\mathsf{EX}\,p]]^e$ iff there exists a letter $\sigma \in \Sigma$, so that a state of $[[p]]^e$ is reached from that state q. Therefore, the operator EX applied to p gives all states from which a state of $[[p]]^e$ can be reached in one step when both players cooperate.
- $[[\neg p]]^e = Q \setminus [[p]]^e$.
- $[[p \wedge q]]^e = [[p]]^e \cap [[q]]^e$.
- The least fixpoint operator μ is defined as

$$[[\mu Y.p]]^e = \bigcup_i Y_i \text{ with } Y_0 = \emptyset \text{ and } Y_{i+1} = [[p]]^{e[Y \leftarrow Y_i]}. \tag{2.1}$$

  We will refer to the intermediate values Y_i as the *iterates* of the fixpoint. Clearly, the computation of these iterates can be stopped at iteration j if $Y_j = Y_{j-1}$. All further iterates Y_{j+a} with $a > 0$ would be equal to Y_{j-1} anyway. Including them into the union over all iterates would not change the result.

- The greatest fixpoint operator ν is defined as

$$[[\nu Y.p]]^e = \bigcap_i Y_i \text{ with } Y_0 = Q \text{ and } Y_{i+1} = [[p]]^{e[Y \leftarrow Y_i]}. \tag{2.2}$$

  The same considerations as for the least fixpoint operator concerning the computation of the iterates apply.

- $[[\mathsf{MX}^{\mathrm{e}}\,p]]^e = \{q \in Q \mid \exists x \in \mathcal{X}\,.\,\forall y \in \mathcal{Y}\,.\,\delta(q, (x, y)) \in [[p]]^e\}$. That is, a state $q \in Q$ is in $[[\mathsf{MX}^{\mathrm{e}}\,p]]^e$ iff there exists an input $x \in \mathcal{X}$ so that for all outputs $y \in \mathcal{Y}$ a state in $[[p]]^e$ is reached from that state q. Thus, $[[\mathsf{MX}^{\mathrm{e}}\,p]]^e$ is the set of states from which the environment can force a play into a state of $[[p]]^e$ in one step.
- $[[\mathsf{MX}^{\mathrm{s}}\,p]]^e = \{q \in Q \mid \forall x \in \mathcal{X}\,.\,\exists y \in \mathcal{Y}\,.\,\delta(q, (x, y)) \in [[p]]^e\}$. That is, a state $q \in Q$ is in $[[\mathsf{MX}^{\mathrm{s}}\,p]]^e$ iff for every input $x \in \mathcal{X}$ there exists some outputs $y \in \mathcal{Y}$ such that a state in $[[p]]^e$ is reached from that state q. Thus, $[[\mathsf{MX}^{\mathrm{s}}\,p]]^e$ is the set of states from which the system can force a play into a state of $[[p]]^e$ in one step.
- $[[\mathsf{IMG}\,p]]^e = \{q' \in Q \mid \exists q \in [[p]]^e\,.\,\exists \sigma \in \Sigma\,.\,q' = \delta(q, \sigma)\}$. Hence, the operator IMG applied to p gives all states that can be reached from $[[p]]^e$ in one step if both players cooperate.

With slight abuse of notation, we also allow subsets $J \subseteq Q$ in a μ-calculus formula and define $[[J]]^e = J$ for $J \subseteq Q$. We furthermore allow other Boolean connectivities than $\neg$ and $\wedge$ by handling them as abbreviations for their reduction to $\neg$ and $\wedge$ (see also Section 2.1).

Each occurrence of a variable $Y \in \mathrm{Var}$ in a sub-formula $\mu Y.p(Y)$ or $\nu Y.p(Y)$ is said to be *bound*. All other occurrences are called *free*. A μ-calculus formula is a *sentence* (or a *closed formula*) if it contains no free variables [39]. We will only use sentences in this document, and hence, the initial environment $e : \mathrm{Var} \to 2^Q$ is irrelevant for the evaluation of a μ-calculus formula. With slight abuse of notation, we will therefore simply write $[[p]]$ instead of $[[p]]^e$ in the following.

As obvious from their definition, the two mixed preimage operators can be transformed into each other by the rules

$$[[\mathsf{MX}^{\mathrm{s}}\, p]] = [[\neg\, \mathsf{MX}^{\mathrm{e}}\, \neg p]] \text{ and} \tag{2.3}$$
$$[[\mathsf{MX}^{\mathrm{e}}\, p]] = [[\neg\, \mathsf{MX}^{\mathrm{s}}\, \neg p]]\,. \tag{2.4}$$

For the fixpoint operators, we have the dualities [39]

$$[[\neg \mu Y\,.\, p(Y)]] = [[\nu Y\,.\, \neg p(\neg Y)]] \text{ and} \tag{2.5}$$
$$[[\neg \nu Y\,.\, p(Y)]] = [[\mu Y\,.\, \neg p(\neg Y)]]\,. \tag{2.6}$$

2.6 Synthesis of GR(1) Specifications

Piterman et al. [80] describe a synthesis procedure for GR(1) specifications. Since our method to compute counterstrategies for unrealizable GR(1) specifications (see Section 4.3) is based on their work, we will briefly explain this synthesis procedure for the case that all environment assumptions and system guarantees are given as DBWs.

Let $\mathcal{G}^{\mathrm{GR}(1)} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ be a game obtained from a GR(1) specification as shown in Section 2.4.3. In a first step, the winning region of the system, i.e., the set of all states from which a winning strategy for the system exists, is computed. In a second step, some intermediate results obtained during the computation of this winning region are used to build up the strategy for the system.

2.6.1 Computation of the Winning Region for the System

Using the notation introduced so far in this chapter, the winning region $W_{\mathrm{sys}}^{\mathrm{GR}(1)}$ of the system can be defined as

$$W_{\mathrm{sys}}^{\mathrm{GR}(1)} = \left[\left[\nu Z\,.\, \bigwedge_{j=1}^{n} \mu Y\,.\, \bigvee_{i=1}^{m} \nu X\,.\, J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y \vee \neg J_i^e \wedge \mathsf{MX}^{\mathrm{s}}\, X\right]\right].$$

The greatest fixpoint in X computes all states from which the system can enforce that the play either stays in $[[\neg J_i^e]]$ or eventually reaches $[[J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y]]$. Both cases are winning for the system. If the play stays in $[[\neg J_i^e]]$ forever, then an environment assumption is violated. The disjunction over all values of i captures the fact that it is sufficient for the system to win when one environment assumption is violated. The fixpoints in Y and Z ensure that a play can be won from any state of $[[J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y]]$. The least fixpoint in Y ensures together with the conjunction over all j that each set $J_{j'}^s$ can be reached from any state of $[[J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y]]$ in a finite number of steps. The greatest fixpoint in Z ensures that after visiting a certain J_j^s, the next set $J_{j\oplus 1}^s$ can be reached, where $j \oplus 1 = (j \mod n) + 1$. Hence, the game can be won from all states in $[[J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y]]$ by visiting all sets J_j^s infinitely often, i.e., by fulfilling all guarantees.

Intermediate Results for the Strategy Computation

The required intermediate results are the sets $Y_{j,r}$ and $X_{j,r,i}$ for all values of $j \in \{1, 2, \ldots, n\}$, $r \in \{0, 1, \ldots, R_j\}$, and $i \in \{1, 2, \ldots, m\}$. All these sets are subsets of Q. The set $Y_{j,r}$ is the r-th iterate

(according to Equation 2.1) of the fixpoint

$$Y_j = \left[\left[\mu Y \,.\, \bigvee_{i=1}^{m} \nu X \,.\, J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y \vee \neg J_i^e \wedge \mathsf{MX}^{\mathrm{s}}\, X\right]\right],$$

where Z is the final value of the variable Z in the computation of $W_{\mathrm{sys}}^{\mathrm{GR}(1)}$, i.e., the set $W_{\mathrm{sys}}^{\mathrm{GR}(1)}$ itself. The maximum value R_j of r is the smallest integer b such that $Y_{j,b} = Y_{j,b-1}$. The set $X_{j,r,i}$ is defined to be

$$X_{j,r,i} = \left[\left[\nu X \,.\, J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y_{j,r-1} \vee \neg J_i^e \wedge \mathsf{MX}^{\mathrm{s}}\, X\right]\right],$$

where Z is again the final value of the variable Z, i.e., the set $W_{\mathrm{sys}}^{\mathrm{GR}(1)}$. To simplify notation, we also introduce

$$\begin{aligned} Y_{j,r}^{\mathrm{new}} &= Y_{j,r} \setminus Y_{j,r-1} \text{ and} \\ X_{j,r,i}^{\mathrm{new}} &= X_{j,r,i} \setminus \left(\bigcup_{(r',i') \prec (r,i)} X_{j,r',i'} \right), \end{aligned}$$

where $(r', i') \prec (r, i)$ iff $r' < r \vee (r' = r \wedge i' < i)$.

2.6.2 Computation of the Winning Strategy for the System

Similar to the definition of a strategy for the environment (Player 1) in Section 2.4, a strategy for the system (Player 2) in the GR(1) game $\mathcal{G}^{\mathrm{GR}(1)} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ can be defined as a tuple $\varrho_{\mathrm{sys}}^{\mathrm{GR}(1)} = (\Gamma, \gamma_0, \rho)$, where Γ is some finite set representing the memory, and γ_0 is the initial memory content. The relation $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \mathcal{Y} \times \Gamma)$ maps a state $q \in Q$ of the game, a current memory content $\gamma \in \Gamma$, and a given input letter $x \in \mathcal{X}$ to an output letter $y \in \mathcal{Y}$ and an updated memory content $\gamma' \in \Gamma$. Note the difference to the definition of a strategy for the environment: the strategy for the system takes an input letter as an additional argument. This difference is due to the fact that the environment is a Moore machine while the system is a Mealy machine (cf. Section 2.3.2).

Piterman et al. define the memory $\Gamma = \{1, 2, \ldots n\}$ to store the index j of the set J_j^s of accepting states of the system which should be reached next. The initial memory content is in fact irrelevant, so we arbitrarily define $\gamma_0 = 1$. The relation ρ is composed of the three sub-strategies ρ_1, ρ_2, and ρ_3 in the way that $\rho = \rho_1 \cup \rho_2 \cup \rho_3$. The sub-strategies are defined below.

Sub-strategy ρ_1 is applied if the the play has reached the next target set J_j^s of accepting states of the system:

$$\rho_1 = \left\{(q, j, x, y, j \oplus 1) \in (Q \times \Gamma \times \mathcal{X} \times \mathcal{Y} \times \Gamma) \mid q \in W_{\mathrm{sys}}^{\mathrm{GR}(1)} \cap J_j^s \wedge \delta(q, (x, y)) \in W_{\mathrm{sys}}^{\mathrm{GR}(1)}\right\}$$

The next goal is to reach the set $J_{j\oplus 1}^s$, so the value of j is updated accordingly.

Sub-strategy ρ_2 is an attractor strategy forcing the play ever closer to the next target set J_j^s:

$$\rho_2 = \left\{(q, j, x, y, j) \in (Q \times \Gamma \times \mathcal{X} \times \mathcal{Y} \times \Gamma) \mid \exists r > 1 \,.\, q \in Y_{j,r}^{\mathrm{new}} \wedge \delta(q, (x, y)) \in Y_{j,r-1}\right\}$$

The strategy ρ_2 is applied if the next target set J_j^s is not yet reached. The value of j remains unchanged as the system is still heading for the same target. From a state $q \in Y_{j,r}^{\mathrm{new}}$, the system can force a play into a state of J_j^s in at most $r - 1$ steps. The strategy forces the play into a state of $Y_{j,r-1}$, from which the set J_j^s can be reached in at most $r - 2$ steps. The set $Y_{j,1}$ is reached eventually if ρ_2 is continuously applied. All states of $Y_{j,1}$ are also in J_j^s, so the target is reached eventually.

Sub-strategy ρ_3 is a strategy to force the environment to violate an assumption:

$$\rho_3 = \big\{(q, j, x, y, j) \in (Q \times \Gamma \times \mathcal{X} \times \mathcal{Y} \times \Gamma) \mid \\ \exists r > 1, i \in \{1 \dots m\} \,.\, q \in X^{\text{new}}_{j,r,i} \setminus J^e_i \wedge \delta(q, (x, y)) \in X_{j,r,i}\big\}$$

If the play is in an iterate $X^{\text{new}}_{j,r,i}$ and not in J^e_i, the strategy ρ_3 forces the play to remain in $X_{j,r,i}$. The continuous application of ρ_3 ensures that an environment assumption is violated.

2.6.3 Synthesis from GR(1) Specifications that are Given with LTL Formulas

In the previous subsections we described the synthesis algorithm as introduced by Piterman et al. [80] for the case that all environment assumptions and system guarantees are given as DBWs. However, the algorithm does not only work for DBWs and games as defined in Section 2.4.3. In the work of Piterman et al., the definition of a game is more general. It allows to use LTL formulas directly in the construction of the game. The formulas do not have to be transformed into DBWs beforehand. We briefly explain how games are defined in the work of Piterman et al., because our implementation (see Section 5.1) works with this definition.

Essentially, the definition of the game in the work of Piterman et al. [80] differs from our definition in two points. First, they define $Q = \mathcal{X} \times \mathcal{Y}$, and second, they use two transition relations ρ_e and ρ_s instead of the transition function δ used in our framework. The relation $\rho_e \subseteq (Q \times \mathcal{X})$ maps states of the game to possible next input letters. It is built from the safety assumptions (the part φ^e_t; see Section 2.2) of the specification. The relation $\rho_s \subseteq (Q \times \mathcal{X} \times \mathcal{Y})$ maps states of the game and next input letters to possible next output letters. It is built from the safety guarantees (the part φ^s_t) of the specification. Furthermore, the fairness assumptions (the part φ^e_g) and guarantees (the part φ^s_g) are used to define the sets J^e_i and J^s_j, and the formulas φ^e_i and φ^s_i of the specification are used to characterize the initial state of the game.

For a game defined in this way, the synthesis procedure works basically in the same way as already described. Only two minor modifications are necessary. First, the semantics of the μ-calculus operators MX^{e} and MX^{s} have to be adopted. They have to be redefined to

- $[[\mathsf{MX}^{\mathrm{e}}\, p]]^e = \{q \in Q \mid \exists x' \in \mathcal{X} \,.\, (q, x') \in \rho_e \wedge \forall y' \in \mathcal{Y} \,.\, (q, x', y') \in \rho_s \Rightarrow (x', y') \in [[p]]^e\}$ and
- $[[\mathsf{MX}^{\mathrm{s}}\, p]]^e = \{q \in Q \mid \forall x' \in \mathcal{X} \,.\, (q, x') \in \rho_e \Rightarrow \exists y' \in \mathcal{Y} \,.\, (q, x', y') \in \rho_s \wedge (x', y') \in [[p]]^e\}$.

Second, the transition function δ has to be replaced by the transition relations ρ_e and ρ_s in the definition of the sub-strategies ρ_1, ρ_2, and ρ_3. That is, the sub-strategies have to be redefined to

$$\rho_1 = \big\{((x, y), j, x', y', j \oplus 1) \in (Q \times \Gamma \times \mathcal{X} \times \mathcal{Y} \times \Gamma) \mid \\ (x, y) \in W^{\mathrm{GR}(1)}_{\text{sys}} \cap J^s_j \wedge ((x, y), x') \in \rho_e \wedge ((x, y), x', y') \in \rho_s \wedge (x', y') \in W^{\mathrm{GR}(1)}_{\text{sys}}\big\},$$

$$\rho_2 = \big\{((x, y), j, x', y', j) \in (Q \times \Gamma \times \mathcal{X} \times \mathcal{Y} \times \Gamma) \mid \\ \exists r > 1 \,.(x, y) \in Y^{\text{new}}_{j,r} \wedge ((x, y), x') \in \rho_e \wedge ((x, y), x', y') \in \rho_s \wedge (x', y') \in Y_{j,r-1}\big\}, \text{ and}$$

$$\rho_3 = \big\{((x, y), j, x', y', j) \in (Q \times \Gamma \times \mathcal{X} \times \mathcal{Y} \times \Gamma) \mid \exists r > 1, i \in \{1 \dots m\} \,. \\ (x, y) \in X^{\text{new}}_{j,r,i} \setminus J^e_i \wedge ((x, y), x') \in \rho_e \wedge ((x, y), x', y') \in \rho_s \wedge (x', y') \in X_{j,r,i}\big\}.$$

2.7 Delta Debugging

Delta Debugging [102] is a method to isolate the trigger of a failure. Given a test case that causes a program to fail, Delta Debugging can be used to simplify it to a minimal test case which still results in the failure of the program. The algorithm performing this minimization was first introduced by Zeller [101], then improved and further investigated in subsequent publications [31; 102]. In this document, we refer to the algorithm as defined in [102].

2.7.1 Definition of the Algorithm

Given a set C that fails some test, the Delta Debugging algorithm computes a minimal subset $\hat{C} \subseteq C$ that still fails the test. We will write $\hat{C} = ddmin(C)$ to denote the minimization algorithm, where C is some set that should be minimized, and $\hat{C} \subseteq C$ is the minimization result. Furthermore, let $test(C') = \text{✘}$ denote that C' fails the test, let $test(C') = \text{✔}$ mean that C' passes the test, and let $test(C') = \text{?}$ denote that $test$ gives an indeterminate result. An indeterminate result can be returned by $test$, for example, if a syntax error occurred, or if a failure was triggered which is different to the one that should be isolated. The algorithm requires that $test(C) = \text{✘}$ and that $test(\emptyset) = \text{✔}$. Under these conditions, it guarantees to return a set $\hat{C} = ddmin(C)$ such that $test(\hat{C}) = \text{✘}$. The algorithm is defined as

$$
\begin{aligned}
ddmin(C) &= ddmin_2(C, 2) \text{ with} \\
ddmin_2(C', n) &= \begin{cases}
ddmin_2\left(C'_i, 2\right) & \text{if } \exists i \,.\, test\left(C'_i\right) = \text{✘} \\
ddmin_2\left(\overline{C'_i}, \max(n-1, 2)\right) & \text{else if } \exists i \,.\, test\left(\overline{C'_i}\right) = \text{✘} \\
ddmin_2\left(C', \min\left(|C'|, 2n\right)\right) & \text{else if } n < |C'| \\
C' & \text{otherwise,}
\end{cases}
\end{aligned}
$$

where $\overline{C'_i} = C' \setminus C'_i$ and the sets $C'_1, \ldots, C'_n$ form a partition of C' into n parts. The size of all these parts is approximately equal.

The algorithm for $ddmin_2(C', n)$ works recursively. The first step is called *reduce to subset*. It tries to find a subset $C'_i \subseteq C'$ that still fails the test. The granularity n determines the size of the examined subsets. If such a subset C'_i that fails the test could be found, it is further reduced by a recursive call of $ddmin_2$ with the lowest possible granularity. If no such subset could be found, the *reduce to complement* step is performed. It checks if a complement $\overline{C'_i} = C' \setminus C'_i$ of a subset $C'_i \subseteq C'$ examined in the previous step fails the test. If so, then this complement $\overline{C'_i}$ is again further reduced by a recursive call of $ddmin_2$. If not, the algorithm doubles the granularity. If the granularity cannot be increased any further, the currently examined set C' is returned as a minimal subset failing the test. See Section 3.2.1 for an example.

2.7.2 Properties of the Algorithm

In the best case, half of the elements of the set are removed with every call to the function $test$. The number of tests is then logarithmic in the size of the set C that should be minimized. In the worst case, however, the number of tests is quadratic in the size of the input (see the work of Zeller and Hildebrandt [102] for an explanation and the proof).

In order to find a minimal subset $\hat{C} \subseteq C$ that still fails the test, the algorithm handles subsets that give inconclusive test results (i.e., subsets $C' \subseteq C$ such that $test(C') = \text{?}$) as though they would have passed the test. For the purpose of our work, we can assume that inconclusive test results do not occur, i.e., that

$$test(C') \neq \text{✔} \Leftrightarrow test(C') = \text{✘}.$$

In the rest of this document, we implicitly assume that the above property holds. This simplifies the reasoning about the algorithm.

The algorithm guarantees that the computed subset $\hat{C} = ddmin(C)$ is *1-minimal*, meaning that $\forall c \in \hat{C} \,.\, test(\hat{C} \setminus \{c\}) = \text{✔}$ holds [102]. Without additional assumptions, the algorithm does not ensure that $\hat{C}$ is a local minimum in the sense that $\forall C' \subset \hat{C} \,.\, test(C') = \text{✔}$ holds. A local minimum is however obtained if $test$ is monotonic, i.e., if

$$\forall C'' \subseteq C' \subseteq C \,.\, test(C') = \text{✔} \Rightarrow test(C'') = \text{✔}.$$

The monotonicity of $test$ obviously implies that every 1-minimal set is also a local minimum. Monotonicity can also be exploited to increase the performance of the implementation: if a superset of a set C' has already passed the test, then so will C'. The (maybe computationally expensive) call to the function $test$ with argument C' does not have to be performed.

3 Debugging Approach

This chapter describes our approach for debugging formal specifications. After outlining some prerequisites, it explains how unrealizable specifications and specifications that allow undesired behavior can be debugged.

3.1 Prerequisites

Our approach is not specific to any particular kind of specification. However, in our setting we expect the specification to meet the following requirements:

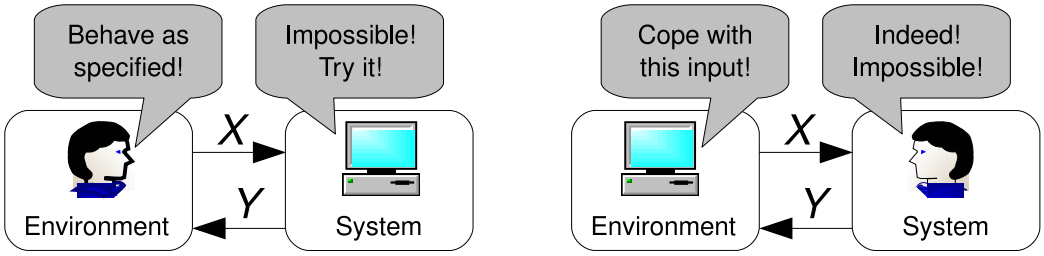
1. The specification defines the temporal behavior of a reactive system, i.e., it defines the allowed interaction between a system and its environment (see also Figure 1.2). The system communicates with its environment over a set of output signals Y and a (possibly empty) set of input signals X. Without loss of generality, we assume that all signals are Boolean.
2. The specification is of the form $\varphi = A \rightarrow G$, where A is a (possibly empty) set of environment assumptions and G is some set of system guarantees. The specification requires the system behavior to fulfill all guarantees if the environment behavior is conform with all assumptions.
3. It is possible to add guarantees to the specification and to remove guarantees from the specification.
4. It is possible to existentially quantify an output $y \in Y$ in all system guarantees. The existential quantification must have the effect that all restrictions on that output y are nullified. After quantification, the system is allowed to choose the value for the output y completely arbitrarily in every time step without any consequences for other outputs.
5. A decision procedure for realizability is available. Given a concrete instance of a specification, this procedure is able to find out whether or not the specification is realizable.
6. A synthesis procedure for a counterstrategy is available. Given an unrealizable specification, this procedure is able to compute a counterstrategy as defined in Section 2.4.1.
7. It is possible to turn the specification into a game as defined in Section 2.4.

These assumptions are rather weak and apply to various widespread specification languages such as LTL and subsets thereof. For one particular subset of LTL, namely for the class of GR(1) specifications, the debugging approach will be elaborated in Chapter 4.

3.2 Debugging Unrealizability

We present an interactive approach for explaining unrealizability. It is based on the following idea: While creating a formal specification for a reactive system, the user must have an imagination of an implementation of the system in his mind. When the user finds out that her specification is unrealizable, we show that the imagined implementation does not conform to her formal specification. This is done by swapping roles as shown in Figure 3.1. The user takes on the role of the system while the debugging tool takes on the role of the environment. The tool provides inputs and the user tries to provide outputs conforming to the specification. The tool uses a counterstrategy to find inputs such that the user is forced to violate the specification. Since the specification is unrealizable, such a counterstrategy always exists and the user is bound to fail. However, while trying, she gains insights into why there is no way for her to fulfill the specification, i.e., why the specification is unrealizable. This knowledge can subsequently be used to correct the specification.

We extend this basic approach as shown in Figure 3.2. This is done to keep the explanations for unrealizability simple. First, we check for satisfiability. If a specification is unsatisfiable, trace-based



(a) Simulating the System **(b)** Debugging Unrealizability

Figure 3.1: Swapping the roles to gain insight into the cause of unrealizability: The user takes on the role of the system and fails to fulfill the specification when the environment utilizes a counterstrategy to find problematic inputs. While failing, she will understand where the specification is too restrictive to be realizable.

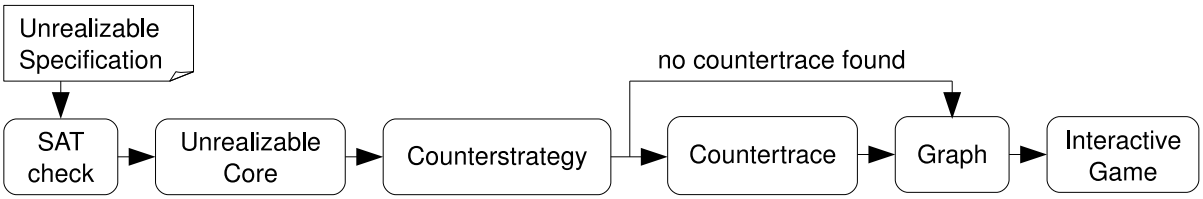


Figure 3.2: The flow of our method to explain unrealizability: After a SAT-check, we generate an unrealizable core by removing requirements and signals that do not contribute to the problem. A counterstrategy and a countertrace are then computed and presented to the user in form of a graph and as an interactive game.

debugging methods (as presented by Pill et al. [79] or by Cimatti et al. [22], for instance) can be applied. Such methods are likely to produce simpler explanations. Nevertheless, as unsatisfiability is just a special case of unrealizability, our method is able to handle both cases. Next, we compute an unrealizable core as suggested by Cimatti et al. [21]. That is, we remove parts of the specification which do not have hand in the unrealizability problem. We improve the work of Cimatti et al. by removing not only unnecessary properties but also unnecessary signals. Moreover, we use Delta Debugging [102] as minimization algorithm with the goal of achieving a better performance. Next, we compute a counterstrategy for the minimized specification. We then try to obtain a *countertrace* from it. A countertrace is a fixed input trace for which there is no output trace that conforms to the specification. Knowing the complete input trace in advance makes it easier for the user to localize the problem. Unfortunately, such a countertrace does not always exist (see Section 3.2.2). Additionally, even if one exists, its computation is expensive. Thus, we present a heuristic algorithm that is fast but does not always find a countertrace, even if one exists. The obtained countertrace or the counterstrategy is finally presented to the user as a summarizing graph and in form of an interactive game.

The following subsections explain the different steps of this debugging procedure in detail. All aspects which are specific to a particular kind of specification language are discussed in Chapter 4 for the class of GR(1) specifications.

3.2.1 Minimization

Finding the cause of unrealizability in a specification becomes especially difficult if the specification is large. Nevertheless, in a large unrealizable specification, it will often be the case that big parts are not involved in the conflict causing the specification to be unrealizable. These parts can be removed,

resulting in a simpler specification $\hat{\varphi}$ that is still unrealizable. A game with a counterstrategy can then be constructed from this simplified specification. The idea is that this game becomes easier to understand as a lot of restrictions that are not directly connected with the unrealizability problem have been removed. The user can focus on the restrictions that really cause the problem. Any counterstrategy for the simplified specification $\hat{\varphi}$ is also a counterstrategy for the original specification (see Theorem 5), and hence, it illustrates a problem of the original specification.

Parts to be Removed from a Specification

Cimatti et al. [21] propose to minimize both environment assumptions and system guarantees in order to find an unrealizable core that can be used for diagnostics. In a first step, system guarantees $g \in G$ are removed as long as removing them preserves unrealizability. This leads to an intermediate specification $\varphi' = A \rightarrow G'$ with $G' \subseteq G$. In a second step, environment assumptions $a \in A$ are removed as long as removing them preserves minimality of G', i.e., as long as removing them does not allow the algorithm to remove any guarantee $g \in G'$ without ending up with a realizable specification.

Our goal is not to obtain a specification $\hat{\varphi}$ that is as short as possible, but to simplify the underlying game as much as possible. Hence, we only remove system guarantees. Removing assumptions would confuse the user during the interactive game as it allows behavior of the environment which the user before forbade. We do not want the counterstrategy to exploit originally forbidden behavior to win the game. Also, removing assumptions such that the set of guarantees remains a minimal set is computationally expensive.

Our experience shows that removing guarantees does not suffice to obtain simple games. In fact, the corresponding game becomes often more difficult to understand. The reason is that removing guarantees leads to more possibilities for the user in the role of the system to choose outputs. The game graph is getting larger and there are more possible plays. The user potentially has to play more often to appreciate that there is no way for her to win any of these possible plays.

To counteract this effect, we propose to additionally remove output signals which are not involved in the conflict causing unrealizability. Removing them is done with an existential quantification in all guarantees. We denote this operation $\exists Y' \,.\, G$ for some set $Y' \subseteq Y$ of outputs to remove. Let $\overline{\tau} = (x_0, y_0)(x_1, y_1)(x_2, y_2) \ldots \in (\mathcal{X} \times \mathcal{Y})^\omega$ be an infinite trace over an input alphabet $\mathcal{X}$ and an output alphabet $\mathcal{Y}$, where the input alphabet is $\mathcal{X} = 2^X$ and the output alphabet is $\mathcal{Y} = 2^Y$ for a set X of Boolean input signals and a set Y of Boolean output signals. Let $\overline{\tau} \models G$ denote that $\overline{\tau}$ fulfills all system guarantees $g \in G$. We define the semantics of the existential quantification operation $\exists Y' \,.\, G$ such that for all traces $\overline{\tau} = (x_0, y_0)(x_1, y_1)(x_2, y_2) \ldots \in (\mathcal{X} \times \mathcal{Y})^\omega$ the equivalence

$$\overline{\tau} \models \exists Y' \,.\, G \Leftrightarrow \exists y'_0 y'_1 y'_2 \in \left(2^{Y'}\right)^\omega .(x_0, y_0^\exists)(x_1, y_1^\exists)(x_2, y_2^\exists) \ldots \models G$$

holds, where $y_i^\exists$ is an abbreviation for $(y_i \setminus Y') \cup y'_i$ for all $i \geq 0$. Note the similarity of this definition to the semantics of the existential quantifier in QPTL formulas (see Section 2.1.1). In fact, if each guarantee $g \in G$ is given as an LTL formula, the quantification operation $\exists Y' \,.\, G$, with $Y' \subseteq Y$, means that each guarantee $g \in G$ is replaced by the QPTL formula $\exists Y' \,.\, g$. If each guarantee $g \in G$ is given as a Büchi automaton, then the quantification $\exists Y' \,.\, G$ reduces to a projection of all output signals $y \in Y'$ from the automaton, i.e., to the existential quantification of the outputs $y \in Y'$ in the transition relation of the automaton. The resulting automaton may be non-deterministic.

Just like guarantees, outputs are removed as long as removing them preserves unrealizability. Suppose the output $y \in Y$ has been removed. This means for the game that the value of y can be chosen completely arbitrarily in any time step without any consequences for other outputs. Still, the specification is unrealizable, and hence, the output y is irrelevant for the unrealizability problem. Choosing a value for y completely arbitrarily in every time step is of course senseless, so the output is not included in the interactive game at all. As the user does not have to care about such outputs that are irrelevant for

the unrealizability problem, the game becomes simpler again. The user has less choices, and hence, she potentially has to play less often to accept that none of her choices can make her win.

The Minimization Algorithm

Different algorithms can be used to perform the minimization of the specification. Cimatti et al. [21] remove one guarantee after the other. If removing a guarantee makes the specification realizable, the guarantee is added again. Otherwise, the guarantee is kept removed. This procedure repeats until there has been an attempt to remove every single guarantee. This simple algorithm requires $|G|$ realizability checks to find an unrealizable core. We use Delta Debugging [102], a more advanced algorithm for minimization problems. As already mentioned in Section 2.7, this algorithm takes a set to be minimized as parameter and utilizes a function $test$. Suppose now that $realizable(\varphi)$ implements a decision procedure for realizability, i.e., that it returns true if φ is realizable and false otherwise. We define

$$test\left(G' \cup Y'\right) = \begin{cases} \checkmark & \text{if } realizable\left(A \rightarrow \exists\left(Y \setminus Y'\right).\, G'\right) \\ \times & \text{otherwise} \end{cases}$$

for $G' \subseteq G$ and $Y' \subseteq Y$. The minimized specification $\hat{\varphi}$ is finally computed as

$$\hat{\varphi} = A \rightarrow \exists\left(Y \setminus \hat{Y}\right).\, \hat{G}, \text{ with } \hat{Y} = ddmin(G \cup Y) \cap Y \text{ and } \hat{G} = ddmin(G \cup Y) \cap G.$$

Fundamental Properties

For the minimization procedure as defined above, the following properties can be observed.

Lemma 1. *For all $G'' \subseteq G' \subseteq G$ and $Y'' \subseteq Y' \subseteq Y$, we have that $realizable\left(A \rightarrow \exists(Y \setminus Y').\, G'\right)$ implies $realizable\left(A \rightarrow \exists(Y \setminus Y'').\, G''\right)$.*

Proof. Every system that implements a specification $\varphi' = A \rightarrow \exists(Y \setminus Y').\, G'$ also implements $\varphi'' = A \rightarrow \exists(Y \setminus Y'').\, G''$ for all $G'' \subseteq G' \subseteq G$ and $Y'' \subseteq Y' \subseteq Y$. Hence, if an implementation of φ' exists, then an implementation of φ'' exists as well. □

Corollary 2. *For all $C'' \subseteq C' \subseteq (Q \cup Y)$, the condition $test(C') = \checkmark$ implies that $test(C'') = \checkmark$.*

Claim 3. *The specification $A \rightarrow \exists\left(Y \setminus \hat{Y}\right).\, \hat{G}$ is unrealizable.*

Proof. All preconditions for $ddmin$ are fulfilled: $test(G \cup Y) = \times$ as $\varphi = A \rightarrow G$ is assumed to be unrealizable, and $test(\emptyset) = \checkmark$ as $realizable(A \rightarrow \mathsf{true})$ holds for all A. In this case, $ddmin(G \cup Y)$ guarantees to return a subset $\hat{C}$ such that $test(\hat{C}) = \times$ (cf. Section 2.7). With the definitions of $test$, $\hat{G}$, and $\hat{Y}$, this is exactly what Claim 3 affirms. □

Claim 4. *For all $G' \subseteq \hat{G}$ and $Y' \subseteq \hat{Y}$, we have that the condition $(G', Y') \neq \left(\hat{G}, \hat{Y}\right)$ implies $realizable\left(A \rightarrow \exists(Y \setminus Y').\, G'\right)$.*

Proof. Claim 4 states that the set $\hat{C} = \hat{G} \cup \hat{Y} = ddmin(G \cup Y)$ is a local minimum. The proof of Proposition 11 in [102] shows that $\hat{C}$ is 1-minimal. The function $test$ is monotonic (Corollary 2), hence, every 1-minimal set is also a local minimum (see also Section 2.7). □

Theorem 5. *Let $\varrho = (\Gamma, \gamma_0, \rho)$ be a counterstrategy for the game based on $\hat{\varphi}$. Then ϱ is also a counterstrategy for the game based on φ.*

Proof. Let $\mathcal{G}$ be the game obtained from φ, and let $\hat{\mathcal{G}}$ be the game obtained from $\hat{\varphi}$. Furthermore, let $M = \hat{\mathcal{G}} \times \varrho$ be a Moore machine implementing the counterstrategy as defined in Section 2.4.2. Since ϱ is a counterstrategy in $\hat{\mathcal{G}}$, $\forall \bar{t} \in L(M) \,.\, \bar{t} \not\models \hat{\varphi}$ holds, where $L(M)$ denotes the set of words that can be produced by M (see Section 2.3.2 for the definition). Clearly, $\forall \bar{t} \in L(M) \,.\, t \not\models \varphi$ holds as well, since φ is stricter than $\hat{\varphi}$, i.e., $\forall \bar{t} \in \Sigma^{\omega} \,.\, \bar{t} \not\models \hat{\varphi} \Rightarrow \bar{t} \not\models \varphi$ applies. Hence, M is also a valid implementation of a counterstrategy in the game $\mathcal{G}$ obtained from the original specification φ. □

Discussion of the Properties

Lemma 1 states that the realizability of a specification is preserved when guarantees or output variables are removed from the specification. This is clear, since removing them can never make it harder for the system to conform to the specification. Corollary 2 follows immediately from Lemma 1 and the definition of the function $test$. It states that $test$ is monotonic. The following optimization [102] can therefore be applied: During minimization, all examined sets $R = G' \cup Y'$ are stored, for which $test(R) = \text{✔}$ holds. If a subset R' of a stored set R is subjected to $test$, the value ✔ can be returned immediately without actually invoking the check for realizability. This has a great impact on the overall performance of the minimization algorithm.

Claim 3 states that the minimized specification $\hat{\varphi}$ obtained by applying Delta Debugging is still unrealizable. Claim 4 says that it contains a (locally) minimal set of guarantees and output signals so that $\hat{\varphi}$ is unrealizable. Theorem 5 finally relates the game based on the simplified specification $\hat{\varphi}$ to the game based on the original specification φ. The counterstrategy for the game based on $\hat{\varphi}$ also applies to the original game. This means that the conflict causing unrealizability, which is exploited by the counterstrategy, must have been preserved by the minimization step. (If there are more conflicts, at least one is preserved.) Thus, minimization is indeed useful for finding the conflict causing the unrealizability in the original specification.

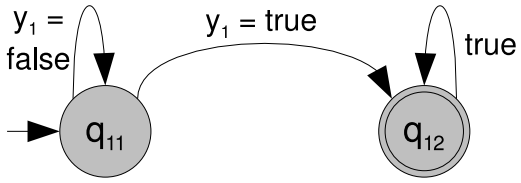
Example

Figure 3.3 depicts an example specification used to illustrate the minimization of a specification by the Delta Debugging algorithm. It contains three Boolean input signals x_1, x_2, and x_3, three Boolean output signals y_1, y_2, and y_3, and four system guarantees g_1, g_2, g_3, and g_4. All guarantees are represented as DBWs and as LTL formulas in Figure 3.3. The guarantees g_i, with $1 \leq i \leq 3$, enforce that the output signals y_i are true eventually. The guarantee g_4 requires that $y_i = x_i$ for $1 \leq i \leq 3$ in all time steps. There are no environment assumptions. This specification is clearly unrealizable since the environment could set an input x_i to false forever for some $1 \leq i \leq 3$. The guarantee g_4 requires y_i to be false in all time steps while guarantee g_i requires y_i to be true eventually. Thus, the system cannot fulfill all guarantees in such a case. However, the specification is satisfiable, because the trace where all inputs and outputs are true in all time steps fulfills the specification.

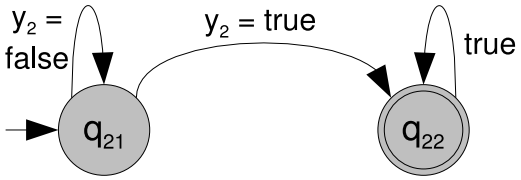
Table 3.1 illustrates the steps performed by the Delta Debugging algorithm. The first column contains a step counter, and the second column contains the current granularity n of the algorithm. The third column states which subset is subjected to the function $test$ (cf. Section 2.7), and the next column contains the elements of this subset. The last two columns finally contain the test result and whether a superset of the tested set already passed the test before.

The set that should be minimized is $G \cup Y = \{g_1, g_2, g_3, g_4, y_1, y_2, y_3\}$. A subset thereof represents an unrealizable specification if it contains the guarantee g_4 and some guarantee g_i together with the output y_i for $1 \leq i \leq 3$. Hence, the function $test$ returns ✘ iff $\{g_1, g_4, y_1\}$, $\{g_2, g_4, y_2\}$, $\{g_3, g_4, y_3\}$, or a superset of one of these sets is used as input.

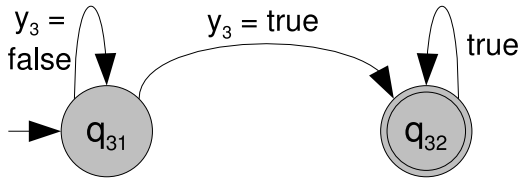
In the first two steps, the subsets with granularity 2 are tested. Both subsets pass the test, so the algorithm would normally proceed with the complements of these two sets. However, for the granularity of 2, the complements are equal to the sets themselves, so this step can be skipped. Next, the



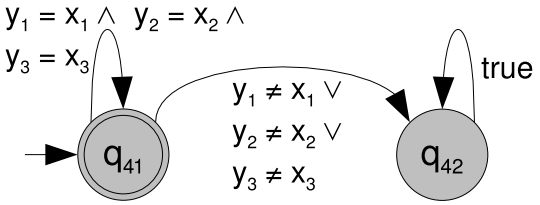
(a) The guarantee g_1 as DBW. It corresponds to the LTL formula $\mathsf{F}(y_1 = \mathsf{true})$.



(b) The guarantee g_2 as DBW. It corresponds to the LTL formula $\mathsf{F}(y_2 = \mathsf{true})$.



(c) The guarantee g_3 as DBW. It corresponds to the LTL formula $\mathsf{F}(y_3 = \mathsf{true})$.



(d) The guarantee g_4 as DBW. It corresponds to the LTL formula $\mathsf{G}(y_1 = x_1 \wedge y_2 = x_2 \wedge y_3 = x_3)$.

Figure 3.3: An example specification to illustrate the minimization with the Delta Debugging algorithm. It consists of the three Boolean input signals x_1, x_2, and x_3, of the three Boolean output signals y_1, y_2, and y_3, and of the four depicted system guarantees g_1, g_2, g_3, and g_4. There are no environment assumptions. Accepting states in DBWs are double bordered. The initial states are marked with incoming arrows.

algorithm doubles its granularity to the value of 4. In the steps 3 to 7, we can see that all subsets pass the test. The first complement, however, fails the test, so the algorithm tries to further minimizes the set $\{g_3, g_4, y_1, y_2, y_3\}$, starting with a granularity of 3. This procedure repeats in the steps 8 to 12 once more. The algorithm again finds a smaller set $\{g_3, g_4, y_3\}$ that fails the test. Since this set cannot be reduced any further in the steps 13 to 20, it is returned as the result.

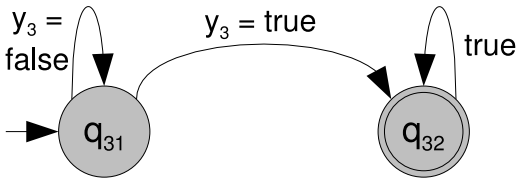
Note that not all steps performed by the Delta Debugging algorithm require a realizability check to be performed. As already mentioned, if a superset of the examined set has passed the test before, then the function $test$ can return ✔ without actually performing a realizability check. The last column of Table 3.1 indicates when this is the case. For this example, only 6 realizability checks have to be performed.

The specification that is represented by the minimization result is depicted in Figure 3.4. Existentially quantifying the outputs y_1 and y_2 in g_3 does not change g_3, since g_3 does not depend on these outputs. In contrast, the guarantee g_4 is simplified by the existential quantification. It now only requires that the output y_3 has to be equal to x_3 in all time steps. It does no longer restrict the outputs y_1 and y_2. If g_4 is given as a Büchi automaton, the projection gives a non-deterministic automaton.

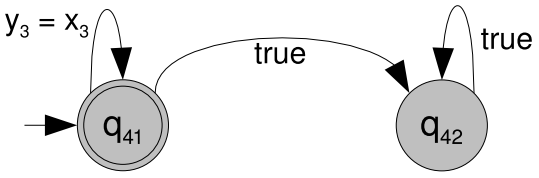
Once the minimization result is obtained, our debugging approach proceeds with the computation and illustration of a counterstrategy. The counterstrategy exploits the conflict between the guarantees g_3 and g_4 regarding the output y_3 by setting $x_3 = \mathsf{false}$ forever. In the interactive game, the user then has the task of finding a value sequence for the output y_3 which fulfills the guarantees g_3 and g_4 (see also Section 3.2.4, where the example is continued). She can ignore all other guarantees and outputs, which makes it easier to understand the problem. In addition, if the specification contains more than one conflict, the user is forced to focus on one of them. This is easier than trying to understand all conflicts at once, possibly by analyzing a counterstrategy that exploits several problems simultaneously.

Table 3.1: The steps performed by the Delta Debugging algorithm for the example in Figure 3.3.

step	n	subset	subset's content							test result	superset passed
1	2	$C'_1 = \overline{C'_2}$	g_1	g_2	g_3	g_4				✔	no
2	2	$C'_2 = \overline{C'_1}$					y_1	y_2	y_3	✔	no
3	4	C'_1	g_1	g_2						✔	yes
4	4	C'_2			g_3	g_4				✔	yes
5	4	C'_3					y_1	y_2		✔	yes
6	4	C'_4							y_3	✔	yes
7	4	$\overline{C'_1}$			g_3	g_4	y_1	y_2	y_3	✘	no
8	3	C'_1			g_3	g_4				✔	yes
9	3	C'_2					y_1	y_2		✔	yes
10	3	C'_3							y_3	✔	yes
11	3	$\overline{C'_1}$					y_1	y_2	y_3	✔	yes
12	3	$\overline{C'_2}$			g_3	g_4			y_3	✘	no
13	2	$C'_1 = \overline{C'_2}$			g_3	g_4				✔	yes
14	2	$C'_2 = \overline{C'_1}$							y_3	✔	yes
15	3	C'_1			g_3					✔	yes
16	3	C'_2				g_4				✔	yes
17	3	C'_3							y_3	✔	yes
18	3	$\overline{C'_1}$				g_4			y_3	✔	no
19	3	$\overline{C'_2}$			g_3				y_3	✔	no
20	3	$\overline{C'_3}$			g_3	g_4				✔	yes
result					g_3	g_4			y_3		



(a) The guarantee g_3 with the outputs y_1 and y_2 projected out. It corresponds to the QPLT formula $\exists\{y_1, y_2\} \,.\, \mathsf{F}(y_3 = \mathsf{true})$, which is equivalent to $\mathsf{F}(y_3 = \mathsf{true})$.



(b) The guarantee g_4 with the outputs y_1 and y_2 projected out. It corresponds to the QPLT formula $\exists\{y_1, y_2\} \,.\, \mathsf{G}(y_1 = x_1 \wedge y_2 = x_2 \wedge y_3 = x_3)$, which is equivalent to $\mathsf{G}(y_3 = x_3)$.

Figure 3.4: The minimization result for the example specification of Figure 3.3. Only the two guarantees g_3 and g_4, and one output, namely y_3, are remaining.

3.2.2 Countertraces

When a specification φ is unrealizable, a counterstrategy can be computed. A counterstrategy is a winning strategy for the environment. It dictates inputs so that the system cannot find outputs that conform to the specification. In every time step, the inputs given by the counterstrategy may depend on previous moves of the opponent, i.e., on previous output values chosen by the system. Hence, it can only be illustrated as a graph or as an interactive game. A counterstrategy would be easier to understand if all moves it dictates would be independent of the previous moves of the opponent. In this case, the counterstrategy could be illustrated with a single trace of inputs.

We therefore define a *countertrace* to be an infinite trace of inputs, for which no trace of outputs exists, such that the specification is fulfilled. More formally, we have that

$$\begin{aligned} \overline{x} \in \mathcal{X}^\omega \text{ is a countertrace for } \varphi &\Leftrightarrow \neg\exists \overline{y} \in \mathcal{Y}^\omega .(\overline{x}||\overline{y}) \models \varphi \\ &\Leftrightarrow \forall \overline{y} \in \mathcal{Y}^\omega .(\overline{x}||\overline{y}) \not\models \varphi. \end{aligned}$$

Problems with Countertraces

Unfortunately, there are two serious problems when trying to use countertraces instead of counterstrategies as explanations for the unrealizability of a specification.

First, a countertrace does not always exist. For example, consider the LTL specification $y \Leftrightarrow \mathsf{F}\, x$, where y is a Boolean output and x is a Boolean input [82]. This specification requires the output y to be true in the first time step iff the input x is true at some point in the future. Clearly, this specification is unrealizable. In order to implement it, the system would have to look into the future [82]. However, a counterstrategy exists. It could for instance dictate to set $x_0 = \mathsf{false}$ and $x_i = \neg y_0$ in all time steps $i > 0$. Yet, every counterstrategy for this example must react to the first move of the system in order to be winning for the environment. A countertrace, which is defined to be independent of the system's moves, does not exist. For every infinite trace of inputs, there is always an infinite trace of outputs so that both traces together fulfill the specification. Mori et al. [75] define a specification φ to be *strongly satisfiable* iff

$$\forall \overline{x} \in \mathcal{X}^\omega \,.\, \exists \overline{y} \in \mathcal{Y}^\omega .(\overline{x}||\overline{y}) \models \varphi.$$

We can use this definition to determine whether a countertrace exists.

Corollary 6. *A countertrace for a specification φ exists iff φ is not strongly satisfiable.*

Proof.

$$\begin{aligned} \varphi \text{ is not strongly satisfiable} &\Leftrightarrow \neg\forall \overline{x} \in \mathcal{X}^\omega \,.\, \exists \overline{y} \in \mathcal{Y}^\omega .(\overline{x}||\overline{y}) \models \varphi \\ &\Leftrightarrow \exists \overline{x} \in \mathcal{X}^\omega \,.\, \forall \overline{y} \in \mathcal{Y}^\omega .(\overline{x}||\overline{y}) \not\models \varphi. \\ &\Leftrightarrow \exists \overline{x} \in \mathcal{X}^\omega \,.\, \overline{x} \text{ is a countertrace for } \varphi \end{aligned}$$

□

As a second problem, the computation of a countertrace is expensive. Checking if a countertrace exists is already expensive. In order to perform such a check, one can remove all output variables from the game automaton with an existential quantification. The resulting automaton is then complemented. Unfortunately, this complementation causes an exponential blow-up of the state space, as the automaton is in general non-deterministic after quantification. We therefore define a heuristic to keep the computation of a countertrace feasible, even for larger specifications.

Heuristic Computation of Countertraces

Our heuristic to compute countertraces works as following. Let φ be an unrealizable specification, and let $\mathcal{G} = (Q, \mathcal{X} \times \mathcal{Y}, \delta, q_0, \mathrm{Win})$ be the game (as defined in Section 2.4) obtained from this specification. Furthermore, let $\varrho = (\Gamma, \gamma_0, \rho)$ be a counterstrategy for $\mathcal{G}$, where $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$. In order to obtain a countertrace, we compute two sequences in parallel. The first one is a sequence $\overline{\tau} = \tau_0\tau_1\tau_2\ldots$ of inputs $\tau_i \in \mathcal{X}$ being the resulting countertrace itself. The second one is a sequence $\overline{S} = S_0S_1S_2\ldots$ of sets $S_i \subseteq (Q \times \Gamma)$. Each set S_i contains exactly those pairs of states and memory contents that are possible after $\tau_0\tau_1\ldots\tau_{i-1}$ has been used as input. In the following, we will refer to such pairs $(q, \gamma) \in (Q \times \Gamma)$ as *situations* in order not to mix them up with states of the game. The computation starts with $S_0 = \{(q_0, \gamma_0)\}$. Further sets of situations are computed as

$$S_{i+1} = \big\{(q', \gamma') \;\big|\; \exists (q, \gamma) \in S_i \,.\, \exists y \in \mathcal{Y} \,.\, q' = \delta(q, (\tau_i, y)) \wedge (q, \gamma, \tau_i, \gamma') \in \rho\big\},$$

where τ_i is chosen arbitrarily from the set

$$T_i = \big\{\tau \in \mathcal{X} \;\big|\; \forall (q, \gamma) \in S_i \,.\, \exists (\gamma' \in \Gamma) \,.(q, \gamma, \tau, \gamma') \in \rho\big\}.$$

The set T_i contains all inputs τ that conform to the counterstrategy from all situations $(q_i, \gamma_i) \in S_i$. The exact situation $(q_i, \gamma_i) \in S_i$ in step i depends on the outputs chosen by the system in all previous time steps. In every time step, we choose $\tau_i \in T_i$, so the input τ_i conforms to the counterstrategy no matter how the system moved in earlier steps.

If $T_i = \emptyset$ for any i, the computation aborts signaling that no countertrace was found. The algorithm terminates with success in step k if $S_k \subseteq S_j$ for some $j < k$. This makes sense, because $S_k \subseteq S_j$ implies $T_k \supseteq T_j$. We can choose $\tau_k = \tau_j$, which leads to a set $S_{k+1} \subseteq S_{j+1}$ in the next step. Again, $T_{k+1} \supseteq T_{j+1}$, so we can choose $\tau_{k+1} = \tau_{j+1}$, and so on. Obviously, the countertrace $\overline{\tau}$ starts to repeat after step k. It is finally composed of the finite stem $\tau_0\tau_1\ldots\tau_{j-1}$ and infinite many repetitions of $\tau_j\tau_{j+1}\ldots\tau_{k-1}$.

Example

Figure 3.5 illustrates the working principle of the heuristic on an example. In this example, there are two possible input letters x_A and x_B, and two possible output letters y_A and y_B. Arrows labeled with input letters represent moves of the environment which comply with the counterstrategy. If there is no outgoing arrow labeled with a certain input letter, then this indicates that the input letter does not conform to the counterstrategy in this situation. Arrows labeled with output letters represent possible moves of the system.

Step 0: The computation starts with the set $S_0 = \{(q_0, \gamma_0)\}$ containing only the initial situation. From (q_0, γ_0), the counterstrategy $\varrho = (\Gamma, \gamma_0, \rho)$ allows only x_B as next input letter, i.e., $\exists\gamma' \,.(q_0, \gamma_0, x_B, \gamma') \in \rho$ and $\nexists\gamma' \,.(q_0, \gamma_0, x_A, \gamma') \in \rho$. As a consequence, $T_0 = \{x_B\}$, so x_B is the only possible choice for τ_0. When x_B is used as input in Step 0, we might end up in (q_1, γ_1) or (q_2, γ_2), depending on the output letter chosen by the system. The set S_1 is therefore $\{(q_1, \gamma_1), (q_2, \gamma_2)\}$.

Step 1: The resulting countertrace must be winning for the environment, independent of which output letter is chosen by the system in Step 0. Hence, the next input τ_1 must conform to the counterstrategy, no matter if the situation (q_1, γ_1) or (q_2, γ_2) occurs in Step 1. Both input letters, x_A and x_B, conform to the counterstrategy from (q_1, γ_1) in our example. From (q_2, γ_2), the counterstrategy only allows x_B as next input. The set T_1 is thus $\{x_B\}$, meaning that x_B is the only input conforming to the counterstrategy from all elements of S_1. We can thus only set $\tau_1 = x_B$. Depending on the choice of the system on the next output letter, we might end up in (q_3, γ_1) or (q_4, γ_2) in Step 2, so $S_2 = \{(q_3, \gamma_1), (q_4, \gamma_2)\}$. The situation (q_2, γ_3) is not possible as we used x_B and not x_A as input in Step 1.

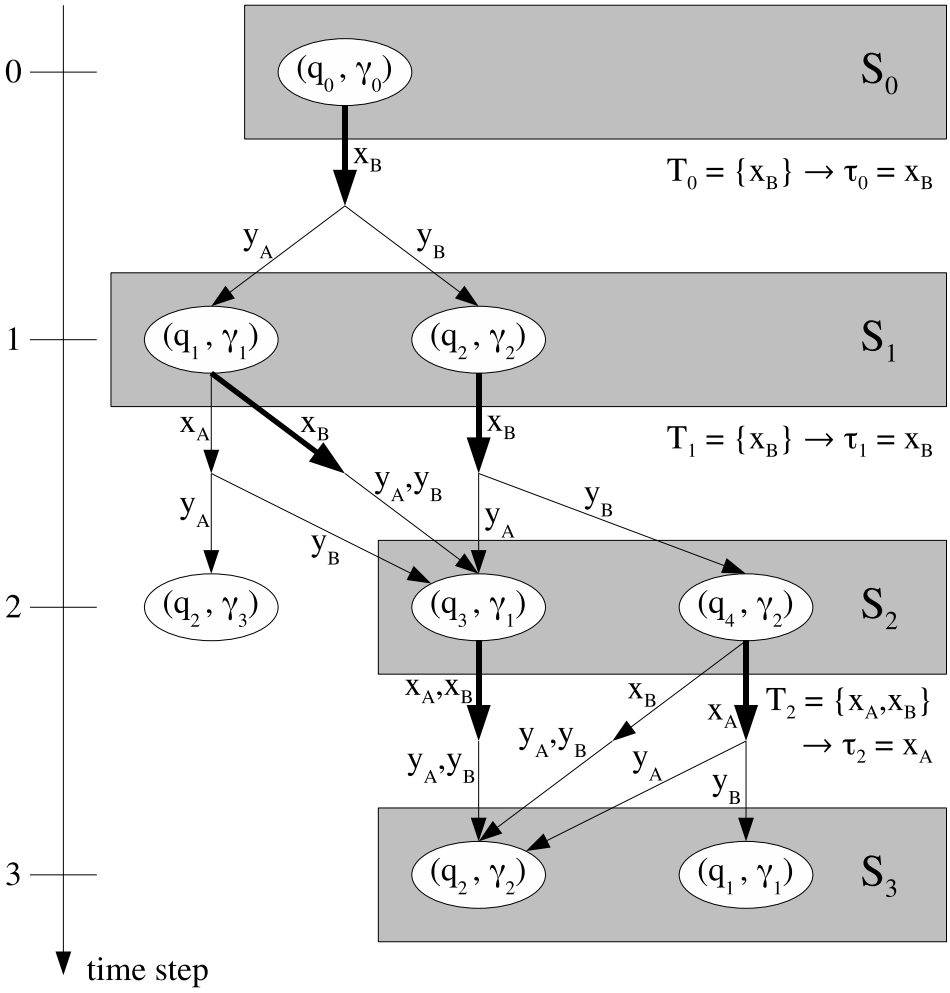


Figure 3.5: An example to illustrate the heuristic for computing countertraces: When x_A and x_B are the only input letters, and y_A and y_B are the only output letters, our heuristic returns $\overline{\tau} = x_B x_B x_A x_B x_A x_B x_A \ldots$ as countertrace for this example.

Step 2: We are now again looking for an input τ_2 which conforms to the countertrace from all situations $(q, \gamma) \in S_2$. The counterstrategy allows both, x_A and x_B, from both situations (q_3, γ_1) and (q_4, γ_2). Hence, $T_2 = \{x_A, x_B\}$ meaning that we could choose either $\tau_2 = x_A$ or $\tau_2 = x_B$ as next input. Suppose that we decide for $\tau_2 = x_A$. Depending on the output letter chosen by the system, the play might be in (q_2, γ_2) or (q_1, γ_1) in Step 3.

Step 3: We have that $S_3 = \{(q_1, \gamma_1), (q_2, \gamma_2)\} \subseteq S_1$, so we can stop the computation. We can do so, because we can set $\tau_3 = \tau_1$ ending up in $S_4 \subseteq S_2$, where we could choose $\tau_4 = \tau_2$, and so on. The resulting countertrace $\overline{\tau}$ is composed of the finite stem τ_0 followed by infinitely many repetitions of the sequence $\tau_1\tau_2$. This gives $\overline{\tau} = x_B x_B x_A x_B x_A x_B x_A \ldots$ for our example.

Analysis of Fundamental Properties

Even if a countertrace exists, our heuristic is not always able to find one. There are two reasons for that.

First, the counterstrategy from which the countertrace is constructed might not embody all ways to force the system to violate the specification. To give an example, refer to Figure 3.5 again. Suppose that from the situation (q_1, γ_1), the input x_A would be the only input conforming to the counterstrategy ϱ.

Then T_1 would be the empty set and our heuristic would fail. Suppose further that a second counterstrategy ϱ' exists, that this counterstrategy ϱ' allows the input x_A from the initial situation (q_0, γ_0) as well, and that our heuristic would succeed if τ_0 would be chosen to be x_A. This is an example where our heuristic fails because of the counterstrategy ϱ not being aware of all ways to win.

Second, our heuristic may fail due to a bad choice of an input letter $\tau_i \in T_i$. The heuristic chooses one such input randomly. The selection influences the next set S_{i+1} and can so also influence the success of the heuristic. The algorithm could be extended with a backtracking mechanism that makes a different selection in a previous time step if it cannot succeed with the undertaken selection. This would exclude failure due to a bad choice of an input letter, but it would not make the heuristic complete.

Although our heuristic does not always find a countertrace even if one exists, our experiments (see Section 6) will show that it works well for many real-world examples. Furthermore, the algorithm can be implemented symbolically, which is especially straight forward if the game $\mathcal{G}$ and the counterstrategy ϱ are encoded symbolically (e.g., using BDDs).

Claim 7. *The number k of iterations needed by the algorithm is $2^{|Q\times\Gamma|} - 1$ in the worst case.*

Proof. All sets S_i are elements of the power set $2^{Q\times\Gamma}$, only the empty set is not possible for any S_i. The power set $2^{Q\times\Gamma}$ consists of $2^{|Q\times\Gamma|}$ elements. Hence, there are $z = 2^{|Q\times\Gamma|} - 1$ different values for the sets S_i. The algorithm computes a sequence $\overline{S} = S_0 S_1 S_2 \ldots$ and aborts at Step k if $\exists j < k \,.\, S_k \subseteq S_j$. (It may also abort at Step l if $T_l = \emptyset$, but as we are considering the worst case for the execution time, we assume that this does not happen.) It is possible to sort all elements of the power set $2^{Q\times\Gamma}$ to obtain a sequence $\overline{S_{MAX}} = S_0 S_1 S_2 \ldots S_{z-1}$ such that $\forall i \,.\, \nexists j < i \,.\, S_i \subseteq S_j$ holds. (Simply start with all sets containing only one pair (q, γ), proceed with all sets containing exactly two pairs, etc.) However, another element of $2^{Q\times\Gamma}$ cannot be added to the sequence $\overline{S_{MAX}}$ without already having an equal element in the sequence. □

Claim 7 states that our heuristic has an exponential execution time in the worst case. To overcome that, searching for a countertrace can be bounded to a certain number t of iterations. That is, if no countertrace was found after t iterations, the algorithm aborts without success. According to our experiments (see Section 6), t can be kept rather small without decreasing the quality of the heuristic significantly: None of our computations needed more than 10 iterations.

Theorem 8. *Every play $\overline{\pi}$ conforming to the countertrace $\overline{\tau}$ also conforms to the counterstrategy $\varrho = (\Gamma, \gamma_0, \rho)$ and is thus won by the environment.*

Proof. In every time step i, the input τ_i is a singleton subset of the inputs allowed by ρ. This is obvious from the construction of $\overline{\tau}$. □

Theorem 8 finally states that a countertrace $\overline{\tau}$ obtained from a counterstrategy ϱ explains unrealizability, as it forces the system to violate the specification, just like a counterstrategy. The countertrace can thus be used instead of a counterstrategy in the interactive game.

3.2.3 Interactive Game

The idea of using an interactive game in order to demonstrate the unrealizability of a specification to the user has already been introduced. The tool takes on the role of the environment and the user takes on the role of the system. In this way, the user can try to demonstrate that a system exists, which implements the specification. The tool can demonstrate that the user is wrong and that the implementation imagined by the user does not conform to the specification. In every time step of the interactive game, the tool applies the counterstrategy or the countertrace to find values for the inputs of the system. Then the user chooses values for the outputs and the next time step starts. A play is won for the user if she fulfills the specification. It is lost by the user and won by the tool otherwise. As the utilized counterstrategy or

countertrace is winning for the environment, the user will not be able to find outputs that conform to the specification. However, while playing, she will find out where the specification is too restrictive for her to win, i.e., where the specification is too restrictive to be realizable.

Using Counterstrategies

More technically speaking, the interactive game works in the following way when given a counterstrategy $\varrho = (\Gamma, \gamma_0, \rho)$. Every play starts in (q_0, γ_0). When the play is in a situation (q_i, γ_i), the tool simply selects one tuple $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ and uses x_i as input. The next memory content γ_{i+1} is already determined with this selection. With the output letter y_i chosen by the user, the next state $q_{i+1} = \delta(q_i, (x_i, y_i))$ is fixed as well. From the situation (q_{i+1}, γ_{i+1}), the play proceeds in the same way.

In order not to confuse the user, the behavior of the environment should be deterministic throughout the play. That is, when the same situation (q, γ) is encountered again, the environment should also give the same input letter. This can be ensured by making the counterstrategy deterministic before starting the game. An explicit determinization step is however not necessary. Determinization can also be done implicitly during the interactive game: Given a certain situation (q_i, γ_i), a tuple $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ is simply chosen in some deterministic way.

Using Countertraces

Instead of a counterstrategy, the tool can also use a countertrace to determine the values of the input signals. The tool maintains a step counter and uses the value τ_i of the countertrace $\overline{\tau} = \tau_0\tau_1\tau_2 \ldots \in \mathcal{X}^\omega$ as input in step i. The moves performed by the environment are then independent of the moves carried out by the user. This makes it easier for the user to stay on top of things in the game. The whole countertrace is printed before the play starts. This helps the user, as she knows in advance how the environment will behave in all further steps. Nevertheless, there is no way for her to win.

When a play engine is used which is only able to utilize counterstrategies but no countertraces, a countertrace can be put into the shape of a strategy in the following way. Let $\overline{\tau}$ be a countertrace composed of a finite stem $\tau_0\tau_1 \ldots \tau_{j-1}$ and infinite many repetitions of $\tau_j\tau_{j+1} \ldots \tau_{k-1}$. A counterstrategy $\varrho^{\overline{\tau}} = \left(\Gamma^{\overline{\tau}}, \gamma_0^{\overline{\tau}}, \rho^{\overline{\tau}}\right)$ can then be defined, where

$$
\begin{aligned}
\Gamma^{\overline{\tau}} &= \{0, 1, \ldots k-1\}, \\
\gamma_0^{\overline{\tau}} &= 0 \text{ , and} \\
\rho^{\overline{\tau}} &= \left\{(q, \gamma, x, \gamma') \in \left(Q, \Gamma^{\overline{\tau}}, \mathcal{X}, \Gamma^{\overline{\tau}}\right) \mid \gamma' = next(\gamma) \wedge x = \tau_\gamma\right\} \text{ , with} \\
next(\gamma) &= \begin{cases} \gamma + 1 & \text{if } \gamma < k-1 \\ j & \text{otherwise.} \end{cases}
\end{aligned}
$$

3.2.4 Summarizing Graph

More than one play of the interactive game might be necessary until the user accepts that none of her alternatives for the output values in the different time steps can make her win. Playing the game more often might be very time consuming. To counteract, we propose to compute a graph $\mathbb{G}$, which summarizes all plays conforming to the counterstrategy. This graph shows how the environment will react to outputs chosen by the system. It can thus be seen as a "cheat sheet" for the user in the interactive game. The user might discard some alternatives for output values without even trying them in the game. This reduces the number of plays necessary to understand the cause of unrealizability.

Definition of the Graph

Let $\mathcal{G} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ with $\Sigma = \mathcal{X} \times \mathcal{Y}$ be a game as defined in Section 2.4, and let $\varrho = (\Gamma, \gamma_0, \rho)$ be a counterstrategy for $\mathcal{G}$, where $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$. The graph $\mathbb{G} = (V, E, l)$ consists of a set

$V \subseteq (Q \times \Gamma)$ of vertices, a set $E \subseteq V \times V$ of directed edges, and an edge labeling function $l : E \to 2^{\Sigma}$. Every vertex $v \in V$ is a tuple $(q, \gamma) \in (Q \times \Gamma)$, i.e., a situation that might occur during a play. We define the set V of vertices inductively:

- (q_0, γ_0) is element of V
- (q', γ') is element of V if:

$$\exists (q, \gamma) \in V \,.\, \exists (x, y) \in \Sigma \,.\, \delta(q, (x, y)) = q' \wedge (q, \gamma, x, \gamma') \in \rho$$

The vertex $v_0 = (q_0, \gamma_0)$ is distinguished as start vertex. The set of edges is defined as

$$E = \{((q, \gamma), (q', \gamma')) \in V \times V \mid \exists (x, y) \in \Sigma \,.\, \delta(q, (x, y)) = q' \wedge (q, \gamma, x, \gamma') \in \rho\}.$$

Edges $(v, v') \in E$ are labeled with all letters $(x, y) \in \Sigma$ for which a transition from v to v' is possible. We therefore define the labeling function to be

$$l((q, \gamma), (q', \gamma')) = \{(x, y) \in \Sigma \mid \delta(q, (x, y)) = q' \wedge (q, \gamma, x, \gamma') \in \rho\}.$$

In presence of a countertrace $\overline{\tau}$, the according counterstrategy $\varrho^{\overline{\tau}}$ as defined in Section 3.2.3 is used in the definition of the graph $\mathbb{G}$.

Computation of the Graph

A symbolic algorithm computing $\mathbb{G}$ is possible but not useful, as the graph needs to be represented in an explicit manner in the end. After all, it must be presented to the user. Thus, the computation is done with a simple depth first search for all situations $(q, \gamma) \in V$, starting with (q_0, γ_0). Edges and their labels are computed simultaneously with new vertices during this search. To overcome performance problems, the computation of the graph is aborted if it exceeds a certain number of vertices. It is intractable for the user to analyze huge graphs anyway.

For the visualization of the graph, we recommend the graph drawing tool DOT[1] [62]. This tool takes a textual description of the graph as input and produces a graphical representation thereof as output. It attempts to avoid edge crossings, it tries to keeps edges short, and it provides a rich set of formatting options. These arguments, together with the simplicity of the input language, made us decide for DOT.

For games with a large state space, the graph $\mathbb{G}$ can become quite large as well. For every situation (q_i, γ_i) that might occur, it contains edges to successor situations (q_{i+1}, γ_{i+1}) for all input letters that conform to the counterstrategy and for all output letters that can then be chosen by the system. However, having more than one input letter for any situation (q_i, γ_i) is not necessary. One counterstrategy-conforming input letter per situation suffices for the tool to win every play. The size of the graph can therefore be reduced by ensuring that the counterstrategy is deterministic, meaning that

$$\forall q \in Q, \gamma \in \Gamma \,.\, \left|\{(q, \gamma, x, \gamma') \in \rho\}\right| \leq 1.$$

As already mentioned in the previous section, determinization can be done implicitly during the graph computation: Given a certain situation (q_i, γ_i), the tool selects one tuple $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ in some deterministic way and behaves as though there would not be any other tuple $(q_i, \gamma_i, x'_i, \gamma'_{i+1}) \in \rho$. Using a deterministic counterstrategy for the graph computation potentially reduces not only the number of edges but also the number of vertices. If some input letters are not used any more in a particular situation, some successor situations might not occur any more.

[1] `http://www.graphviz.org/` (last visit in October 2009)

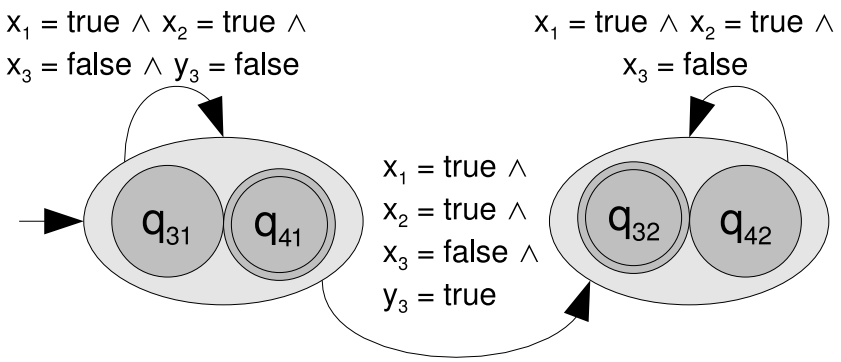


Figure 3.6: An example to illustrate the idea of the graph. The graph was created for the specification in Figure 3.4 under the assumption of a memoryless counterstrategy that dictates $x_1 = \mathsf{true}$, $x_2 = \mathsf{true}$, and $x_3 = \mathsf{false}$ forever.

Example

We will use the minimization result for the example specification of Section 3.2.1 to illustrate the idea of the graph. This minimization result is depicted in Figure 3.4. Suppose the state space Q of the game $\mathcal{G} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ is the cross product (as defined in Section 2.4.3) of the state spaces of the automata g_3 and g_4 of Figure 3.4, i.e., $Q = \{q_{31}, q_{32}\} \times \{q_{41}, q_{42}\}$. Let us further assume for simplicity that the counterstrategy ϱ has no memory and dictates to set $x_1 = \mathsf{true}$, $x_2 = \mathsf{true}$, and $x_3 = \mathsf{false}$ forever.

Under these assumptions, the graph $\mathbb{G}$ would look as depicted in Figure 3.6. The graph summarizes all plays that are possible in the interactive game. It shows all possibilities for the user in the role of the system to choose outputs in the game. It also shows how the counterstrategy reacts to the choices of the user. The user can set $y_3 = \mathsf{false}$ forever to stay in the state (q_{31}, q_{41}). This state is, however, not accepting in the automaton representing the guarantee g_3, i.e, this behavior violates guarantee g_3. The user can also set $y_3 = \mathsf{true}$ eventually. In this case, the play will come to the state (q_{32}, q_{42}), which cannot be left any more. This state is not accepting in the automaton representing guarantee g_4, i.e., such a behavior violates g_4. So, whatever the user in the role of the system does, she cannot fulfill both guarantees g_3 and g_4. The states (q_{31}, y_{42}) and (q_{32}, y_{41}) cannot be reached when the environment adheres to the counterstrategy of setting $x_1 = \mathsf{true}$, $x_2 = \mathsf{true}$, and $x_3 = \mathsf{false}$ forever. Hence, these states are not included in the graph.

Note that the graph $\mathbb{G}$ as well as the corresponding interactive game would be much more complex if the game was constructed not from the minimized specification depicted in Figure 3.4 but from the original specification depicted in Figure 3.3. The state space of the game would be much larger and, as the system would have to choose values for outputs other than y_3 as well, the user would have much more choices with which she can influence the course of the play. Thus, she potentially would have to play more often in order to accept that none of her choices can make her win, i.e., that the specification is unrealizable.

3.3 Debugging Undesired Behavior

A formal specification is typically derived manually from some informal design intent. Mistakes in this process can result in a formal specification that does not express what the designer originally wanted to express. Such mismatches with the design intent often show up when a concrete implementation of the specification is simulated or tested. The implemented system might exhibit undesired behavior. It is clear in such a situation that the simulated system does not implement the design intent. If the incorrect

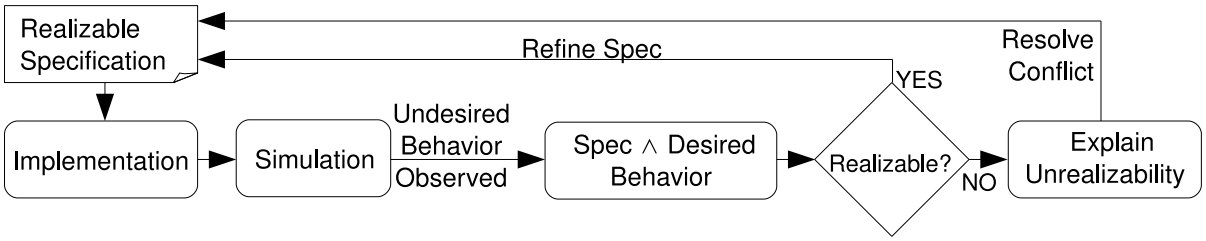


Figure 3.7: The flow of our method to handle mismatches with the design intent: When an implementation of a formal specification shows undesired behavior during simulation, its specification is augmented with a guarantee that enforces the desired behavior. If the resulting specification is realizable, the undesired behavior has been eliminated by the additional guarantee. If not, we explain unrealizability as shown in Figure 3.2.

system conforms to the specification, this further means that the specification does not represent the design intent. In such a case, it is often difficult to correct or to refine the specification so that the design intent is expressed.

The problem of having a system that does not implement the original design intent can arise with automatically synthesized systems as well as with manual implementations. In case of automatic synthesis, the resulting system is guaranteed to conform to the specification. If the synthesized system is incorrect, then the specification must be incorrect. Manual implementations can deviate from the design intent if they are implemented from the formal specification without full knowledge about the informal design intent. Conformance with the formal specification can be ensured by model checking.

3.3.1 Our Debugging Procedure

Figure 3.7 illustrates our approach for handling mismatches between a formal specification and the design intent. Suppose that some undesired behavior was observed while simulating an implementation of the specification. Then, the following two cases can be distinguished:

1. There exists a system (i.e., a Mealy machine as defined in Section 2.3.2) that fulfills the specification and, at the same time, shows the desired behavior. This means that the specification leaves enough freedom to choose either the observed or the desired behavior. The specification is incomplete.
2. Any system exhibiting the desired behavior violates the specification. This means that there exists no implementation of the specification that shows the desired behavior. The specification is in conflict with the desired behavior.

In order to find out which of the two cases applies, an additional guarantee g_d enforcing the desired behavior is added to the specification $\varphi = A \rightarrow G$ to obtain $\varphi' = A \rightarrow (G \cup \{g_d\})$. If φ' is realizable, then the first case applies. The specification φ is incomplete and needs to be refined. The augmented specification φ' is a refinement of φ which eliminates the undesired behavior. If φ' is unrealizable, the second case applies. The specification is so restrictive that it forbids the desired behavior. We need to explain why enforcing the desired behavior makes the specification unrealizable, i.e., why φ' is unrealizable. This is done as illustrated in Figure 3.2 and explained in Section 3.2. The minimization step removes parts of the specification which are not in conflict with the design intent. The conflict is then illustrated by a counterstrategy or a countertrace with a graph and in form of an interactive game. Once the user has understood the conflict, she can resolve it. The process of resolving the conflict cannot be automated since there are typically various different fixes. One can add environment assumptions, remove or weaken certain guarantees, etc. Only the user can decide which of these solutions is best suitable, because only the user knows how the system should finally behave.

3.3.2 Formalization of the Desired Behavior

We suggest to use the simulation trace to specify the desired behavior. We suppose that this simulation trace $\overline{s} = s_0 s_1 s_2 \ldots \in \Sigma^\omega$ is represented with a finite stem $f_0 f_1 \ldots f_{a-1}$ of length a and a loop $l_0 l_1 \ldots l_{b-1}$ of length b such that

$$s_i = \begin{cases} f_i & \text{if } i < a \\ l_{(i-a) \bmod b} & \text{otherwise} \end{cases}$$

for all $i \in \mathbb{N}$. In a first step, we allow the user to change any signal value in any time step of the loop or the stem to 0, 1, or "?", where "?" stands for "don't care". The user has to modify the trace in such a way that it represents the desired behavior: Every trace that matches the input part of the desired behavior must also match the output part of the desired behavior. After the desired behavior has been specified by the user, the tool checks if the input part conforms to the environment assumptions. If not, a warning is given as the system does not have to fulfill any guarantees in such a case. If the input part conforms to the assumptions, the desired behavior is automatically turned into the guarantee g_d which enforces the desired behavior.

In the following, we formalize how g_d must look like. Let $\overline{d} = d_0 d_1 d_2 \ldots$ be the desired behavior as specified by the user. For every time step i, this desired behavior can be seen as a function $d_i \colon (X \cup Y) \to \{0, 1, ?\}$. We will refer to the input part of the desired behavior as $\overline{d}^x = d_0^x d_1^x d_2^x \ldots$, where $d_i^x \colon X \to \{0, 1, ?\}$ for all time steps i. The output part will be written as $\overline{d}^y = d_0^y d_1^y d_2^y \ldots$ with $d_i^y \colon Y \to \{0, 1, ?\}$, respectively. Let $\sigma \sqsubseteq d_i^x$ denote that the letter $\sigma = (x, y) \in \Sigma$ conforms to the function d_i^x in the sense that

$$(x, y) \sqsubseteq d_i^x \Leftrightarrow \forall v \in X \,.\, (d_i^x(v) = 0 \Rightarrow v \notin x) \wedge (d_i^x(v) = 1 \Rightarrow v \in x) \,.$$

Analogously, let $\sigma \sqsubseteq d_i^y$ denote that the letter $\sigma = (x, y) \in \Sigma$ conforms to the function d_i^y, defined as

$$(x, y) \sqsubseteq d_i^y \Leftrightarrow \,.\, \forall v \in Y \,.\, (d_i^y(v) = 0 \Rightarrow v \notin y) \wedge (d_i^y(v) = 1 \Rightarrow v \in y) \,.$$

We extend the semantics of the operator $\sqsubseteq$ to traces in the natural way. We write $\overline{\sigma} \sqsubseteq \overline{d}^x$ to state that a trace $\overline{\sigma} = (x_0, y_0)(x_1, y_1)(x_2, y_2) \ldots \in \Sigma^\omega$ matches the input part $\overline{d}^x$ of the desired behavior. We define

$$\overline{\sigma} \sqsubseteq \overline{d}^x \Leftrightarrow \forall i \in \mathbb{N} \,.\, \sigma_i \sqsubseteq d_i^x \,.$$

Analogously, $\overline{\sigma} \sqsubseteq \overline{d}^y$ means that the trace $\overline{\sigma}$ matches the output part $\overline{d}^y$ of the desired behavior in the sense that

$$\overline{\sigma} \sqsubseteq \overline{d}^y \Leftrightarrow \forall i \in \mathbb{N} \,.\, \sigma_i \sqsubseteq d_i^y \,.$$

The guarantee g_d which enforces the desired behavior must be constructed such that

$$g_d \text{ accepts a trace } \overline{\sigma} \Leftrightarrow \left(\overline{\sigma} \sqsubseteq \overline{d}^x \Rightarrow \overline{\sigma} \sqsubseteq \overline{d}^y \right) .$$

How the guarantee g_d can be built depends on the actual specification language. Section 4.5.2 explains the construction for GR(1) specifications. This construction can also be used for other kinds of specifications if the guarantees are represented by DBWs.

3.3.3 Example

For an example we refer the reader to Section 6.2.2, in which an industrial-size specification is debugged with our approach. A simpler example can be found in Section 5.2.

3.3.4 Application to Specification Development

When systems can be synthesized automatically from the specification, our debugging procedure can also be used for specification development. The user simply starts with an empty specification and simulates a synthesized system with some input scenario. She modifies the trace to obtain the desired behavior for that scenario. The desired behavior is turned into an additional guarantee automatically. Next, the user simulates an implementation of this refined specification with another input scenario, and so on. This is done until all scenarios are covered and all conflicts are resolved. When no conflicts arise, the user does not have to know anything about the underlying specification language.

4 Debugging GR(1) Specifications

This chapter explains how the debugging approach presented in Chapter 3 can be applied to GR(1) specifications. All prerequisites are met:

1. A GR(1) specification defines the allowed behavior of a reactive system in a temporal manner.
2. The specification is composed of environment assumptions and system guarantees.
3. Guarantees can be added to the specification and removed from it.
4. Output variables can be existentially quantified.
5. Realizability can be decided as explained by Piterman et al. [80].
6. Counterstrategies for unrealizable GR(1) specifications can be computed. We are not aware of any work that explains how this is done. Hence, we discuss counterstrategy computation for GR(1) specifications in this chapter.
7. A GR(1) specification can be turned into a game $\mathcal{G}$. Section 2.4.3 describes this step.

The following sections concretize aspects of our debugging approach which are specific to the particular kind of specification. Note that the computation of countertraces has already been sufficiently explained in Section 3.2.2. We assume that $\varphi = A \rightarrow G$ is a GR(1) specification, where $A = \{\mathcal{A}_i^e\}$ is a set of m DBWs representing the environment assumptions, and $G = \{\mathcal{A}_j^s\}$ is a set of n DBWs representing the system guarantees. The corresponding game $\mathcal{G}^{\mathrm{GR}(1)} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$ is constructed as explained in Section 2.4.3.

4.1 Checking for Satisfiability

A lot of tools exist that are able to check an LTL specification for satisfiability. An overview and a performance comparison is given by Rozier and Vardi [87]. Such an existing tool could have been used to check GR(1) specifications for satisfiability as well, because the class of GR(1) is just a subset of LTL. However, in order to avoid the drawbacks of using an external tool, we developed an own symbolic algorithm that fits into our GR(1) setting.

4.1.1 Definition of Satisfiability

A specification is satisfiable iff there exists at least one trace $\overline{\sigma} \in \Sigma^\omega$ of signal values that fulfills the specification. A GR(1) specification is composed of system guarantees and environment assumptions. It is fulfilled by a trace if all guarantees are fulfilled by the trace, or if at least one assumption is violated by the trace. Hence, we split the problem of deciding satisfiability into two sub-problems.

Sub-Problem 1: Is there a trace that violates an assumption? This decision problem can be solved by computing the set

$$S_{\neg A} = [[\mu X \,.\, \mathsf{EX}\, X \vee T]] \text{ with } T = \left[\left[\bigvee_{i=1}^{m} T_i\right]\right] \text{ and } T_i = [[\nu Y \,.\, \neg J_i^e \wedge \mathsf{EX}\, Y]]$$

(see also the work of Emerson et al. [41]). The sets T_i contain all states $q \in Q$ such that $q \notin J_i^e$ and $\exists \sigma \in \Sigma \,.\, \delta(q, \sigma) \in T_i$. In other words, the states of T_i are not part of the set J_i^e of accepting states of the environment. Furthermore, the greatest fixpoint in Y ensures together with the operator EX that there is always a letter such that T_i is not left. So, from a state $q \in T_i$, a trace exists such that the set J_i^e is never visited. The set T is the union of all sets T_i. Hence, from all states $q \in T$, there exists a trace such that

J_i^e is never visited for some i. The set $S_{\neg A}$ contains all states from which a state of T can be reached in a finite number of steps. Hence, from all states $q \in S_{\neg A}$, there exists a trace such that J_i^e is visited only finitely often for some i. This means that there exists a trace that violates some assumption when starting from a state of $S_{\neg A}$.

Lemma 9. *A trace $\overline{\sigma} \in \Sigma^\omega$ that violates some assumption of a GR(1) specification φ exists iff $q_0 \in S_{\neg A}$.*

Sub-Problem 2: Is there a trace that fulfills all guarantees? In order to solve this decision problem, the set

$$S_G = [[\mu X \,.\, \mathsf{EX}\, X \vee U]] \text{ with } U = \left[\left[\nu Y \,.\, \bigwedge_{j=1}^{n} \mathsf{EX}\left(\mu Z \,.\, Y \wedge (J_j^s \vee \mathsf{EX}(Z))\right)\right]\right]$$

is computed (see also the work of Emerson et al. [41]). The set U contains all states $q \in Q$ from which all sets J_j^s can be reached while never leaving U. The least fixpoint in Z ensures together with the conjunction over all j that all sets J_j^s can be reached from any state $q \in U$. The greatest fixpoint in Y and the operator EX ensure that every state in U has a successor in U, i.e., that U does not need to be left. Hence, from all states of U, there exists a trace such that all sets J_j^s of accepting states of the system are visited infinitely often. However, the states in U are not the only ones from which this is possible. Such a trace can also be found when starting in states from which a state in U can be reached in a finite number of steps. These states are added in the computation of S_G. The set S_G finally contains all states from which a trace exists that fulfills all guarantees.

Lemma 10. *A trace $\overline{\sigma} \in \Sigma^\omega$ that fulfills all guarantees of a GR(1) specification φ exists iff $q_0 \in S_G$.*

The following theorem finally states how the satisfiability problem can be decided for GR(1) specifications.

Theorem 11. *A GR(1) specification φ is satisfiable iff $q_0 \in (S_{\neg A} \cup S_G)$.*

Proof. Theorem 11 follows immediately from Lemma 9 and Lemma 10. □

4.1.2 Symbolic Algorithm

Listing 4.1 depicts a symbolic algorithm that checks if a GR(1) specification is satisfiable. It is written in Python-like pseudo code. The operator `|` implements the μ-calculus operator $\vee$, i.e., the union of two sets. The operator `&` implements the μ-calculus operator $\wedge$, and `!` implements $\neg$. The expression `{}` denotes the empty set of states.

4.2 Minimization

The minimization approach proposed in Section 3.2.1 requires that a decision procedure for realizability is available. Piterman et al. [80] explain how such a check for realizability can be performed. The winning region

$$W_{\text{sys}}^{\text{GR(1)}} = \left[\left[\nu Z \,.\, \bigwedge_{j=1}^{n} \mu Y \,.\, \bigvee_{i=1}^{m} \nu X \,.\, J_j^s \wedge \mathsf{MX}^{\text{s}}\, Z \vee \mathsf{MX}^{\text{s}}\, Y \vee \neg J_i^e \wedge \mathsf{MX}^{\text{s}}\, X\right]\right]$$

of the system is computed (see also Section 2.6). The specification is realizable iff $q_0 \in W_{\text{sys}}^{\text{GR(1)}}$. We use this procedure with the following performance improvement. For every iterate Z_a of Z (according

```
is_sat():
    T = {}
    for i in range(1, m):
        Y = Q
        while Y changes:
            Y = (!Je[i]) & EX(Y)
        T = T | Y
    X = {}
    while X changes:
        X = T | EX(X)
    if q_0 in X:
        return True
    Y = Q
    while Y changes:
        tmp = Q
        for j in range(1, n):
            Z = {}
            while Z changes:
                Z = Y & (Js[j] | EX(Z))
            tmp = tmp & EX(Z)
        Y = tmp
    X = {}
    while X changes:
        X = Y | EX(X)
    if q_0 in X:
        return True
    return False
```

Listing 4.1: A symbolic algorithm that checks if a GR(1) specification is satisfiable. The algorithm is written in Python-like pseudo code. The operator `|` implements the μ-calculus operator $\vee$, `&` implements $\wedge$, `!` implements $\neg$, and $\{\}$ denotes the empty set of states.

to Equation 2.2), we check if $q_0 \in Z_a$. If $q_0 \notin Z_a$ for some a, we abort the computation, signaling that the specification is unrealizable. This is possible, because the iterates in a greatest fixpoint computation are monotonically decreasing, so

$$(\exists a \in \mathbb{N} \,.\, q_0 \notin Z_a) \Rightarrow q_0 \notin W_{\text{sys}}^{\text{GR(1)}}.$$

The minimization method introduced in Section 3.2.1 furthermore requires that guarantees and output signals can be removed from the specification. In our GR(1) setting, the set G of guarantees is a set of DBWs, so removing guarantees reduces to removing DBWs from this set. Output signals can be removed by projecting them from the DBWs, i.e., by existentially quantifying them in the transition relations of the DBWs. If the DBWs are encoded symbolically with BDDs, the existential quantification operation provided by the BDD library can be used. In fact, the automata may be non-deterministic after quantification, so the term DBW is no longer appropriate. However, our symbolic algorithms can handle this non-determinism without difficulty.

4.3 Counterstrategies

This section explains how a counterstrategy for an unrealizable GR(1) specification can be computed. As discussed in Section 2.4.3, a GR(1) specification φ can be transformed into a game $\mathcal{G}^{\text{GR(1)}} = (Q, \Sigma, \delta, q_0, \text{Win})$. A counterstrategy is a winning strategy for the environment in this game. Piterman et al. [80] show how to compute a winning strategy for the system in this game (see Section 2.6).

They first compute the winning region for the system. In a second step, they derive the strategy from some intermediate results obtained during the computation of the winning region. We follow their approach. We first compute a winning region for the environment. Intermediate results are then used to derive a counterstrategy.

4.3.1 Computation of the Winning Region

The winning region for the system in the game $\mathcal{G}^{\mathrm{GR}(1)}$ is defined as [80]

$$W_{\mathrm{sys}}^{\mathrm{GR}(1)} = \left[\left[\nu Z \,.\, \bigwedge_{j=1}^{n} \mu Y \,.\, \bigvee_{i=1}^{m} \nu X \,.\, J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y \vee \neg J_i^e \wedge \mathsf{MX}^{\mathrm{s}}\, X \right]\right].$$

The winning region for the environment is the complement of the winning region for the system. With the equalities defined in Equation 2.5 and Equation 2.6, this complement can be written as

$$\begin{aligned}
W_{\mathrm{env}}^{\mathrm{GR}(1)} &= Q \setminus W_{\mathrm{sys}}^{\mathrm{GR}(1)} \\
&= \left[\left[\neg\nu Z \,.\, \bigwedge_{j=1}^{n} \mu Y \,.\, \bigvee_{i=1}^{m} \nu X \,.\, J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, Z \vee \mathsf{MX}^{\mathrm{s}}\, Y \vee \neg J_i^e \wedge \mathsf{MX}^{\mathrm{s}}\, X \right]\right] \\
&= \left[\left[\mu Z \,.\, \bigvee_{j=1}^{n} \nu Y \,.\, \bigwedge_{i=1}^{m} \mu X \,.\, \neg \left(J_j^s \wedge \mathsf{MX}^{\mathrm{s}}\, \neg Z \vee \mathsf{MX}^{\mathrm{s}}\, \neg Y \vee \neg J_i^e \wedge \mathsf{MX}^{\mathrm{s}}\, \neg X \right) \right]\right] \\
&= \left[\left[\mu Z \,.\, \bigvee_{j=1}^{n} \nu Y \,.\, \bigwedge_{i=1}^{m} \mu X \,.(\neg J_j^s \vee \neg\, \mathsf{MX}^{\mathrm{s}}\, \neg Z) \wedge \neg\, \mathsf{MX}^{\mathrm{s}}\, \neg Y \wedge (J_i^e \vee \neg\, \mathsf{MX}^{\mathrm{s}}\, \neg X) \right]\right].
\end{aligned}$$

The duality stated in Equation 2.4 can now be applied to obtain

$$W_{\mathrm{env}}^{\mathrm{GR}(1)} = \left[\left[\mu Z \,.\, \bigvee_{j=1}^{n} \nu Y \,.\, \bigwedge_{i=1}^{m} \mu X \,.(\neg J_j^s \vee \mathsf{MX}^{\mathrm{e}}\, Z) \wedge \mathsf{MX}^{\mathrm{e}}\, Y \wedge (J_i^e \vee \mathsf{MX}^{\mathrm{e}}\, X) \right]\right]. \tag{4.1}$$

Theorem 12. *The set $W_{\mathrm{env}}^{\mathrm{GR}(1)}$ is the winning region for the environment in the game $\mathcal{G}^{\mathrm{GR}(1)}$.*

Intermediate Results for the Counterstrategy Computation

The computation of a counterstrategy relies on some intermediate results of the nested fixpoint computation defined by Equation 4.1. The required intermediate results are the sets Z_a, $Y_{a,j}$, and $X_{a,j,i,c}$ for all values of $a \in \{0, 1, \ldots, A\}$, $j \in \{1, 2, \ldots, n\}$, $i \in \{1, 2, \ldots, m\}$, and $c \in \{0, 1, \ldots, C_{a,j,i}\}$. All these sets are subsets of Q. They are defined as following.

The set Z_a is the a-th iterate of the outermost fixpoint in Equation 4.1. The iterates are defined according to Equation 2.1. The maximum value of a is the smallest integer A such that $Z_A = Z_{A-1}$. The sets $Y_{a,j}$ are defined as

$$Y_{a,j} = \left[\left[\nu Y \,.\, \bigwedge_{i=1}^{m} \mu X \,.\, \left(\neg J_j^s \vee \mathsf{MX}^{\mathrm{e}}\, Z_{a-1}\right) \wedge \mathsf{MX}^{\mathrm{e}}\, Y \wedge (J_i^e \vee \mathsf{MX}^{\mathrm{e}}\, X) \right]\right] \tag{4.2}$$

for all values of a and j. The set $X_{a,j,i,c}$ is defined to be the c-th iterate (again according to Equation 2.1) of the fixpoint computation

$$X_{a,j,i} = \left[\left[\mu X \,.\, \left(\neg J_j^s \vee \mathsf{MX}^{\mathrm{e}}\, Z_{a-1}\right) \wedge \mathsf{MX}^{\mathrm{e}}\, Y_{a-1,j} \wedge (J_i^e \vee \mathsf{MX}\, X) \right]\right].$$

The maximum value of c for some a, j, and i, is $C_{a,j,i}$. It is defined to be the smallest integer b such that $X_{a,j,i,b} = X_{a,j,i,b-1}$. We have that $X_{a,j,i,c} \supseteq X_{a,j,i,c-1}$ and $Z_a \supseteq Z_{a-1}$ for all values of a, j, i, and c in their respective domains. Hence, we define

$$\begin{aligned} Z_a^{\text{new}} &= Z_a \setminus Z_{a-1} \quad \text{and} \\ X_{a,j,i,c}^{\text{new}} &= X_{a,j,i,c} \setminus X_{a,j,i,c-1} \end{aligned}$$

to denote the part of Z_a or $X_{a,j,i,c}$ which was added in the last iteration of the least fixpoint computation.

Symbolic Algorithm

Listing 4.2 shows how the μ-calculus formula defining $W_{\text{env}}^{\text{GR(1)}}$ can be turned into a symbolic algorithm. It is written in the same Python-like pseudo code as Listing 4.1. The variable `Z` contains the resulting set $W_{\text{env}}^{\text{GR(1)}}$ at the end of the algorithm. The intermediate results Z_a, $Y_{a,j}$, and $X_{a,j,i,c}$ are computed in the arrays `z_array[a]`, `y_array[a][j]`, and `x_array[a][j][i][c]`.

```
W_env():
    Z = {}
    z_array[0] = Z
    a = 1
    while Z changes:
        unionY = {}
        for j in range(1, n):
            Y = Q
            while Y changes:
                interX = Q
                for i in range(1, m):
                    X = {}
                    x_array[a][j][i][0] = X
                    c = 1
                    while X changes:
                        X = ((!Js[j]) | MXe(Z)) &
                            MXe(Y) &
                            (Je[i] | MXe(X))
                        x_array[a][j][i][c++] = X
                    interX = interX & X
                Y = interX
            y_array[a][j] = Y
            unionY = unionY | Y
        Z = unionY
        z_array[a++] = Z
return (Z, z_array, y_array, x_array)
```

Listing 4.2: A symbolic algorithm to compute the winning region $W_{\text{env}}^{\text{GR(1)}}$ for the environment in the game $\mathcal{G}^{\text{GR(1)}}$. The intermediate results required for the computation of a counterstrategy are computed as well. The algorithm is written in the same Python-like pseudo code syntax as Listing 4.1.

The algorithm in Listing 4.2 can be optimized in the following way. When the initial state q_0 is an element of some iterate Z_a of Z, the computation can be aborted. As explained later, the counterstrategy ensures that once the play is in Z_a, it can only move to lower iterates of Z. Therefore, if $q_0 \in Z_a$, a counterstrategy from higher iterates of Z is not necessary. The intermediate results for higher iterates of Z are also not necessary to compute a counterstrategy that is winning for the environment from a state of Z_a.

Discussion

The following observations are important in order to understand the construction of a counterstrategy from the intermediate results that have been obtained.

From all states of $Y_{1,j}$, an infinite play can be enforced by the environment such that the set J_j^s of accepting states of the system is never visited while all sets J_i^e of accepting states of the environment are visited infinitely often. Equation 4.2 ensures that as following. The term $(J_i^e \vee \mathsf{MX}^{\mathrm{e}}\, X)$ together with the least fixpoint in X enforces that all sets J_i^e can be visited from every state of $Y_{1,j}$. From a state $q \in X_{1,j,i,c}$, the set J_i^e can be reached in at most $c-1$ steps. The conjunction with $\mathsf{MX}^{\mathrm{e}}\, Y$ together with the greatest fixpoint in Y guarantees that $Y_{1,j}$ is not left. The conjunction with the term $(\neg J_j^s \vee \mathsf{MX}^{\mathrm{e}}\, Z_{a-1})$ in Equation 4.2 makes sure that the set J_j^s is never visited, because we consider the case where $a = 1$, and the iterate Z_0 is the empty set $\emptyset$.

The set Z_1 is the union of all sets $Y_{1,j}$. Thus, from all states of Z_1, the environment can ensure that some set J_j^s of accepting states of the system is never visited while all sets J_i^e of accepting states of the system are visited infinitely often. This is a sufficient but not a necessary condition for the environment to win. The environment wins already if some set J_j^s is visited only finitely often (and all sets J_i^e are visited infinitely often). This circumstance is covered by higher iterates of Z.

When the play is in some state $q \in Z_a^{\mathrm{new}}$ for $a > 1$, there are two possibilities. First, the play might stay in Z_a^{new} forever. Second, the play might move to Z_{a-1} eventually. The system can only ensure that the play stays in Z_a^{new}, if some set J_j^s is never visited. If all sets J_j^s are visited, it is possible for the environment to take the play into a smaller iterate of Z. This is ensured by the term $(\neg J_j^s \vee \mathsf{MX}^{\mathrm{e}}\, Z)$ of Equation 4.1. If the play moves to the next smaller iterate of Z, we have the same situation: Either the play never visits some set J_j^s, or the environment is able to force the play into the next smaller iterate of Z. If the system forces the play to stay in some set Z_a^{new} forever, the play can of course be won by the environment, as the environment can enforce that some J_j^s is never visited in Z_a^{new}. If the play does not stay in any set Z_a^{new} forever, it will eventually reach Z_1, because the number of iterates of Z is finite and the play is infinite. From the states of Z_1, the environment is able to win as already discussed.

4.3.2 Computation of the Counterstrategy

We now define a counterstrategy $\varrho^{\mathrm{GR}(1)} = (\Gamma, \gamma_0, \rho)$ for the game $\mathcal{G}^{\mathrm{GR}(1)} = (Q, \Sigma, \delta, q_0, \mathrm{Win})$. The counterstrategy must be winning for the environment. That is, it has to ensure that all sets J_i^e of accepting states of the environment are visited infinitely often while at least one set J_j^s of accepting states of the system is visited only finitely often.

The Memory of the Counterstrategy

We define the finite memory Γ of the counterstrategy as $\Gamma = \mathcal{I} \times \mathcal{J}$. The set $\mathcal{I} = \{1, \ldots, m\}$ stores the index i of the set J_i^e of accepting states of the environment that will be reached next. The set $\mathcal{J} = \{0, 1, \ldots, n\}$ stores the index j of the set J_j^s of accepting states of the system, which the environment tries to evade. The value 0 means that the environment has not yet committed to any such set J_j^s. The environment has to provide inputs before the system responds with outputs. Which set J_j^s can be evaded might be unknown until the system has made its move. In such a situation, the value 0 is chosen for the index $j \in \mathcal{J}$. In the next step, j is set to a proper value depending on the outputs chosen by the system. The initial memory content is $\gamma_0 = (1, 0)$.

The Relation of the Counterstrategy

The relation $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$ of the counterstrategy can be defined using the intermediate results Z_a, $Y_{a,j}$, and $X_{a,j,i,c}$, obtained during the computation of the winning region for the environment. The

counterstrategy is composed of the sub-strategies $\rho_1, \rho_2, \rho_3, \rho_4 \subseteq (\hat{Q} \times \Gamma \times \mathcal{X} \times \Gamma)$ in the way that

$$\rho = \rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4.$$

The sub-strategies are defined below. In order to simplify notation, we define

$$\mathsf{MX}^{\mathrm{e}}_{\mathrm{x}}(P) = \{q \in Q \mid \forall y \in \mathcal{Y} \,.\, \delta(q,(x,y)) \in P\}.$$

Intuitively, $\mathsf{MX}^{\mathrm{e}}_{\mathrm{x}}(P)$ gives all states from which the environment can force the play in one step into a state of P using input x. With slight abuse of notation, we will also use the μ-calculus operator MX^{e} without the square brackets in order to keep the definitions more readable. Furthermore, we will write $i \oplus 1$ as abbreviation for $(i \mod m) + 1$.

Sub-strategy ρ_1 is used to force the play into a smaller iterate of Z whenever possible:

$$\rho_1 = \{(q,(i,j),x,(i,0)) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid \exists a \geq 2 \,.\, q \in Z_a^{\mathrm{new}} \cap \mathsf{MX}^{\mathrm{e}}_{\mathrm{x}}(Z_{a-1})\}$$

Taking the play into a smaller iterate of Z has a higher priority than all other sub-strategies. If the environment misses a chance, the system might be able to win. When ρ_1 is applied, the play moves from a state $q \in Z_a^{\mathrm{new}}$ to a state $q' \in Z_{a-1}$. The set Z_{a-1} is the union of all sets $Y_{a-1,j}$, so $q' \in Y_{a-1,j}$ for some j. In $Y_{a-1,j}$, the environment can evade J_j^s. The value of j, however, might depend on the next move of the system. The environment cannot foresee this next move, so it cannot set j to a proper value. The memory content j is thus set to 0 in order to remember to set it to an adequate value in the next step.

Sub-strategy ρ_2 is used to choose a proper value for j if it was set to 0 in the previous step:

$$\rho_2 = \{(q,(i,0),x,(i,j)) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid \exists a \geq 1 \,.\, q \in Z_a^{\mathrm{new}} \cap \mathsf{MX}^{\mathrm{e}}_{\mathrm{x}}(Y_{a,j}) \setminus \mathsf{MX}^{\mathrm{e}}(Z_{a-1})\}$$

The strategy ρ_2 forces the play into a state of $Y_{a,j}$ and sets the value of j accordingly. Requiring that $q \notin \mathsf{MX}^{\mathrm{e}}(Z_{a-1})$ ensures that ρ_2 is only applied if ρ_1 cannot be applied.

Sub-strategy ρ_3 is applied if the the play has reached the next target set J_i^e of accepting states of the environment:

$$\rho_3 = \big\{(q,(i,j),x,(i \oplus 1,j)) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid \\ j \neq 0 \wedge q \in J_i^e \wedge \exists a \geq 1 \,.\, q \in Z_a^{\mathrm{new}} \cap \mathsf{MX}^{\mathrm{e}}_{\mathrm{x}}(Y_{a,j}) \setminus \mathsf{MX}^{\mathrm{e}}(Z_{a-1})\big\}$$

The next goal is to reach the set $J^e_{i\oplus 1}$, so the value of i is updated accordingly. The value of j remains the same, since the same J_j^s as before should be evaded. The play is forced into a state of $Y_{a,j}$ so that J_j^s can be evaded in the next step as well. Requiring that $q \notin \mathsf{MX}^{\mathrm{e}}(Z_{a-1})$ again ensures that ρ_3 is only applied if ρ_1 cannot be applied. Furthermore, requiring $j \neq 0$ ensures that ρ_2 has priority over ρ_3.

Sub-strategy ρ_4 is an attractor strategy forcing the play ever closer to the next target set J_i^e:

$$\rho_4 = \big\{(q,(i,j),x,(i,j)) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \;\big|\; \\ j \neq 0 \wedge \exists a \geq 1, c \geq 2 \,.\, q \in Z_a^{\mathrm{new}} \cap X^{\mathrm{new}}_{a,j,i,c} \cap \mathsf{MX}^{\mathrm{e}}_{\mathrm{x}}(X_{a,j,i,c-1}) \setminus \mathsf{MX}^{\mathrm{e}}(Z_{a-1})\big\}$$

The strategy ρ_4 is applied if the next target set J_i^e is not yet reached. The value of i remains unchanged as the environment is still heading for the same target. The value of j remains unchanged as well, because the same set J_j^s as before must be evaded. From a state $q \in X^{\mathrm{new}}_{a,j,i,c}$, the environment can force a play into a state of J_i^e in at most $c-1$ steps. The strategy forces the play into a state of $X_{a,j,i,c-1}$, from which the set J_i^e can be reached in at most $c-2$ steps. The set $X_{a,j,i,1}$ is reached eventually. All states of $X_{a,j,i,1}$ are also in J_i^e, so the target is reached eventually. The conditions $j \neq 0$ and $q \notin \mathsf{MX}^{\mathrm{e}}(Z_{a-1})$ ensure that ρ_4 has lower priority than ρ_1 and ρ_2.

Theorem 13. *The strategy $\varrho^{\mathrm{GR}(1)} = (\mathcal{I} \times \mathcal{J}, (1,0), \rho)$, where $\rho = \rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4$, is winning for the environment in the GR(1) game $\mathcal{G}^{\mathrm{GR}(1)}$.*

Symbolic Algorithm

Listing 4.3 shows how the relation ρ of the counterstrategy $\varrho^{\mathrm{GR}(1)}$ can be computed symbolically. It is written in the same Python-like pseudo code syntax as Listing 4.1. It uses the intermediate results `z_array[a]`, `y_array[a][j]`, and `x_array[a][j][i][c]`, obtained from the computation in Listing 4.2. The utilized functions are defined as

$$
\begin{aligned}
\texttt{SymPi(n)} &= \{(q,(i,j),x,(i',j')) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid i = n\}\,,\\
\texttt{SymNi(n)} &= \{(q,(i,j),x,(i',j')) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid i' = n\}\,,\\
\texttt{SymPj(n)} &= \{(q,(i,j),x,(i',j')) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid j = n\}\,,\\
\texttt{SymNj(n)} &= \{(q,(i,j),x,(i',j')) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid j' = n\}\,,\\
\texttt{MXex(P)} &= \{(q,(i,j),x,(i',j')) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid \forall y \in \mathcal{Y}\,.\,\delta(q,(x,y)) \in P\}\text{ , and}\\
\texttt{MXe(P)} &= \{(q,(i,j),x,(i',j')) \in (Q \times \Gamma \times \mathcal{X} \times \Gamma) \mid \exists x \in \mathcal{X}\,.\,\forall y \in \mathcal{Y}\,.\,\delta(q,(x,y)) \in P\}.
\end{aligned}
$$

```
1  rho1 = {}
2  for a in range(2, length(z_array) - 1):
3      for i in range(1, m):
4          tmp = SymPi(i) & SymNi(i) & SymNj(0)
5                & z_array[a] & (!z_array[a-1]) & MXex(z_array[a-1])
6          rho1 = rho1 | tmp
7  rho2 = {}
8  for a in range(1, length(z_array) - 1):
9      for j in range(1, n):
10         for i in range(1, m):
11             tmp = SymPi(i) & SymNi(i) & SymPj(0) & SymNj(j)
12                   & z_array[a] & (!z_array[a-1])
13                   & MXex(y_array[a][j]) & (!MXe(z_array[a-1]))
14             rho2 = rho2 | tmp
15 rho3 = {}
16 for a in range(1, length(z_array) - 1):
17     for j in range(1, n):
18         for i in range(1, m):
19             tmp = SymPi(i) & SymNi((i mod m)+1) & SymPj(j) & SymNj(j)
20                   & Je[i] & z_array[a] & (!z_array[a-1])
21                   & MXex(y_array[a][j]) & (!MXe(z_array[a-1]))
22             rho3 = rho3 | tmp
23 rho4 = {}
24 for a in range (1, length(z_array) - 1):
25     for j in range(1, n):
26         for i in range(1, m):
27             for c in range(2, length(x_array[a][j][i]) - 1):
28                 tmp = SymPi(i) & SymNi(i) & SymPj(j) & SymNj(j)
29                       & x_array[a][j][i][c] & (!x_array[a][j][i][c-1])
30                       & z_array[a] & (!z_array[a-1])
31                       & MXex(x_array[a][j][i][c-1])
32                       & (!MXe(z_array[a-1]))
33                 rho4 = rho4 | tmp
34 rho = rho1 | rho2 | rho3 | rho4
```

Listing 4.3: A symbolic algorithm to compute the counterstrategy. It utilizes the intermediate results computed within Listing 4.2 and it is written in the same Python-like pseudo code syntax as Listing 4.1.

Optimization

The algorithm depicted in Listing 4.3 can be optimized in various ways. This was not done in order to keep it readable. The loops can be merged. Intermediate values that are needed in different parts of the strategy need to be computed only once. Also, the computation can be restricted to states that are reachable from the initial state q_0 [95]. The set R of reachable states is defined with the μ-calculus formula

$$R = [[\mu X \,.\, q_0 \vee \mathsf{IMG}(X)]].$$

A symbolic algorithm that computes the set R can be derived from this formula in the usual way. It is depicted in Listing 4.4.

```
1 R():
2     X = {}
3     while X changes:
4         X = q_0 | IMG(X)
5     return X
```

Listing 4.4: A symbolic algorithm to compute the set of reachable states. It is written in the same Python-like pseudo code syntax as Listing 4.1.

4.4 Interactive Game and Graph

4.4.1 Additional Information for the User

In order to help the user to understand why a certain specification is unrealizable, she can be provided with additional information during the interactive game as well as by the graph $\mathbb{G}$.

First, the current memory content of the counterstrategy should be presented in each time step. When the user knows the index j of the set J_j^s which the environment tries to evade, she can concentrate on reaching this set. That is, the user can work against the counterstrategy more focused. The user might indeed be able to bring the play into a state of the set J_j^s, but in this case, the environment can force the play into a smaller iterate of Z. The number of iterates of Z is finite, so at least one set J_j^s of accepting states of the system will be visited only a finite number of times and thus the user loses for sure.

Second, the user should be informed about the iterate Z_a, in which the play is currently in. Whenever the play enters a smaller iterate of Z, the memory content $j \in \mathcal{J}$ might change. It is first set to 0, and in the next step set to a proper value that may be different to the previous proper value. When the play is in a state of Z_a, the memory content j of the counterstrategy can change at most $a - 1$ times (not counting the changes to the special value 0) in the future of the play. Knowing the iterate Z_a, the user knows how often she will be able to reach a set of accepting states which the environment tries to evade. If the play got to Z_1, the memory content j won't change any more and the user won't be able to reach a J_j^s state from there on.

Third, the user should know to which sets J_j^s and J_i^e of accepting states the current state of the play belongs to. When analyzing a play, this helps the user to accept that she has lost. She can comprehend that all sets J_i^e have indeed been visited infinitely often and that she did not manage to visit all sets J_j^s infinitely often.

4.4.2 Combining Countertraces with Counterstrategies

When a countertrace is used instead of a counterstrategy, the memory content of the counterstrategy cannot be presented to the user. In order to have both advantages, the simplicity of the countertrace as well

as the additional information provided by the memory content of the counterstrategy, the countertrace can be used in parallel with the counterstrategy.

The entire countertrace is presented right from the beginning of the play. In every situation (q_i, γ_i), where i is the step counter, the counterstrategy is first applied. Instead of selecting an arbitrary tuple $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ from the counterstrategy and using x_i as input in step i, a tuple $(q_i, \gamma_i, \tau_i, \gamma_{i+1}) \in \rho$ is chosen, where τ_i is the input dictated by the countertrace in step i. Doing so, we obtain the inputs dictated by the countertrace and at the same time the memory contents of the counterstrategy. It is always possible to find a tuple $(q_i, \gamma_i, \tau_i, \gamma_{i+1}) \in \rho$ where the input is the very same as the input suggested by the countertrace in step i. The reason is that the countertrace is constructed in such a way that it conforms to the counterstrategy no matter how the system behaves.

4.5 Debugging Undesired Behavior

In our approach to debug undesired behavior, only the step of turning the desired behavior into a guarantee g_d depends on the specification language which is used. Our GR(1) setting assumes that guarantees are represented as DBWs, so g_d has to be incorporated by a DBW as well.

4.5.1 Recap

Remember that the desired behavior $\overline{d}$ is composed of an input part $\overline{d}^x$ and an output part $\overline{d}^y$ (see Section 3.3). Both parts are defined using a finite stem of length a and a loop of length b, so

$$\forall i \geq a \,.\, \left(d_i^x = d_{i+b}^x\right) \wedge \left(d_i^y = d_{i+b}^y\right)$$

holds. The DBW g_d must be constructed in such a way that it accepts a trace $\overline{\sigma} = \sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$ iff $(\overline{\sigma} \sqsubseteq \overline{d}^x) \Rightarrow (\overline{\sigma} \sqsubseteq \overline{d}^y)$. This means that the DBW must accept all traces that do not match the input part of the desired behavior. It must also accept all traces that match the output part of the desired behavior. It has to reject all traces that match the input part but do not match the output part of the desired behavior.

4.5.2 Definition of the DBW Representing the Desired Behavior

The DBW g_d can be defined as a tuple $g_d = (Q_d, \Sigma, \delta_d, V_0, F_d)$, where

$$\begin{aligned}
Q_d &= \{V_0, V_1, \ldots, V_{a+b-1}, V_1^{\text{✘}}, \ldots, V_{a+b-1}^{\text{✘}}, V_{\text{✔}}\},\\
F_d &= \{V_0, V_1, \ldots, V_{a+b-1}, V_{\text{✔}}\}, \text{ and}\\
\delta_d &: Q_d \times \Sigma \to Q_d \quad \text{such that}\\
\delta_d(q,\sigma) &= \begin{cases}
V_{i+1} & \text{if } 0 \leq i < a+b-1 \wedge q = V_i \wedge \sigma \sqsubseteq d_i^x \wedge \sigma \sqsubseteq d_i^y\\
V_a & \text{if } q = V_{a+b-1} \wedge \sigma \sqsubseteq d_{a+b-1}^x \wedge \sigma \sqsubseteq d_{a+b-1}^y\\
V_{i+1}^{\text{✘}} & \text{if } 0 < i < a+b-1 \wedge q = V_i^{\text{✘}} \wedge \sigma \sqsubseteq d_i^x\\
V_a^{\text{✘}} & \text{if } q = V_{a+b-1}^{\text{✘}} \wedge \sigma \sqsubseteq d_{a+b-1}^x\\
V_{i+1}^{\text{✘}} & \text{if } 0 \leq i < a+b-1 \wedge q = V_i \wedge \sigma \sqsubseteq d_i^x \wedge \sigma \not\sqsubseteq d_i^y\\
V_a^{\text{✘}} & \text{if } q = V_{a+b-1} \wedge \sigma \sqsubseteq d_{a+b-1}^x \wedge \sigma \not\sqsubseteq d_{a+b-1}^y\\
V_{\text{✔}} & \text{otherwise.}
\end{cases}
\end{aligned}$$

The general structure of the DBW g_d is illustrated in Figure 4.1. The state $V_{\text{✔}}$ is drawn twice in order to keep some edges from crossing. The edges are labeled with subsets of the alphabet Σ. These labels

define which edge is taken when a certain letter is observed. They are defined as

$$I_i = \{v \in \Sigma \mid v \not\sqsubseteq d_i^x\}, \tag{4.3}$$
$$D_i = \{v \in \Sigma \mid v \sqsubseteq d_i^x \wedge v \sqsubseteq d_i^y\}, \tag{4.4}$$
$$U_i = \{v \in \Sigma \mid v \sqsubseteq d_i^x \wedge v \not\sqsubseteq d_i^y\}, \text{ and} \tag{4.5}$$
$$L_i = U_i \cup D_i. \tag{4.6}$$

4.5.3 The Rationale Behind this Construction

When a trace matches the input part as well as the output part of the design intent, the edges labeled with D_i will be traversed. The states V_0 to V_{a-1} are visited once, and the run keeps then looping through the states V_a to V_{a+b-1}. All states V_a to V_{a+b-1} are accepting, so the run is accepting and the trace is accepted.

When a trace matches the input part but not the output part of the design intent, some edge labeled with U_i will be taken and the run comes to some of the states $V_1^{\text{✘}}$ to $V_{a+b-1}^{\text{✘}}$. As the trace matches the input part of the design intent, only edges labeled with L_i will be traversed and the run eventually starts to loop through the states $V_a^{\text{✘}}$ to $V_{a+b-1}^{\text{✘}}$. These states are not accepting, so the trace is not accepted.

When a trace does not match the input part of the design intent, some edge labeled with I_i will be traversed. The run gets trapped in $V_{\text{✔}}$, which is accepting. Hence, the trace is accepted.

The states $V_1^{\text{✘}}$ to $V_{a+b-1}^{\text{✘}}$ are necessary, because it might be the case that a trace first violates the output part of the desired behavior and at a later time step violates the input part. Such traces should be accepted as well. We can only judge whether the input part is violated in a later time step if we keep track of the position in the loop or the stem. This is exactly what the states $V_1^{\text{✘}}$ to $V_{a+b-1}^{\text{✘}}$ are for.

4.5.4 Analysis of Fundamental Properties

Claim 14. *The automaton g_d is complete and deterministic.*

Proof. For every state $q \in Q_d$ and for every letter $\sigma \in \Sigma$, there is a unique successor state $q' = \delta(q, \sigma)$. This is obvious, because the sets I_i, D_i, and U_i form a partition of Σ for all i. The sets I_i and L_i do so for all i as well. □

Theorem 15. *The DBW g_d accepts a trace $\overline{\sigma} \in \Sigma^\omega$ iff $(\overline{\sigma} \sqsubseteq \overline{d}^x) \Rightarrow (\overline{\sigma} \sqsubseteq \overline{d}^y)$ holds.*

Proof. The DBW g_d accepts all traces $\overline{\sigma}$ for which $(\overline{\sigma} \sqsubseteq \overline{d}^x) \wedge (\overline{\sigma} \sqsubseteq \overline{d}^y)$ holds by visiting the states V_0 to V_{a-1} once and looping through the states V_a to V_{a+b-1}. All traces $\overline{\sigma}$ for which $(\overline{\sigma} \not\sqsubseteq \overline{d}^x)$ holds are accepted by g_d as well, since the run gets trapped in $V_{\text{✔}}$. The DBW rejects all traces $\overline{\sigma}$ for which $(\overline{\sigma} \sqsubseteq \overline{d}^x) \wedge (\overline{\sigma} \not\sqsubseteq \overline{d}^y)$ holds. An edge labeled with U_i is taken eventually and the run finally loops through the states $V_a^{\text{✘}}$ to $V_{a+b-1}^{\text{✘}}$, which are not accepting. □

A DBW is required to be complete and deterministic. Claim 14 attributes these properties to the automaton g_d. This implies that g_d is indeed a DBW. Theorem 15 finally states that the construction is correct, i.e., that the DBW g_d enforces the desired behavior according to the definitions in Section 3.3.

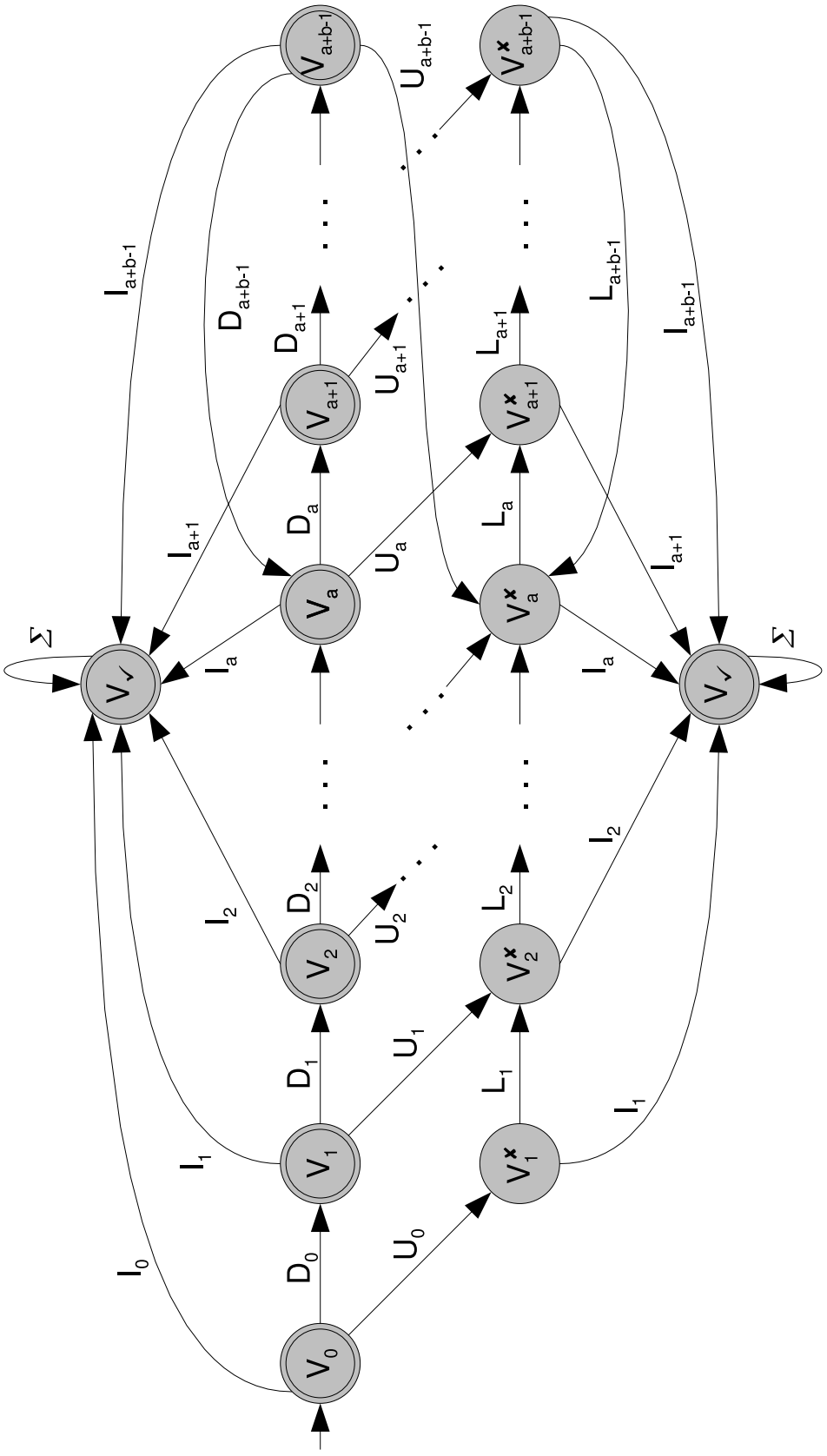


Figure 4.1: The structure of the DBW g_d representing the desired behavior $\overline{d}$. The state $V_{\checkmark}$ is drawn twice to keep edges from crossing. Edges are labeled with subsets of the alphabet Σ according to the Equations 4.3 to 4.6. Accepting states are double bordered and V_0 is the initial state.

5 Implementation

This chapter presents the prototypical implementation of our debugging approach for the class of GR(1) specifications. We have integrated our approach into the synthesis tools Anzu[1] [59] and Marduk[2]. Anzu is written in Perl, Marduk is written in Python. Both tools provide a textual user interface. Marduk is also integrated into RATSY, a successor of the requirement analysis tool RAT [79], and can be used with the graphical user interface (GUI) of RATSY. The implementation in Anzu is very similar to the one in Marduk. Hence, this chapter describes only the implementation in Marduk and its integration into RATSY.

5.1 Differences to the Theoretical Framework

In our theoretical framework, we assume that a GR(1) specification is represented by two sets of DBWs. The first set A contains DBWs incorporating the environment assumptions. The second set G contains DBWs that represent system guarantees. If the specification is given with LTL formulas as described in Section 2.2, each formula has to be turned into a DBW at first. Such a transformation is straight forward. Once the DBWs for all environment assumptions and system guarantees are available, a game can be defined as shown in Section 2.4.3.

The implementation deviates from this procedure mainly for efficiency reasons. In the implementation, the user can define system guarantees and environment assumptions either with DBWs or by the use of LTL formulas. Arbitrary combinations thereof are allowed as well. In order to achieve a better performance, the implementation does not turn every formula into a DBW before computing the game. Instead, formulas are used directly in the construction of the game. This is done as explained in Section 2.6.3. As a consequence, all safety constraints of the specification, i.e., the parts φ_t^e and φ_t^s (cf. Section 2.2), are directly incorporated by the transition relation of the game. State transitions violating safety constraints are disallowed by the transition relation.

5.2 Features available from the RATSY GUI

We extended the GUI of RATSY in order to make our specification debugging features accessible. Three new features are now provided:

- The user can play a *testing game* in order to test a specified system. In this game, the user is in the role of the environment and the tool takes on the role of the system. In every time step, the user provides values for the input signals and the tool responds with values for the output signals so that the specification is fulfilled. In order to find such values, a winning strategy for the system is synthesized and queried by the tool.
- The user can specify desired behavior of the system, starting from a trace that was obtained during a play of the testing game. The tool is able to automatically convert this desired behavior into a guarantee which enforces it.
- If a certain specification is unrealizable, the user can play a *debugging game* in order to find out why the specification is unrealizable. In the debugging game, the user is in the role of the system while the tool is in the role of the environment. In every time step, the tool first provides the values for the input signals and the user has to respond with values for the output signals so that the specification is fulfilled. The tool uses a counterstrategy to find inputs such that no behavior of the system can fulfill the specification. A satisfiability check, the computation of an unrealizable

[1] Available at `http://www.ist.tugraz.at/staff/jobstmann/anzu/` (last visit in October of 2009)
[2] Available at `http://rat.fbk.eu/ratsy` (last visit in October of 2009)

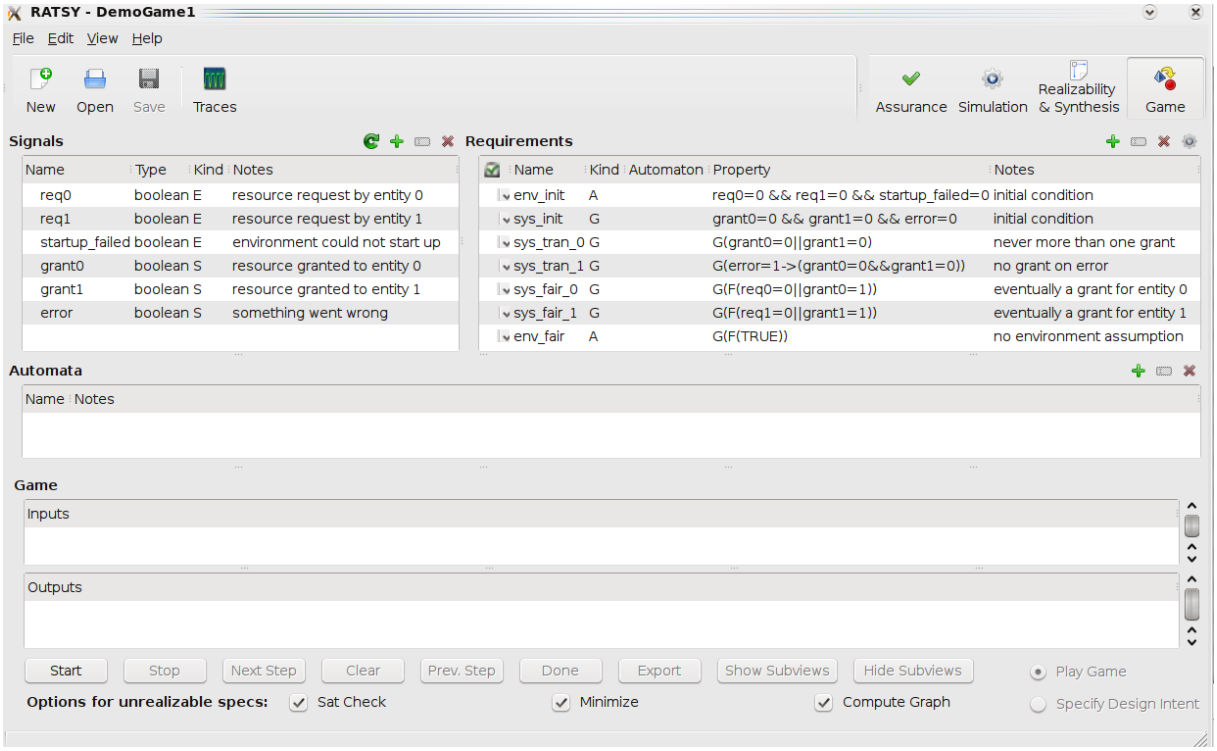


Figure 5.1: A screenshot of the game part of the RATSY GUI. It also contains the specification used for demonstrating the tool features. Signals of kind `E` are inputs, and signals of kind `S` are outputs. Assumptions are marked with `A`, and guarantees with `G`. Assumptions and guarantees are written in LTL syntax.

core, the heuristic search of a countertrace, and the computation of a summarizing graph precede the debugging game as explained in Section 3.2.

In the following, we will use an example to demonstrate all these new features of RATSY. This example will also make clear, which information is provided to the user, and how this information can be used for debugging. Figure 5.1 shows the game part of the RATSY GUI, also containing the specification used for the demonstration. This specification defines a simple arbiter for a resource shared by two entities. With the input signals `req0` and `req1`, access to the resource can be requested by Entity 0 and Entity 1, respectively. The outputs `grant0` and `grant1` signal that the resource is granted to the entities. The output `error` is raised in case of an error. The input signal `startup_failed` indicates that the environment did not start correctly. The output `error` must be set in such a case. All signals are initialized to 0 (`env_init` and `sys_init`). The guarantee `sys_tran_0` enforces that the resource is not granted to both entities at the same time. Guarantee `sys_tran_1` makes sure that no grant is given in case of an error. The guarantees `sys_fair_0` and `sys_fair_1` finally state that every request must be granted eventually. There is no assumption about the environment.

5.2.1 The Testing Game

When the user clicks onto the button `Start` (see Figure 5.1), the tool first checks if the specification is realizable. If so, it starts a testing game. Otherwise, it starts a debugging game. The specification used for demonstration is indeed realizable, so a testing game is started. Figure 5.2 shows a possible play. The current time step is marked with red letters. The user can choose values for the inputs in that time step only. Signals values can be set to 0, 1, or “don’t care”. When all inputs have the desired values in the current step, the user clicks onto `Next Step`. All inputs which still have the value “don’t care” are

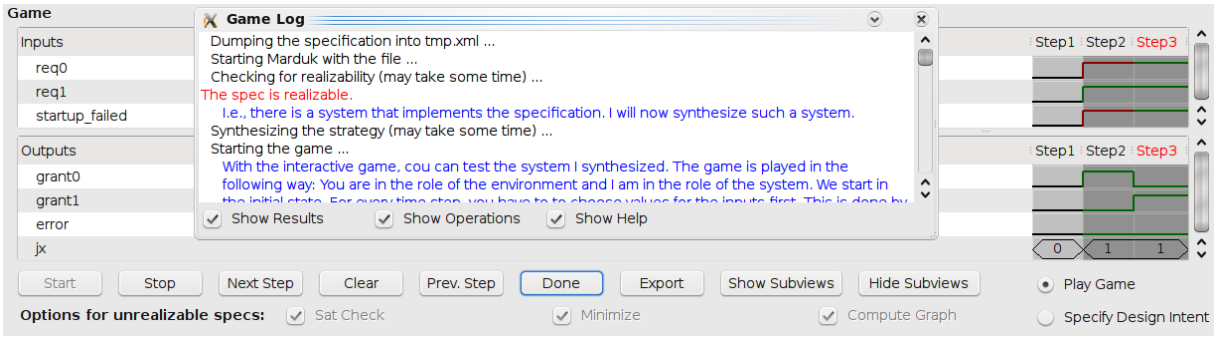


Figure 5.2: A screenshot of RATSY when playing a testing game. The current time step of the play is marked with red letters. The user can define values for the inputs in the current time step, and the tool responds with outputs. The finite part of the traces has light gray background, the infinite loop is marked with dark gray. The Game Log Window containing log messages is shown as well.

chosen arbitrarily by the tool. Then, the tool gives values for the outputs, utilizing a winning strategy for the system. After that, the next time step is started. The button `Clear` can be used to clear the user selection in the current time step. The button `Prev. Step` lets the user go back in time if she wants to change the values in some previous time step.

The tool does not let the user choose signal values that violate safety requirements of the specification. If the user attempts to do so, an error message is given. Different colors are used in the waveforms to indicate different origins of signal values:

- Black is used if the signal value is the only value fulfilling the safety requirements.
- Red is used if the value of a signal was chosen by the user.
- Blue is used if the signal value is determined by the safety requirements and some user selection for the values of other signals.
- Green is used when the signal value was chosen arbitrarily by the tool.

The different colors should remind the user of her choices and the consequences of her selections in all previous time steps. When analyzing a lost play, she can see at one glance where she might have chosen differently, and where she did not have a choice at all.

The user can specify the finite part and the infinite loop of the game trace. The finite part is marked with a light gray background, and the infinite part has a dark gray background in Figure 5.2. When the user clicks onto `Done`, an explanation is created, which argues why the tool managed to fulfill the specification. The Log Window in Figure 5.5 gives an idea of how this explanation looks like. When the user finally clicks on `Stop`, the play engine is reset, all data from the traces vanishes, and a new game can be started with a subsequent click onto `Start`.

As can be seen in Figure 5.2, the output trace contains a waveform `jx` that does not correspond to a signal of the specification. This special waveform shows the memory content of the strategy as defined by Piterman et al. [80]. That is, it contains the index j of the set J_j^s which the system tries to reach next. It can be used to gain a deeper insight into what the system is currently doing.

Exporting Game Traces

Game traces can be exported when clicking onto the button `Export`. The user can choose between three different output formats: `png`, `jpeg`, and `vcd` (Value Change Dump [57]). Traces exported into `png` or `jpeg` files are better than simple screenshots as no data is hidden due to scroll-bars. When game traces

are exported into `vcd` files, the colors in the traces, as well as the information about the position of the infinite loop, are lost. This is due to a lack of support of such elements in the definition of the Value Change Dump format. The main advantage of this format is that it is understood by most waveform viewers (e.g., by GTKWave[3] to name a freely available and powerful instance of a waveform viewer).

In the current version of the implementation, there is no way to save the state of a play. The main reason is that a lot of data would have to be stored. The strategy by itself can become huge. Storing this data together with the rest of the project information would lead to huge project files. Additionally, the use case of disrupting a debugging or testing session during a play, and continuing the play later, is not a very common one.

Subordinated Windows

There are two subordinated windows associated with the game part of RATSY. They can be shown or hidden with the buttons `Show Subviews` and `Hide Subviews`.

First, there is the Game Log Window. This window is also visible in Figure 5.2. It contains three types of log messages:

- Results are written in red. Such messages contain the main outcomes obtained by the tool during the play.
- Operations are printed in black. They show what the tool is currently doing.
- Help messages guide the user through a game. They are written in blue.

All kinds of messages can be enabled or disabled with the according check-boxes (see Figure 5.2). Disabled kinds of messages can be enabled later without any loss of information during the period of being disabled.

Second, there is the Automata Window. If the specification contains automata that were constructed with the Automaton Editor of RATSY, this window contains the current state of the play in all these automata. Otherwise, this window is not available at all. As our example specification does not contain any automata yet, this window is not yet available in the game. We will have a look at it when discussing the debugging game.

5.2.2 Specifying Desired Behavior

Suppose the user is not satisfied with the simulation trace obtained in Figure 5.2. As already mentioned in the introduction of the example, an error should be signaled when the environment could not start correctly. To be more precise, suppose that the informal design intent was that the output signal `error` has to be set in all time steps if the input `startup_failed` is set in all time steps (except for the first step, where the requirements `env_init` and `sys_init` require these signals to be 0).

The user can now switch into the *Specify Design Intent* mode by clicking on the according radio button, shown at the bottom of the right-hand side in Figure 5.2. In this mode, the user can change the value of any input and output to 0, 1 or "don't care" in order to express the design intent. There is also the possibility of editing signal values for all time steps simultaneously. The position of the infinite loop in the trace can be modified. Furthermore, new time steps can be inserted, and existing time steps can be removed. In the end, the trace should represent the desired behavior in the sense that every behavior that matches its input part must also match its output part (cf. Section 3.3.2).

Figure 5.3 shows the result of expressing the design intent. Two different colors are used for the waveforms. Black is used for signal values that were taken from the game. Red is used for signal values that were changed by the user. The user finally clicks on `Done` and a DBW is created automatically,

[3] `http://gtkwave.sourceforge.net/` (last visit in October of 2009)

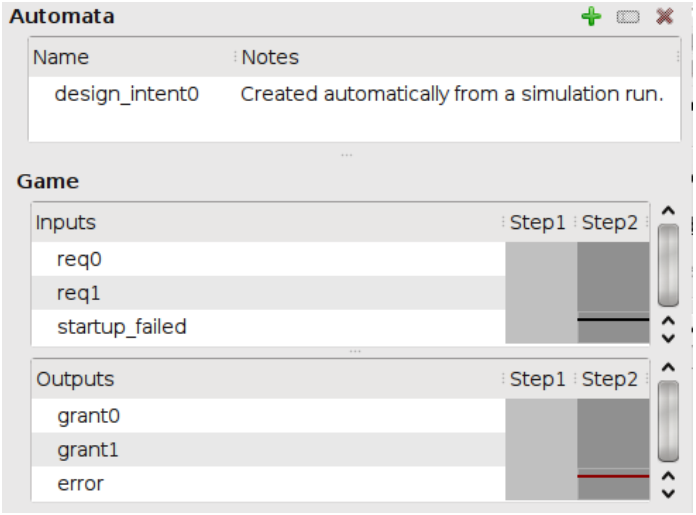


Figure 5.3: A screenshot of RATSY when specifying some design intent. Again, the finite part of the trace has light gray background, and the infinite loop is marked with dark gray. The automatically generated DBW which represents the design intent is already contained in the table of automata of the RATSY project.

which accepts only the desired behavior (`design_intent0` in Figure 5.3). This automaton is illustrated in Figure 5.4. It is constructed according to the definition of g_d in Section 4.5.2. The state `accept` corresponds to $V_{\checkmark}$, `R2` corresponds to $V_1^{\times}$, and the states `V1` and `V2` correspond to V_0 and V_1 of g_d.

The user can finally add this DBW as an additional guarantee to the specification, in order to eliminate the undesired behavior.

5.2.3 The Debugging Game

When a game is started with the enhanced specification which contains the DBW of Figure 5.4 as an additional guarantee, the tool signals that this specification is unrealizable. Following our debugging approach, the tool first checks the specification for satisfiability with the algorithm of Listing 4.1. The result of the check is printed into the Game Log Window. In our case, the specification is unrealizable but still satisfiable.

Next, the specification is minimized following the ideas presented in Section 3.2.1 and Section 4.2. All guarantees which are irrelevant for the unrealizability problem are deactivated in the table of requirements (on the top of the right-hand side in Figure 5.1) of the project. Which signals are irrelevant can be seen from the Game Log Window. The user might be able to see the conflict in the remaining guarantees already at one glance if not too many of them are left over. In our example, the guarantees `sys_tran_0` and `sys_fair_1`, as well as the output `grant1`, are irrelevant for the unrealizability problem. Hence, they will be ignored in all subsequent steps. The remaining guarantees are `sys_tran_1` (no grant on error), `sys_fair_0` (eventually a grant for Entity 0), and the behavior specified in Figure 5.3 (`error=1` if `startup_failed=1`). The problem is already obvious: When the environment sets `startup_failed=1`, then `error` has to be raised and no grant can be given. When the resource is additionally requested by Entity 0, the guarantee `sys_fair_0` cannot be fulfilled.

After the minimization step, the tool computes a counterstrategy. Thereafter, it attempts to obtain a countertrace from this counterstrategy with the heuristic presented in Section 3.2.2. This countertrace or the counterstrategy is finally used for the computation of the summarizing graph $\mathbb{G}$ as well as in the interactive debugging game. Per default, the computation of the graph $\mathbb{G}$ is aborted if $\mathbb{G}$ exceeds 100 vertices. The check for satisfiability, the minimization step, and the computation of the graph can each be disabled with the according check-boxes shown at the bottom of Figure 5.2.

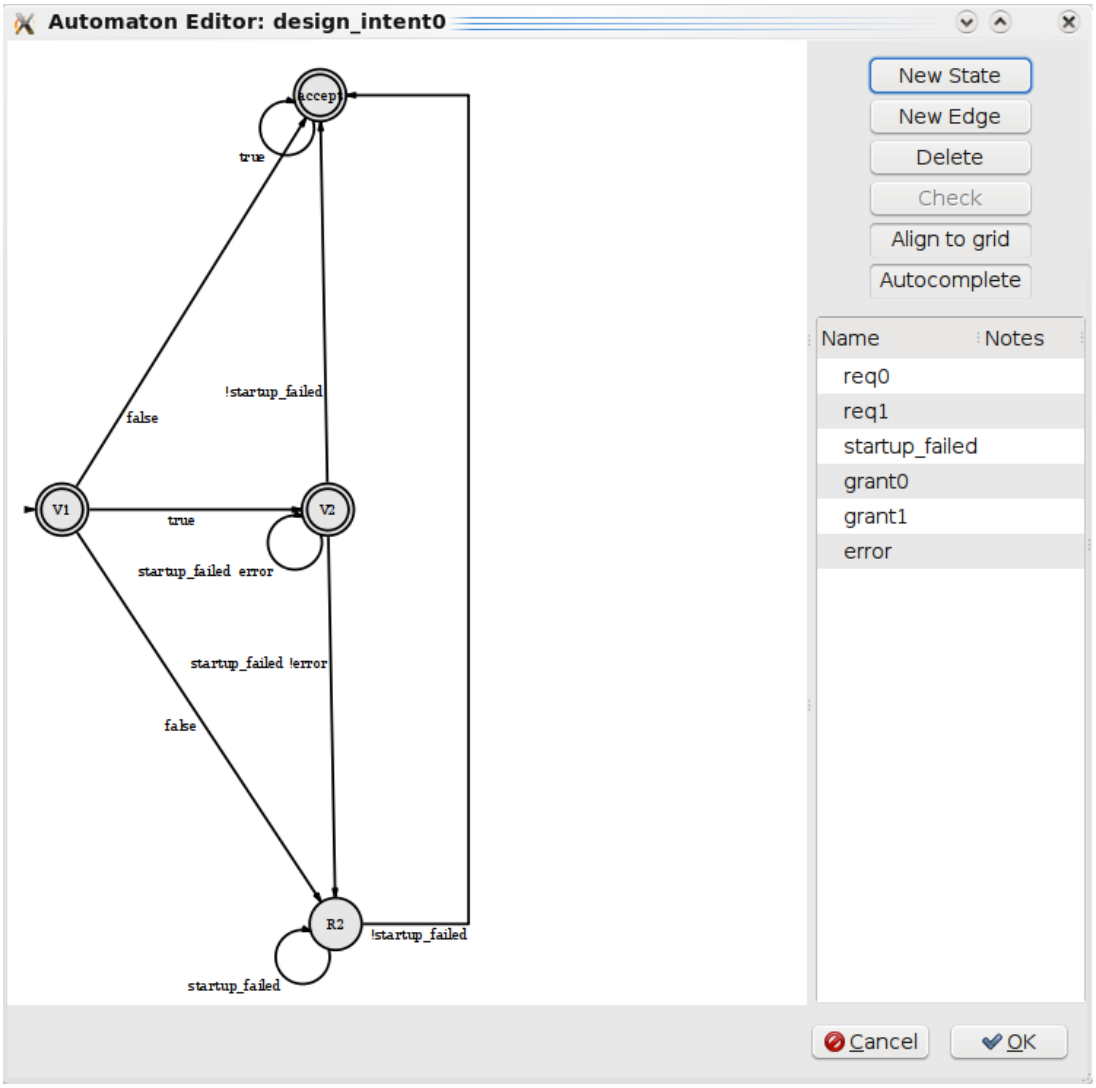


Figure 5.4: The automatically generated DBW enforcing the desired behavior. It has been created according to the definition of g_d in Section 4.5.2 for the desired behavior specified in Figure 5.3.

Figure 5.5 shows a screenshot of the debugging game. Again, the current state is marked with red letters, the finite part of the trace is marked with light gray background, and the infinite loop has a dark gray background. The meaning of the different waveform colors is the same as for the testing game as well. The only difference to the testing game is that the tool gives values for the inputs and the user has to respond with values for the outputs. For our example, the tool is able to find a countertrace. It is shown in the input part of the trace in Figure 5.5. It is presented right from the beginning of the play, so that the user knows already in advance how the environment will behave. The countertrace exploits the already described problem by setting `startup_failed=1` and `req0=1` forever.

When the user finally clicks onto `Done`, the tool explains why the user did not manage to fulfill the specification. For our example, the explanation is contained in the Game Log Window of Figure 5.5. The message states that the first fairness constraint of the system (which is the guarantee `sys_fair_0`) could not be fulfilled.

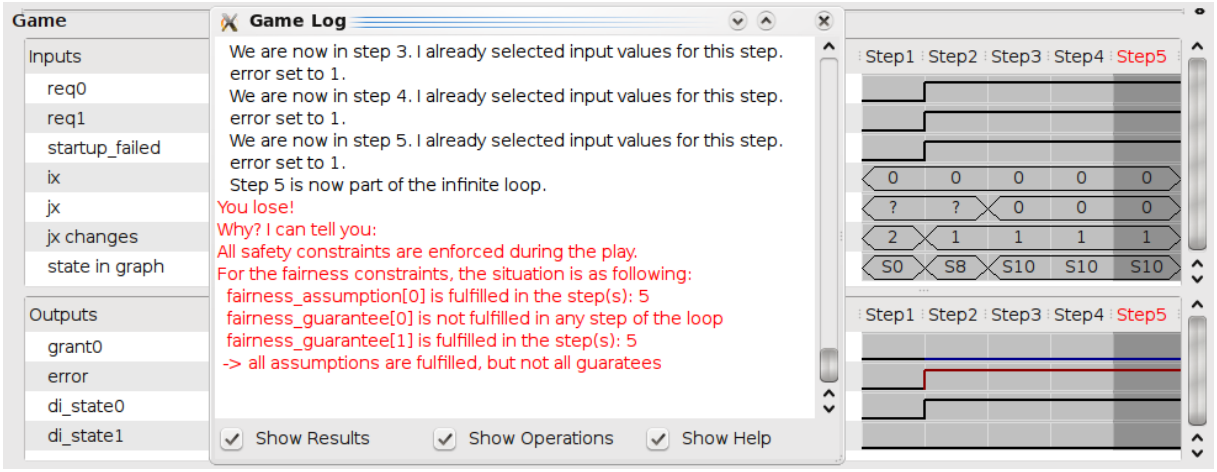


Figure 5.5: A screenshot of RATSY when playing a debugging game. It also contains the countertrace found by our heuristic, as well as the Game Log Window explaining why the user has lost the play, i.e., why she did not manage to fulfill the specification.

Additional Information Presented in the Debugging Game

As can be seen from Figure 5.5, there are some waveforms in the input trace that do not correspond to input signals. The waveforms `ix` and `jx` contain the memory content $(ix, jx) \in \Gamma$ of the counterstrategy[4]. Following the idea in Section 4.4.2, the counterstrategy is used in parallel with the countertrace in order to be able to present the memory content of the counterstrategy to the user. The symbol `?` in Figure 5.5 corresponds to the case where the environment has not yet decided which set J_j^s it tries to evade. The waveform `jx changes` contains the maximum number of times the content of `jx` might change in the future of the play (cf. Section 4.4.1). The stripe `state in graph` finally contains the current state of the play in the graph $\mathbb{G}$ (see also Figure 5.7). As already argued in Section 4.4.1, all this information can help to understand the cause of unrealizability.

Integration with the Automaton Editor of RATSY

The stripes `di_state0` and `di_state1` in the output part of the trace in Figure 5.5 contain the bits that are used for encoding the current state of the play in the DBW depicted in Figure 5.4. However, there is also a more comfortable way of keeping track of the current state of the play in the automata.

If the specification contains automata which were either constructed with the Automaton Editor of RATSY, or which were constructed automatically from design intents, the Automata Window of the game part of RATSY becomes available. It shows the current state of the play in all automata. For example, Figure 5.6 shows the Automata Window in Step 2 of the play. The user can select an automaton on the left-hand side (in case of Figure 5.6, there is only one). The automaton is then shown on the right-hand side. The current state of the play is marked yellow in the selected automaton. Furthermore, all edges, which are still possible with respect to all safety constraints and already undertaken user selections for signals values, are also highlighted in yellow. For instance, when the user sets `error=1` in the main window of the game, then only the self loop over the state `V2` is highlighted as a possible edge in the automaton of Figure 5.6. If the user wants to traverse a certain (yellow) edge in the automaton, she can also simply click onto this edge. All constraints on signal values associated with this edge will be added as an additional user selection in the main window of the game. Constraints on signals that are not under the control of the user are of course ignored.

[4]The reader might wonder about the value 0 for `ix`. The reason is that the counting of the sets of accepting states starts with 0 in the implementation while counting starts with 1 in our theoretical framework.

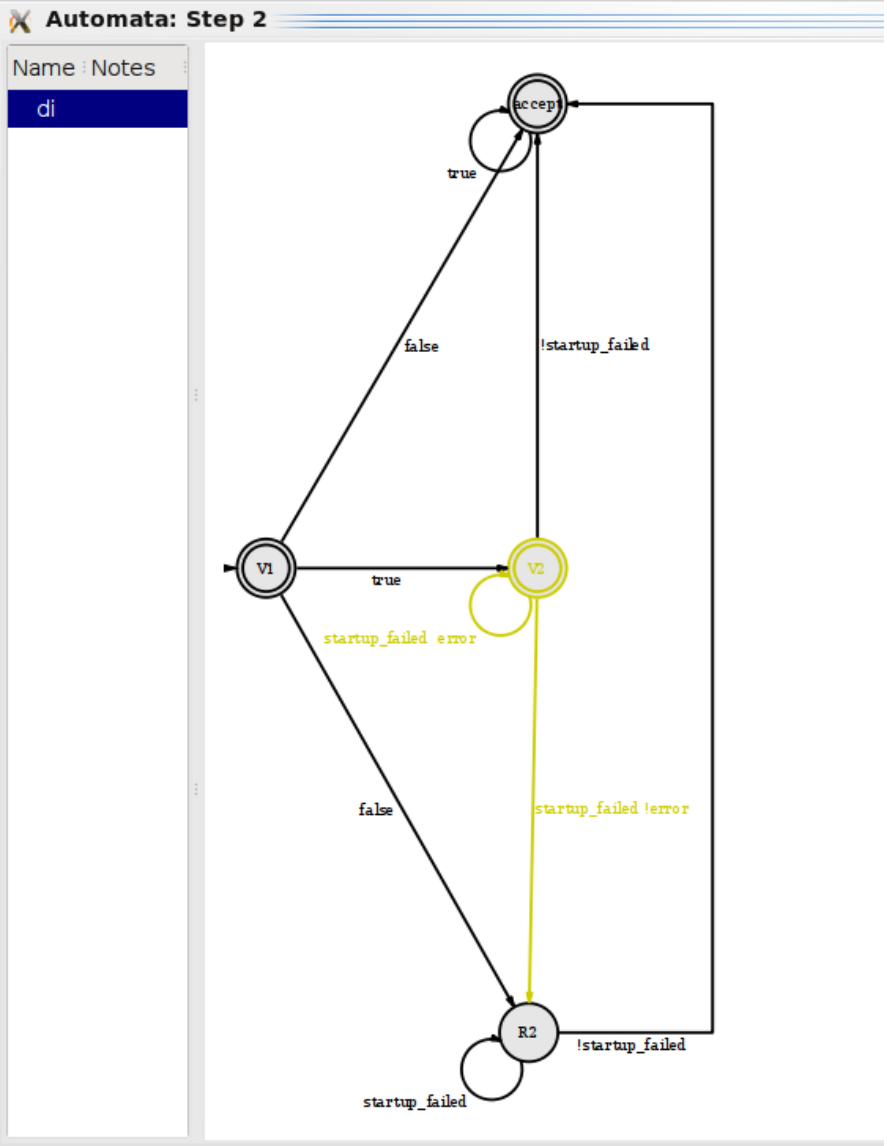


Figure 5.6: A screenshot of the Automata Window belonging to the game part of RATSY. On the left-hand side, the user can select one of the automata of the specification. On the right-hand side, the selected automaton is shown. The current state of the play in the automaton is highlighted in yellow. Also, all state transitions which are still possible in the current state of the play are marked yellow.

As an additional feature, not only the current step, but arbitrary time steps of the play can be displayed in the automata. This can be useful when analyzing a lost play, or when looking for better choices on signal values in previous time steps. All in all, the integration of the game part with the Automaton Editor of RATSY greatly increases usability and makes it easier for the user to stay on top of things in the game.

Recall our unrealizable specification. The automaton in Figure 5.6 makes the other part of the dilemma, into which the user is brought by the countertrace, obvious: The input `startup_failed` is always set according to the countertrace. If the user also sets the output signal `error`, she stays in the state `V2`, which is accepting. If she selects the value of `error` to be 0 in any time step, she will get trapped in the state `R2`, which is not accepting. Hence, when setting `error=0`, she loses the game. On the other hand, when setting `error=1`, she cannot give a grant, and she will lose because of being unable to fulfill the guarantee `sys_fair_0`, as already explained.

The Summarizing Graph $\mathbb{G}$

Figure 5.7 shows the graph $\mathbb{G}$ as created by RATSY for our example. Every vertex corresponds to a certain state-memory pair $(q, \gamma) \in (Q, \Gamma)$ that might occur in a play. As suggested in Section 4.4, some additional information is also written to the vertices of the graph. The meaning of the different values in a vertex is explained in the box labeled with "Explanation" in Figure 5.7.

Besides a unique name, every vertex contains the memory content $(ix, jx) \in \Gamma$. It also contains the indices i and j of all sets J_i^e and J_j^s to which the state belongs. The maximum number of changes of the memory content $jx \in \mathcal{J}$ in the future of the play is printed in the vertices too. Finally, the input letter $x \in \mathcal{X}$ chosen by the environment from a certain state is included in the vertices as well. This input letter is not written to the edges to keep the graph more readable. In fact, all inputs which have equal values for all vertices are printed only once in the left upper corner. Only changing inputs are written into the vertices. This is done to further increase the readability of the graph.

Every play starts in `S0`. Edges represent possible choices for the user. They are labeled with the corresponding values for the outputs. Output values that cannot be chosen due to safety constraints are not printed on the edges. Their value can be read form an extra file that is created together with the graph and contains detailed information about the graph. This extra file also contains the exact signal values corresponding to the vertices of the graph. As graphs tend to become quite large for larger specifications, a simplified representation of the graph is created by the tool as well. This simplified representation does not contain any signal values.

The graph in Figure 5.7 explains the unrealizability problem in our example specification as following. When the user selects `error=1` in all time steps, the play will get stuck in `S10` after two steps. The state corresponding to `S10` is not part of the set J_0^s of accepting states of the the system[5], so this set is not visited infinitely often. The set J_0^s represents the guarantee `sys_fair_0`. This means that `sys_fair_0` cannot be fulfilled when `error` is always set to 1. When the user sets `error` to 0 in any time step, the play will get trapped in the vertices `S3` to `S5`. None of the states represented by these vertices is element of J_1^s, so this set of accepting states of the system will be visited only finitely often. The set J_1^s represents the accepting states of the DBW depicted in Figure 5.4. This DBW enforces the design intent concerning the output `error`. Hence, when setting `error` to 0 in any time step, the user violates exactly this design intent.

5.3 Features Available from the Textual User Interface

Most of the specification debugging features have also been made available over a textual user interface to the tools Marduk and Anzu. Listing A.1 in Appendix A gives an idea of how a session with Marduk looks like when using this textual user interface. For Anzu, the output looks quite the same. The different processing steps can be enabled or disabled with command line arguments.

Of course the usability is much lower due to the limitations of the text based user interface. This is especially true for the interactive game. There is no graphical representation of the traces, so it is harder for the user to keep track of previous signal values during a play. Only when the user quits the game, a text file is created which contains a table summarizing all signal values in all time steps. This table looks similar to the traces in the GUI of RATSY. It is intended to help the user analyze a lost play. As a further intricateness, the user has to read the countertrace from an extra file in order to know in advance how the environment will behave during the play. Which parts of the specification have been removed can also be seen only from an extra file containing the minimized specification. Specifying design intents is not possible in the text mode at all. The reason is that it would be much more laborious for the user to specify the traces with the textual user interface, than to simply formulate the design intent in terms of properties of the specification directly.

[5] In case the reader is wondering about the index 0: While indices start at 1 in our theoretical framework, they start at 0 in the implementation.

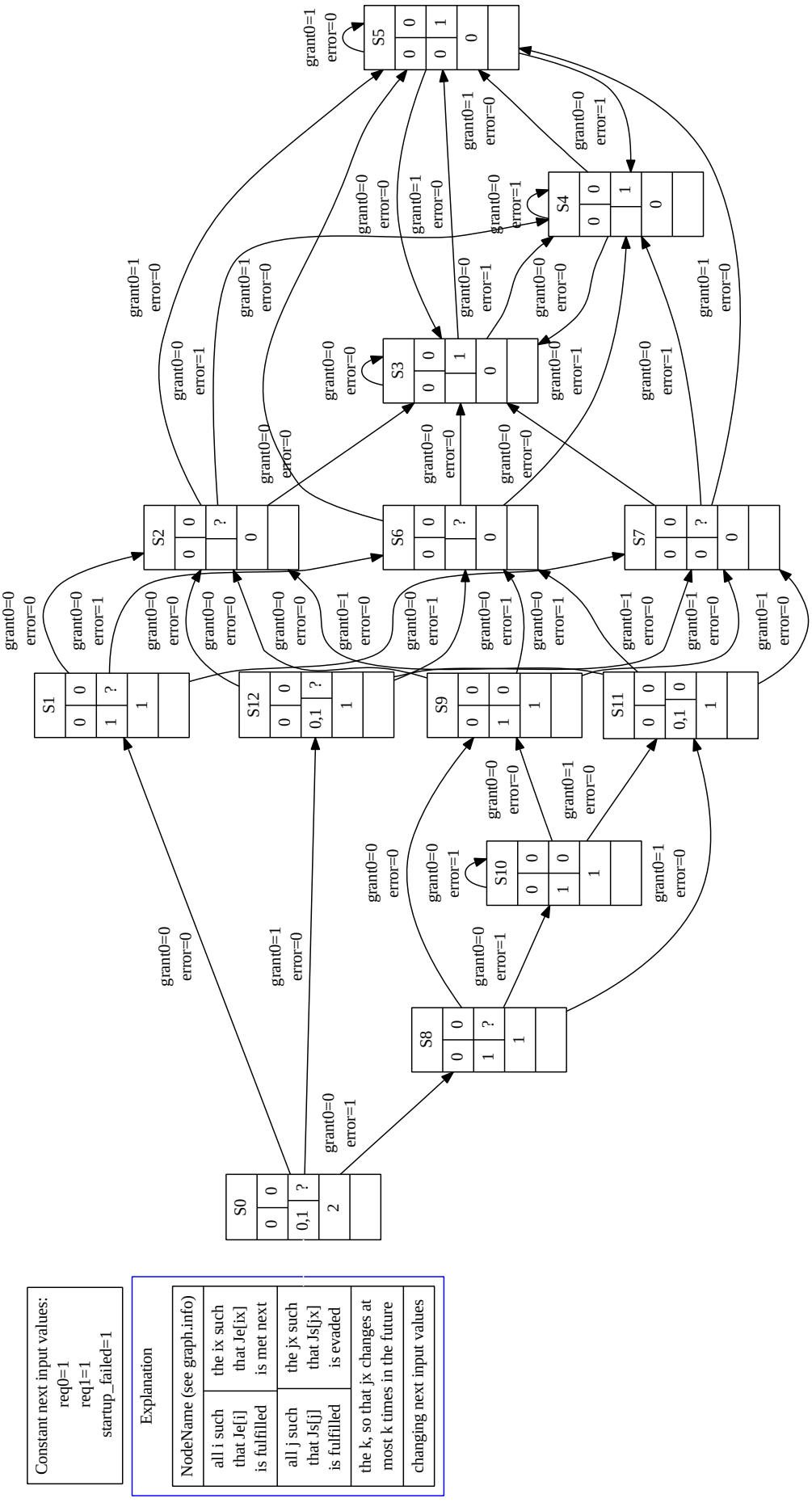


Figure 5.7: The automatically generated graph $\mathbb{G}$ which summarizes all possible plays for our example. An explanation to the values in the vertices is given on the left side. Vertices correspond to situations in the game, edges to choices of the user. All plays start in `S0`.

In the text based mode, detailed information about performance measures is printed to the user and logged into special files. Hence, this mode is perfectly suitable for scripts that automatically execute performance tests on lists of specifications to be used as input for the tool. This was done to obtain the results presented in Section 6.1.

5.4 Software Design

5.4.1 Integration into Marduk

Figure 5.8 depicts a class diagram which shows the software design for the implementation of our debugging method in Marduk. In order to keep the diagram simple, only the most important public methods are included. Attributes are not listed at all. Also, from the dependencies among the classes, only the most relevant ones are shown. Subordinated helper classes are not printed either.

The class `SpecDebugger` implements the flow of our debugging method as illustrated in Figure 3.2. This class is also responsible for collecting performance measures, which are then logged via the class `PLog`. `PLog` is implemented as a Singleton (cf. Gamma et al. [47]) in order to be accessible from all modules. The implementation of the different steps of the debugging procedure is encapsulated in other classes. The class `SpecDebugUtils` contains a lot of convenient utility functions that are used by almost all other classes.

The class `SatSolver` is able to perform a check for satisfiability with the algorithm of Listing 4.1. The `Minimizer` is an interface which abstracts the details of the minimization algorithms. Concrete minimization algorithms are implemented in the classes `DeltaDebugger` and `SimpleMinimizer`, where the latter contains the implementation of the algorithm as used by Cimatti et al. [21]. The interface `MinTarget` hides details of the subject to the minimization. It allows to handle any such subject in the same manner. It contains methods that return the set to be minimized, and it provides a method `test`, which checks subsets thereof with respect to a certain property. Currently, `RatMinTarget` is the only implementation. This class is able to return the set of guarantees and output signals. It furthermore implements the method `test` by performing a realizability check with a given subset of the guarantees and outputs. The idea of the interfaces is that every `Minimizer` can be used in the same way, and that every `Minimizer` can work with every `MinTarget`. This provides a maximum of flexibility and extendability.

The `CounterstrategyComputer` is able to compute the winning region as well as the winning strategy for the environment in a GR(1) game. Given the so computed counterstrategy, the class `CountertraceComputer` heuristically searches for a countertrace as described in Section 3.2.2. The countertrace is returned as an `InfiniteTraceOfBdds`, a class which simplifies the access to elements of the trace. The trace can be written into a file with the help of the `Writer`.

The counterstrategy and the countertrace, if found, are handled over to the `PathFinder`, which computes the graph $\mathbb{G}$. The graph is represented by a set of `GraphNodes`, where every `GraphNode` has references to its successors and predecessors. `GraphNodes` are furthermore able to transform themselves into their DOT representation. The class `InteractiveGame` finally implements the interactive game against the user providing a textual user interface. It uses the `Writer` to produce a summary of the play when the user quits.

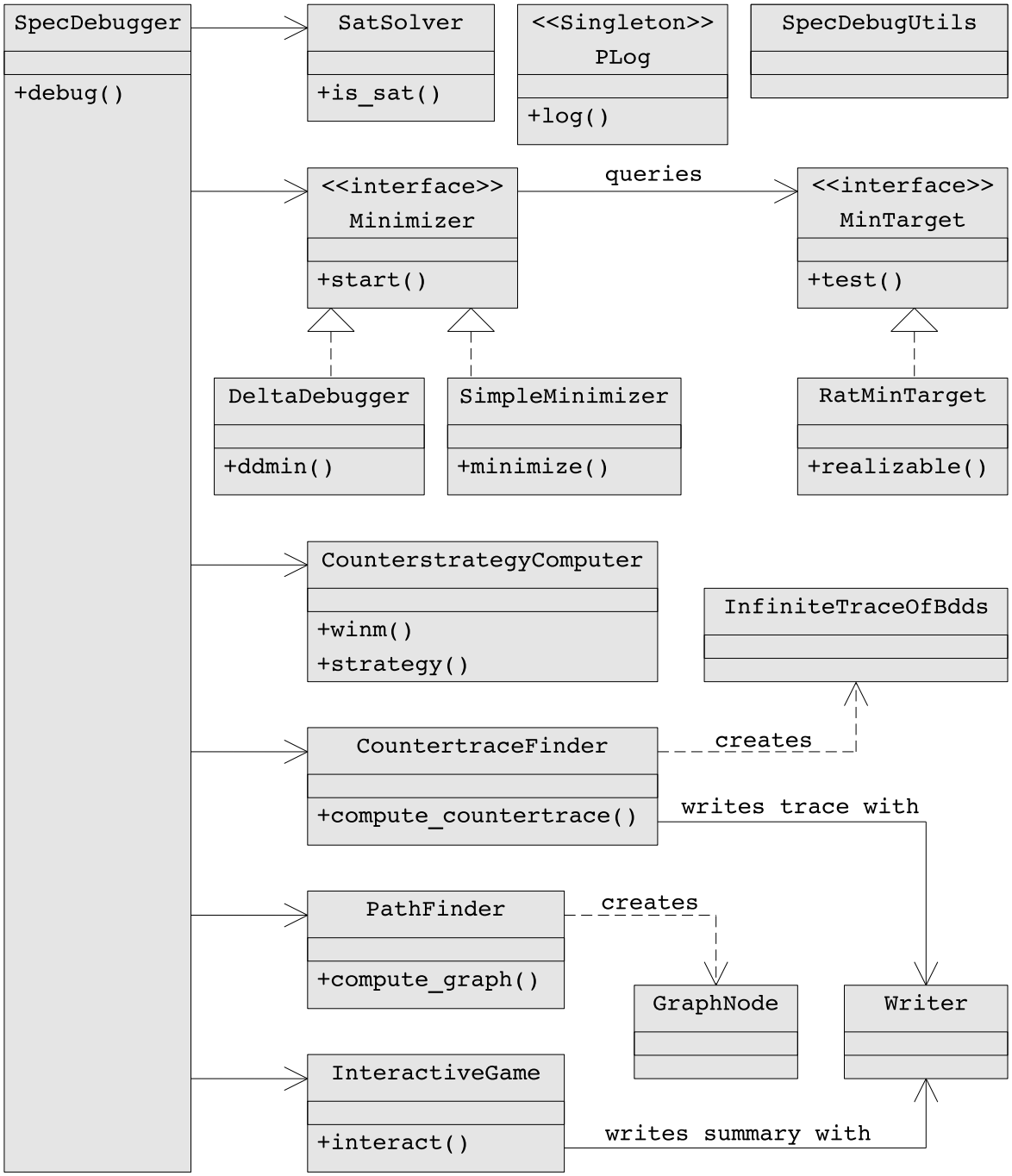


Figure 5.8: A class diagram showing the software design for the extension of Marduk. The `SpecDebugger` implements the debugging flow as depicted in Figure 3.2, utilizing classes that implement the different steps of this flow.

5.4.2 Integration into RATSY

Figure 5.9 gives a class diagram illustrating the software design for the integration of our debugging approach into RATSY. The existing GUI of RATSY is interconnected with the back-end of the tool over the Model-View-Controller (MVC) design pattern (cf. Buschmann et al. [15]), refined by the Observer

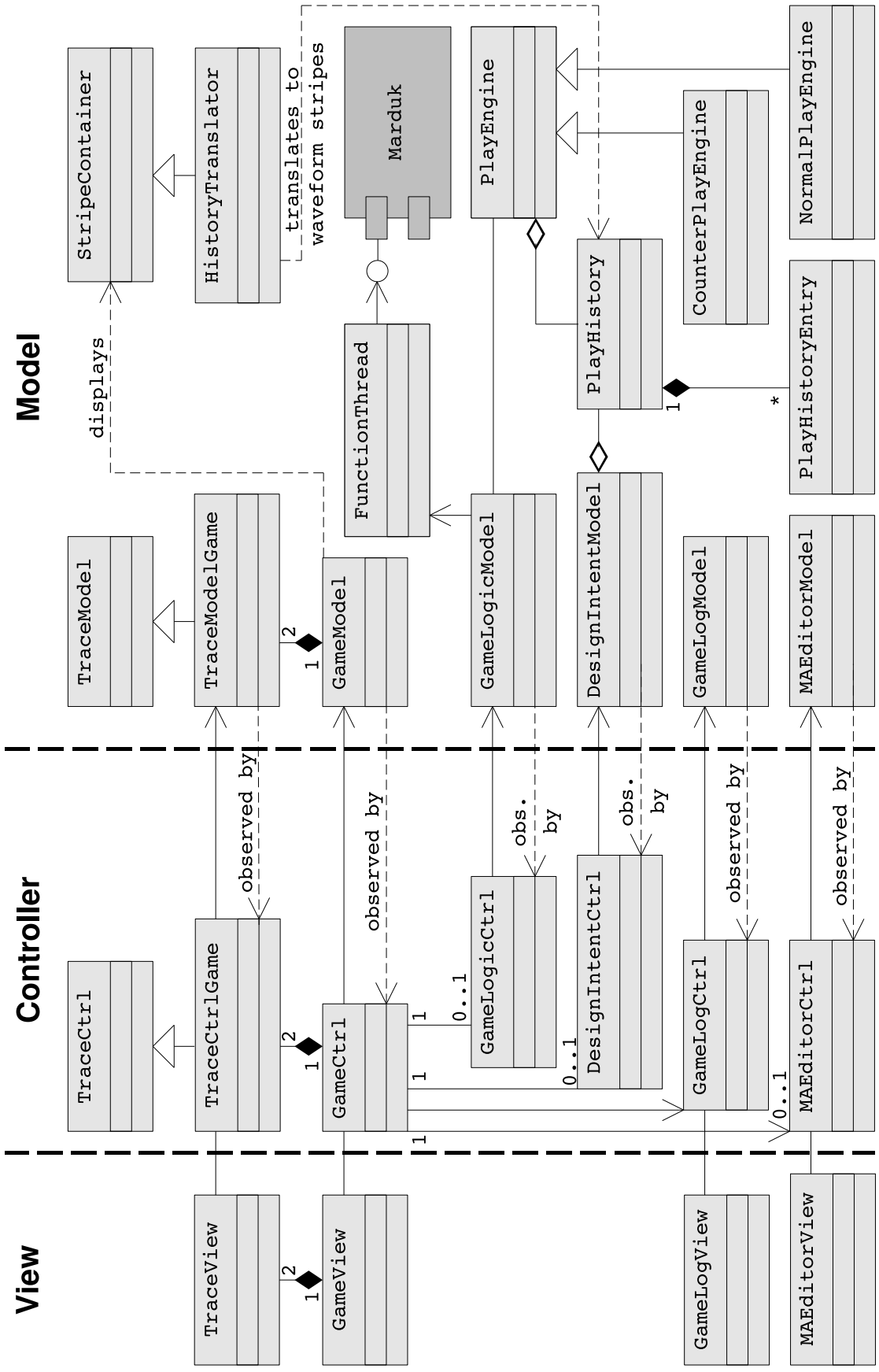


Figure 5.9: A class diagram showing the software design for the integration into RATSY. The connection to the GUI of RATSY is done with the MVC design pattern in combination with the observer pattern.

pattern (cf. Gamma et al. [47]). The implementation of these patterns is done with the pygtkmvc[6] framework [16]. This framework provides the base classes `Model`, `View`, and `Controller`, which implement the initialization and the communication between these three layers in a transparent manner.

The extension of the RATSY GUI in order to have the specification debugging features available is done in the same manner. Although not shown in Figure 5.9, all models, views, and controllers are derived from the respective base classes provided by the pygtkmvc framework.

Views

The views are responsible for the presentation of contents on the screen. The `GameView` represents the main window of the game as shown in Figure 5.2. It consists of two `TraceViews`, one for the input trace and one for the output trace, and it additionally presents some buttons and checkboxes. The class `TraceView` already existed and could be reused. Only the according model and controller had to be extended to make them suitable for the management of game traces. The `GameLogView` handles the presentation of the Game Log Window as depicted in Figure 5.2. The `MAEditorView` maintains the presentation of the Automata Window as shown in Figure 5.6. It is composed of some sub-views with corresponding sub-controllers and sub-models. Details on that are not shown in the class diagram and will also be omitted in this discussion.

Controllers

The controllers handle signals from the views, often by notifying them to the models. They furthermore observe changes in the corresponding models and trigger updates of the view.

In our design, the `GameCtrl` class can be seen as the main controller of the game. It maintains references to all other controllers. Not all other controllers are always available, though. While the user plays the game, there is no `DesignIntentCtrl` (and also no corresponding model). While the user specifies some design intent, the controller `GameCtrl` has no references to instances of the classes `GameLogicCtrl` and `MAEditorCtrl`. The `MAEditorCtrl` is also unavailable while playing the game if the specification does not contain any automata. These controllers and their respective models are created and destructed on demand.

The controllers have a further responsibility. In our design, different models often do not interact directly with each other. The often solely communicate over their controllers. This is done to keep the different models decoupled from each other. The `GameCtrl` serves as a mediator in this communication. For example, whenever the state of the play changes, the change is observed by the `GameLogicCtrl` and forwarded to the `GameCtrl`. The `GameCtrl` spreads the information to the classes `MAEditorCtrl` and `TraceCtrlGame` so that they can update their models. The changes in the models are again observed by the controllers, which in turn trigger updates of the views.

Models

The models implement the logic behind the available functionality. They store the relevant data for the application, and they provide methods to manipulate this data.

The back-end of our design works in the following way. The `GameLogicModel` utilizes `Marduk` in order check the specification for realizability. If the specification turns out to be realizable, `Marduk` is used to compute a winning strategy for the system. A `NormalPlayEngine` is created with the strategy. If the specification turns out to be unrealizable, the `GameLogicModel` uses `Marduk` in order to check for satisfiability, to minimize the specification, to compute a counterstrategy and a countertrace, and to compute the summarizing graph $\mathbb{G}$ (see also Figure 5.8). Only the interactive debugging game is not

[6] `http://sourceforge.net/projects/pygtkmvc/` (last visit in October of 2009)

played using Marduk. Instead, a CounterPlayEngine is created, which implements the debugging game.

The class FunctionThread, which already existed and could be reused, is employed to execute expensive functions provided by Marduk. Instead of calling the functions directly, this class executes them in an own thread. Executing expensive functions within the thread of the user interface would cause the user interface the be unresponsive during the operation. With the help of the class FunctionThread, the user interface remains operable.

After the initialization phase, the GameLogicModel does not have to care about the differences between the testing game and the debugging game any longer. It simply forwards requests (e.g., requests to start a new time step, to set a certain signal to a certain value, etc.) to its PlayEngine. The concrete instance of the PlayEngine (either a NormalPlayEngine or a CounterPlayEngine) handles the requests correctly according to the rules defined by the kind of the game. Deriving the two different play engines from a common base class also allows to share code between the implementations of the two kinds of games. After all, the two kinds of games do not differ that much.

The history of the play is maintained by the class PlayHistory. It contains an instance of the class PlayHistoryEntry for every time step that has already been played. A HistoryTranslator is used to translate the information into waveform stripes, i.e., into a format that is understood by the TraceModel. The PlayEngine could also have been built to manipulate the TraceModel directly. The indirection with the HistoryTranslator is performed to have a higher flexibility in the representation of the data inside the PlayEngine. Also, the PlayEngine is more decoupled from the traces when using a translator. The efficiency of the implementation does not suffers perceptibly from the translation.

When the user switches from playing the game into the *Specify Design Intent* mode, an instance of a DesignIntentModel (and a corresponding controller) is created. It is initialized with the PlayHistory of the PlayEngine, after which the GameLogicModel and the PlayEngine are destructed. From that point on, the GameCtrl forwards all requests to the DesignIntentModel instead of the GameLogicModel.

6 Experimental Results

We used two different GR(1) specifications to evaluate our approach. Both are parameterized. The first one defines an arbiter for the AMBA AHB bus [7]. It is parameterized with the number of bus masters. We will write `An` to denote this specification for n bus masters. Furthermore, we will write `Anei` to denote mutants of the specification, where `e` describes the kind of modification and `i` is a running index. The term `woef` inserted for `e` states that some sub-formula was removed from the part φ_g^e of the GR(1) specification (see Section 2.2). The term `wsf` means that some conjunct was added to φ_g^s, and `wst` means that the part φ_t^s of the GR(1) specification was further restricted. The second specification we utilize is a generalized buffer [8], which is used by n senders and two receivers. Again, we write `Gn` to denote the unmodified specification for n senders. We write `Gnei` to denote mutants of this specification with the same syntax as for the bus arbiter specification. All mutants of the two specifications are satisfiable but not realizable.

6.1 Performance Evaluation

6.1.1 Performance Results

The Tables 6.1, 6.2, 6.3, and 6.4 summarize the results of our performance evaluation for Marduk. The results for Anzu are similar and thus not printed in this document. All experiments were performed on a machine equipped with an Intel® Core™ 2 Duo Mobile Processor P7350 and 3 GB of RAM. Kubuntu 8.10 served as operating system, and Python 2.5.2 was used as interpreter.

Table 6.1 and Table 6.2 show the results when the minimization step is skipped. Table 6.1 gives the results for the different mutants of the AMBA AHB bus arbiter specification, and Table 6.2 does so for the variants of the specification of the generalized buffer. The two tables have the same columns. The first two columns give an idea of the size of the examined specification variants by listing the number of signals and the number of properties. Column 3 presents the time needed for the satisfiability check with the algorithm of Listing 4.1. Column 4 and Column 5 list the time for the computation of the winning region for the environment and the system, respectively. The times needed for the computation of a winning strategy for the environment are presented in Column 6. Column 7 gives the time needed for the computation of a winning strategy for the system in the game that was constructed from the unmodified and thus realizable specification. Column 8 lists the number of vertices in the graph $\mathbb{G}$ that summarizes all possible plays, and Column 9 shows the time needed for the computation of the DOT description of this graph. The computation of the graph was aborted if it exceeded $1\,000$ vertices. Such cases are indicated with $> 1\,k$ in Column 8. Column 10 indicates if our heuristic was able to find a countertrace. If so, the last two columns of the Tables 6.1 and 6.2 give the length of the finite stem and the length of the infinite loop of the lasso-shaped countertrace. The time needed for the computation of the countertrace when given a counterstrategy, or the time until this computation aborts without success, is negligible (< 1 second in all cases) and thus not included in the tables. Entries preceded with a $>$ indicate time-outs. As a consequence of such time-outs, data for other columns may be missing. This is indicated with "?" in the tables.

The Tables 6.3 and 6.4 summarize the results when the minimization step is performed for the variants of the two specifications. Column 1 lists the number of realizability checks needed when using the minimization method of Cimatti et al. [21], which we reimplemented for comparison. Column 2 gives the number of realizability checks when Delta Debugging is used. Column 3 relates the first two columns with each other by listing the reduction in the number of checks due to the use of Delta Debugging. The Columns 4 and 5 give the times for minimization with the algorithm of Cimatti et al. [21] and with Delta Debugging, respectively. The time savings when Delta Debugging is used are shown as speed-up factor in Column 6. The Columns 7 and 8 present the number of guarantees and output variables before and

Table 6.1: Performance results for the mutants of the AMBA AHB bus arbiter [7] specification when the minimization step is skipped.

column	1	2	3	4	5	6	7	8	9	10	11	12
	# signals	# properties	time: SAT-check	time: W_{env}	time: W_{sys}	time: ρ_{env}	time: ρ_{sys}	# vertices in $\mathbb{G}$	time: Graph $\mathbb{G}$	$\overline{\tau}$ found	$\lvert stem(\overline{\tau})\rvert$	$\lvert loop(\overline{\tau})\rvert$
	[-]	[-]	[sec]	[sec]	[sec]	[sec]	[sec]	[-]	[sec]	[-]	[-]	[-]
A2woef1	22	90	0.1	0.3	0.4	0.5	2.6	27	0.6	yes	3	1
A3woef1	28	121	0.3	1.8	1.8	2.2	118	43	1.9	yes	3	1
A4woef1	34	152	1.1	39	25	9.0	2.6 k	75	8.0	yes	3	1
A5woef1	40	183	7.2	307	146	288	12 k	139	43	yes	3	1
A6woef1	50	213	4.7	439	147	212	53 k	267	320	yes	3	1
A2wsf1	22	92	0.1	0.4	0.9	0.2	2.6	59	1.0	yes	5	2
A3wsf1	28	123	0.3	3.4	5.5	2.3	118	251	12	yes	5	2
A4wsf1	34	154	0.9	57	96	7.2	2.6 k	171	7.6	yes	5	2
A5wsf1	40	185	4.0	389	520	104	12 k	683	205	yes	5	2
A6wsf1	50	215	4.0	640	569	297	53 k	715	181	yes	5	2
A2wsf2	22	92	0.1	0.5	1.2	0.2	2.6	9	0.1	yes	3	2
A3wsf2	28	123	0.2	3.5	6.3	2.4	118	9	0.2	yes	3	2
A4wsf2	34	154	0.8	35	128	25	2.6 k	35	0.9	yes	5	2
A5wsf2	40	185	3.4	391	579	159	12 k	9	0.2	yes	3	2
A6wsf2	50	215	3.4	669	629	316	53 k	9	0.3	yes	3	2
A2wst1	22	92	0.1	0.3	0.3	0.4	2.6	51	0.6	yes	5	2
A3wst1	28	123	0.2	0.8	1.1	1.8	118	211	8.0	yes	5	2
A4wst1	34	154	1.4	3.9	5.0	3.6	2.6 k	139	4.8	yes	5	2
A5wst1	40	185	20	41	582	55	12 k	>1 k	241	yes	5	2
A6wst1	50	215	10	116	70	110	53 k	683	142	yes	5	2
A2wst2	22	92	0.1	0.2	0.2	0.1	2.6	7	0.1	yes	3	2
A3wst2	28	123	0.1	0.5	0.8	0.3	118	7	0.1	yes	3	2
A4wst2	34	154	0.4	13	5.9	1.7	2.6 k	7	0.1	yes	3	2
A5wst2	40	185	0.4	22	15	4.3	12 k	7	0.1	yes	3	2
A6wst2	50	215	0.7	41	84	6.8	53 k	7	0.1	yes	3	2
total			64	3.2 k	3.6 k	1.6 k	339 k		1.2 k			

after Delta Debugging has been applied. The reduction in this number is shown in Column 9. The minimization method of Cimatti et al. approximately leads to the same reduction. Hence, the exact values are not included in the tables. Whether or not a countertrace could be found is indicated in Column 10. If a countertrace was found, the length of its finite stem as well as the length of its infinite loop are listed in the Columns 11 and 12. The last column finally contains the number of vertices in the summarizing graph $\mathbb{G}$. The times needed for the computation of this graph, for the computation of the winning region of the environment, for the computation of the counterstrategy, and for the computation of the countertrace are negligible after minimization. All these times are therefore not listed in the Tables 6.3 and 6.4. Again, entries preceded with a $>$ indicate time-outs and missing data due to time-outs is marked with "?".

Table 6.2: Performance results for the mutants of the generalized buffer [7] specification when the minimization step is skipped. Time-outs are preceded with $>$ and missing data due to a time-out is marked with "?".

column	1	2	3	4	5	6	7	8	9	10	11	12
	# signals	# properties	time: SAT-check	time: W_{env}	time: W_{sys}	time: ρ_{env}	time: ρ_{sys}	# vertices in $\mathbb{G}$	time: Graph $\mathbb{G}$	τ found	$\lvert stem(\tau)\rvert$	$\lvert loop(\tau)\rvert$
	[-]	[-]	[sec]	[sec]	[sec]	[sec]	[sec]	[-]	[sec]	[-]	[-]	[-]
G5woef1	24	108	0.04	0.2	0.1	0.3	0.2	192	10	no	-	-
G20woef1	56	440	0.3	1.4	0.6	3.4	7.0	>1 k	653	no	-	-
G40woef1	97	1 231	1.2	6.0	1.5	18	71	>1 k	4.3 k	no	-	-
G60woef1	137	2 421	2.8	15	3.4	53	516	>1 k	632	no	-	-
G80woef1	178	4 012	5.9	52	6.3	216	1.7 k	?	>10 k	no	-	-
G100woef1	218	6 002	9.2	62	8.0	255	3.1 k	?	>10 k	no	-	-
G5wsf1	24	110	0.04	0.1	0.4	0.1	0.2	249	15	no	-	-
G20wsf1	56	442	0.2	0.8	5.0	0.6	7.0	789	911	no	-	-
G40wsf1	97	1 233	0.7	3.2	20	2.2	71	>1 k	4.8 k	no	-	-
G60wsf1	137	2 423	1.9	7.1	49	4.5	516	309	175	no	-	-
G80wsf1	178	4 014	3.4	18	195	10	1.7 k	?	>10 k	no	-	-
G100wsf1	218	6 004	4.7	27	228	14	3.1 k	?	>10 k	no	-	-
G5wsf2	24	110	0.02	0.1	0.1	0.1	0.2	>1 k	216	no	-	-
G20wsf2	56	442	0.1	1.0	0.1	0.6	7.0	>1 k	2.1 k	no	-	-
G40wsf2	97	1 233	0.5	5.0	0.3	2.0	71	>1 k	3.2 k	no	-	-
G60wsf2	137	2 423	0.9	11	0.6	4.6	516	>1 k	7.2 k	no	-	-
G80wsf2	178	4 014	2.6	34	1.0	11	1.7 k	?	>10 k	no	-	-
G100wsf2	218	6 004	3.6	41	1.6	13	3.1 k	?	>10 k	no	-	-
G5wst1	24	110	0.02	0.1	0.1	0.1	0.2	7	0.1	yes	3	2
G20wst1	56	442	0.1	0.5	0.2	0.9	7.0	22	1.5	yes	3	2
G40wst1	97	1 233	0.3	1.6	0.6	3.5	71	40	8.7	yes	3	2
G60wst1	137	2 423	0.7	4.6	1.0	9.7	516	10	1.0	yes	3	2
G80wst1	178	4 014	1.3	7.6	1.9	27	1.7 k	76	77	yes	3	2
G100wst1	218	6 004	2.0	16	1.7	36	3.1 k	46	32	yes	3	2
G5wst2	24	110	0.02	0.1	0.1	0.1	0.2	37	0.9	no	-	-
G20wst2	56	442	0.1	0.5	0.2	0.9	7.0	118	28	no	-	-
G40wst2	97	1 233	0.4	1.8	0.5	4.8	71	226	297	no	-	-
G60wst2	137	2 423	0.7	3.9	1.1	12	516	46	9.8	no	-	-
G80wst2	178	4 014	1.2	9.6	1.8	24	1.7 k	442	5.2 k	no	-	-
G100wst2	218	6 004	1.8	18	1.7	47	3.1 k	262	1.1 k	no	-	-
total			47	349	532	774	27 k		31 k[a]			

[a] Not including G80woef1, G100woef1, G80wsf1, G100wsf1, G80wsf2, and G100wsf2.

Table 6.3: Performance results for the mutants of the AMBA AHB bus arbiter [7] specification when the minimization step is carried out. DD is short for Delta Debugging.

column	1	2	3	4	5	6	7	8	9	10	11	12	13
	# checks in [21]	# checks during DD	reduction of checks	time: $\hat{\varphi}$ in [21]	time: $\hat{\varphi}$ with DD	speed-up factor	$\lvert G \cup Y \rvert$	$\lvert \hat{G} \cup \hat{Y} \rvert$	reduction of $\lvert G \cup Y \rvert$	$\overline{\tau}$ found	$\lvert stem(\overline{\tau}) \rvert$	$\lvert loop(\overline{\tau}) \rvert$	# vertices in $\mathbb{G}$
	[-]	[-]	[%]	[sec]	[sec]	[-]	[-]	[-]	[%]	[-]	[-]	[-]	[-]
A2woef1	80	47	41	7.6	1.3	5.8	80	9	89	yes	3	1	5
A3woef1	108	52	52	36	2.4	15	108	10	91	yes	3	1	13
A4woef1	136	56	59	165	3.5	47	136	11	92	yes	3	1	13
A5woef1	164	52	68	2.6 k	5.3	499	164	12	93	yes	3	1	5
A6woef1	191	58	70	1.4 k	7.8	179	191	12	94	yes	3	1	29
A2wsf1	81	37	54	11	1.2	9.2	81	8	90	yes	2	2	5
A3wsf1	109	40	63	63	2.3	27	109	8	93	yes	2	2	5
A4wsf1	137	46	66	617	3.5	176	137	9	93	yes	2	2	5
A5wsf1	165	48	71	17 k	8.0	2 181	165	9	95	yes	2	2	5
A6wsf1	192	47	76	5.2 k	11	473	192	9	95	yes	2	2	5
A2wsf2	81	53	35	11	2.7	4.1	81	12	85	yes	2	2	7
A3wsf2	109	59	46	63	9.4	6.7	109	11	90	yes	2	2	17
A4wsf2	137	77	44	649	40	16	137	20	85	yes	2	2	13
A5wsf2	165	65	61	16 k	225	73	165	12	93	yes	2	2	17
A6wsf2	192	69	64	5.0 k	303	17	192	12	94	yes	2	2	17
A2wst1	81	41	49	6.0	1.4	4.3	81	9	89	yes	3	2	7
A3wst1	109	47	57	27	3.2	8.4	109	10	91	yes	3	2	19
A4wst1	137	52	62	120	4.3	28	137	11	92	yes	3	2	19
A5wst1	165	51	69	1.1 k	12	92	165	12	93	yes	3	2	43
A6wst1	192	55	71	3.5 k	9.3	371	192	12	94	yes	3	2	43
A2wst2	81	46	43	6.9	1.9	3.6	81	10	88	yes	3	2	7
A3wst2	109	51	53	28	3.7	7.6	109	11	90	yes	3	2	31
A4wst2	137	57	58	79	5.2	15	137	12	91	yes	3	2	19
A5wst2	165	56	66	166	6.8	24	165	13	92	yes	3	2	63
A6wst2	192	62	68	267	9.7	28	192	13	93	yes	3	2	63
total	3.4 k	1.3 k	61	55 k	684	80	3.4 k	277	92				

6.1.2 Discussion

Results without minimization

Computation Times: The SAT-check is very fast compared to all other computations. The times needed for the computation of a winning region for the environment are approximately equal to the times needed for the computation of the winning region for the system, which is not surprising. Fortunately, the computation of a counterstrategy is faster than the computation of a winning strategy for the system

Table 6.4: Performance results for the mutants of the generalized buffer [7] specification when the minimization step is carried out. DD is short for Delta Debugging, time-outs are preceded with >, and missing data due to a time-out is marked with "?".

column	1	2	3	4	5	6	7	8	9	10	11	12	13
	# checks in [21]	# checks during DD	reduction of checks	time: $\hat{\varphi}$ in [21]	time: $\hat{\varphi}$ with DD	speed-up factor	$\lvert G \cup Y \rvert$	$\lvert \hat{G} \cup \hat{Y} \rvert$	reduction of $\lvert G \cup Y \rvert$	$\overline{\tau}$ found	$\lvert stem(\overline{\tau}) \rvert$	$\lvert loop(\overline{\tau}) \rvert$	# vertices in $\mathbb{G}$
	[-]	[-]	[%]	[sec]	[sec]	[-]	[-]	[-]	[%]	[-]	[-]	[-]	[-]
G5woef1	81	50	38	3.5	2.4	1.5	81	15	81	no	-	-	52
G20woef1	368	70	81	61	14	4.4	368	15	96	no	-	-	52
G40woef1	1.1 k	83	92	715	35	20	1.1 k	15	99	no	-	-	52
G60woef1	2.2 k	82	96	4.4 k	94	47	2.2 k	15	99	no	-	-	52
G80woef1	3.8 k	85	98	19 k	206	93	3.8 k	15	99	no	-	-	52
G100woef1	5.7 k	89	98	60 k	253	236	5.7 k	15	99	no	-	-	52
G5wsf1	82	76	7.3	3.7	5.0	0.7	82	19	77	yes	2	2	3
G20wsf1	369	220	40	97	409	0.2	369	49	87	yes	2	2	3
G40wsf1	1.1 k	?	?	1.2 k	>70 k	?	1.1 k	90[a]	92[a]	yes[a]	2[a]	2[a]	5[a]
G60wsf1	2.2 k	?	?	6.6 k	>70 k	?	2.3 k	130[a]	94[a]	yes[a]	2[a]	2[a]	5[a]
G80wsf1	3.8 k	?	?	25 k	>70 k	?	3.8 k	170[a]	95[a]	yes[a]	2[a]	2[a]	5[a]
G100wsf1	?	?	?	>70 k	>70 k	?	5.7 k	?	?	?	?	?	?
G5wsf2	82	32	61	2.1	0.8	2.6	82	6	93	no	-	-	36
G20wsf2	369	52	86	37	4.7	7.9	369	13	96	no	-	-	58
G40wsf2	1.1 k	40	96	556	16	35	1.1 k	5	99	no	-	-	36
G60wsf2	2.2 k	57	97	3.8 k	60	64	2.3 k	6	99	no	-	-	36
G80wsf2	3.8 k	53	99	17 k	108	155	3.8 k	6	99	no	-	-	36
G100wsf2	5.7 k	55	99	56 k	181	309	5.7 k	6	99	no	-	-	36
G5wst1	82	34	59	2.6	1.1	2.4	82	7	91	yes	3	2	4
G20wst1	369	42	89	50	5.7	8.8	369	7	98	yes	3	2	4
G40wst1	1.1 k	55	95	641	19	34	1.1 k	7	99	yes	3	2	4
G60wst1	2.2 k	55	98	4.1 k	57	72	2.3 k	7	99	yes	3	2	4
G80wst1	3.8 k	60	98	18 k	111	161	3.8 k	7	99	yes	3	2	4
G100wst1	5.7 k	59	99	57 k	238	240	5.7 k	7	99	yes	3	2	4
G5wst2	82	43	48	2.9	1.7	1.7	82	9	89	yes	3	2	4
G20wst2	369	54	85	51	7.5	6.8	369	9	98	yes	3	2	4
G40wst2	1.1 k	65	94	651	22	30	1.1 k	9	99	yes	3	2	4
G60wst2	2.2 k	68	97	4.1 k	68	61	2.3 k	9	99	yes	3	2	4
G80wst2	3.8 k	69	98	18 k	125	141	3.8 k	9	99	yes	3	2	4
G100wst2	5.7 k	73	99	57 k	263	215	5.7 k	9	99	yes	3	2	4
total	53 k[b]	1.7 k[b]	97[b]	312 k[b]	2.3 k[b]	139[b]	53 k[b]	334[b]	99[b]				

[a] Computed for the minimized specification obtained by the algorithm of Cimatti et al. [21].
[b] Not including G40wsf1, G60wsf1, G80wsf1, and G100wsf1.

in most cases, especially for larger specification variants. A possible reason is that the inputs dictated by the counterstrategy depend on the current state and the current memory content of the counterstrategy only. The outputs dictated by a strategy for the system additionally depend on the given inputs. That is, the strategy for the system additionally has to be able to react to the moves of the environment, since the system acts like a Mealy machine while the environment acts like a Moore machine (cf. Section 2.4).

Graphs: The size of the graph $\mathbb{G}$ which summarizes all possible plays is a good indicator for how simple it is for the user to understand the cause of unrealizability when playing the interactive game against the counterstrategy. Without minimization, the number of vertices in this graph is rather high in most cases. The graph is often way too complex to be visualized or even analyzed by the user. The computation of the DOT description of the graph is fast if the graph does not contain too many vertices. For large graphs, the computation time is rather high. However, as such large graphs are typically no help for the user, their computation can be aborted much earlier than done in our experiments. The circumstance that a lot of time is sometimes spent on graph computation until this computation aborts without success is annoying. The root of this problem is that the depth first search algorithm looking for new graph nodes has often collected huge amounts of backtracking possibilities until the limit of graph nodes is reached. See Section 8.3.2 for a solution to this problem.

Countertraces: For the mutants of the AMBA AHB bus arbiter specification, our heuristic for computing countertraces performs well: It is able to find a countertrace in all cases. For the specification variants of the generalized buffer, a countertrace is found for only one kind of modification, at least without minimization. The time for the computation of a countertrace is negligible when a counterstrategy is given. The length of the finite stems and the infinite loops of the countertraces are quite low. The overall length (stem plus loop) corresponds to the number of iterations performed by our heuristic. In theory, this number of iterations is exponential in the number of states in the worst case. Much to our pleasure, this number is rather low in our experiments. Hence, the quantity of iterations to perform can be limited to a relatively low value without making the heuristic fail in significantly more cases.

Improvements due to Minimization

Reduction in the Number of Checks due to Delta Debugging: Compared to the simple minimization algorithm of Cimatti et al. [21], the number of realizability checks which are necessary in order to compute an unrealizable core are reduced with the use of the Delta Debugging algorithm by about 95 % in total. Additionally, there is a trend to higher reduction rates for larger specifications. Although no such case is contained in our experimental results, Delta Debugging can also require more checks for realizability than the simple algorithm. This is in particular the case if the specification that should be minimized is already nearly an unrealizable core, i.e., if only a few guarantees or output signals can be removed while preserving unrealizability. In such a case, Delta Debugging wastes a lot of checks not removing anything and only increasing its granularity until it finally behaves similar to the simple minimization algorithm (see also Section 2.7).

Reduction in the Computation Time due to Delta Debugging: Although one could presume so, there is not always a strong correlation between the number of checks and the time needed for minimization. The reason is that the time per check is not constant. As a consequence, the reduction in the computation time is even higher than the reduction in the number of checks in most of our experiments. All in all, the time needed for minimization is reduced by a factor of 125 when using Delta Debugging instead of the simple minimization algorithm. Figure 6.1 illustrates the reduction in the form of a scatter plot. Delta Debugging needs more time only for the specification variants `Gnwsf1`. For these mutants, there is a reduction in the number of checks due to Delta Debugging, but there is an increase in the computation time. This circumstance is investigated in the next subsection. For all other specification variants, Delta

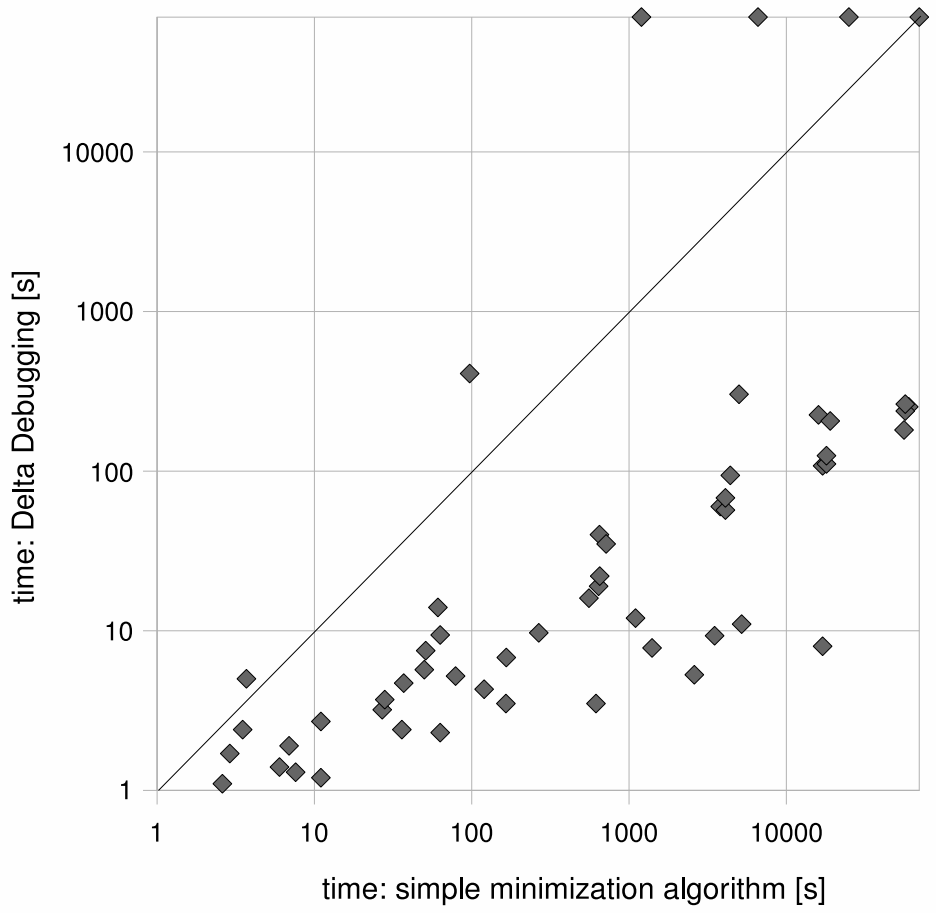


Figure 6.1: The reduction in the computation time due to the use of Delta Debugging instead of the simple minimization algorithm as a scatter plot. In most cases, Delta Debugging is much faster.

Debugging is much faster. Note that the axes are scaled logarithmically. With a linear scaling, all points would be located close to the abscissa and one could not distinguish them clearly.

Amount of Reduction: The number of guarantees and output signals is reduced greatly due to minimization. In average, a reduction of about 95 % can be achieved. The high reductions are not specific to Delta Debugging, the method of Cimatti et al. leads to similar results.

Countertraces: Minimization also increases the chances of finding a countertrace. Without minimization, our heuristic is able to find a countertrace in about 55 % of the cases. When minimization is applied, our heuristic succeeds for about 80 % of the specification variants. Also, the found countertraces are often shorter in their representation than the countertraces found without minimization.

Graphs: Finally, minimization greatly reduces the size of the graphs $\mathbb{G}$ which summarize all possible plays. Figure 6.2 illustrates this reduction as a scatter plot. The graph size slightly increases only in a few cases (`Anwst2` and `Anwsf2` for some n) in which the graph is rather small anyway. It is dramatically reduced in most cases. Note that the axes are scaled logarithmically. With a linear scaling, all points would be located close to the abscissa and one could not distinguish single points clearly. The significant reductions indicate that the underlying game is greatly simplified and that it is way easier to understand the cause of unrealizability in the interactive game played on the minimized specification.

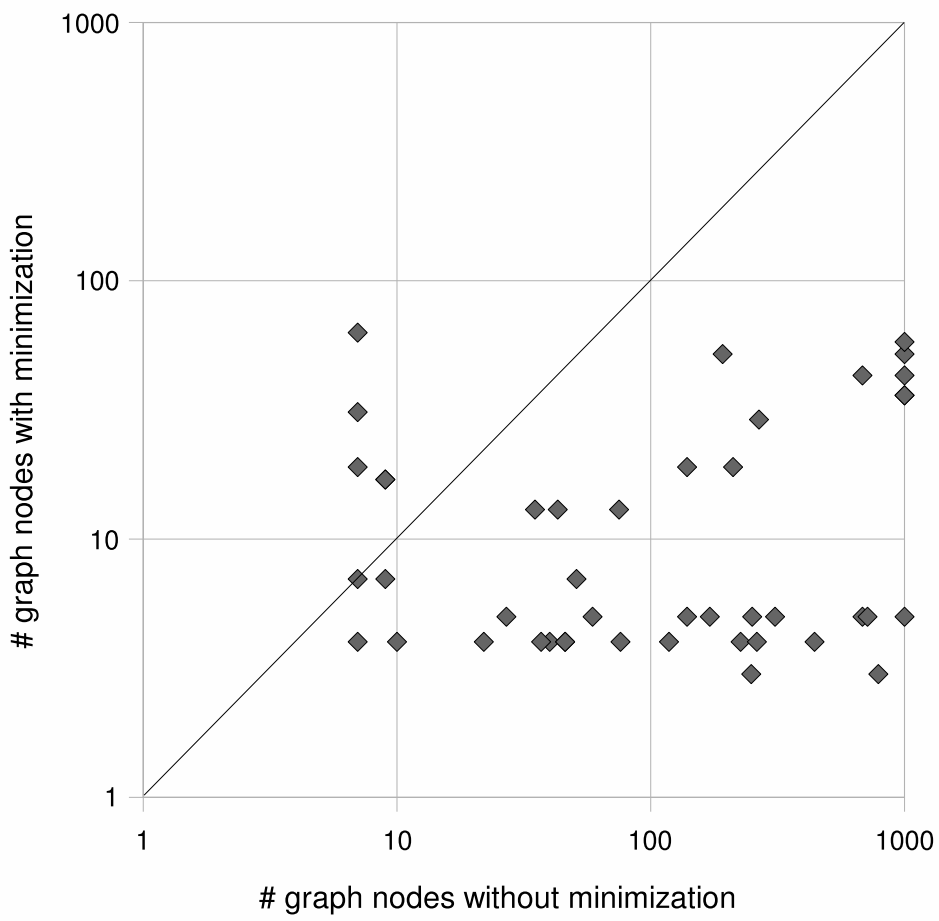


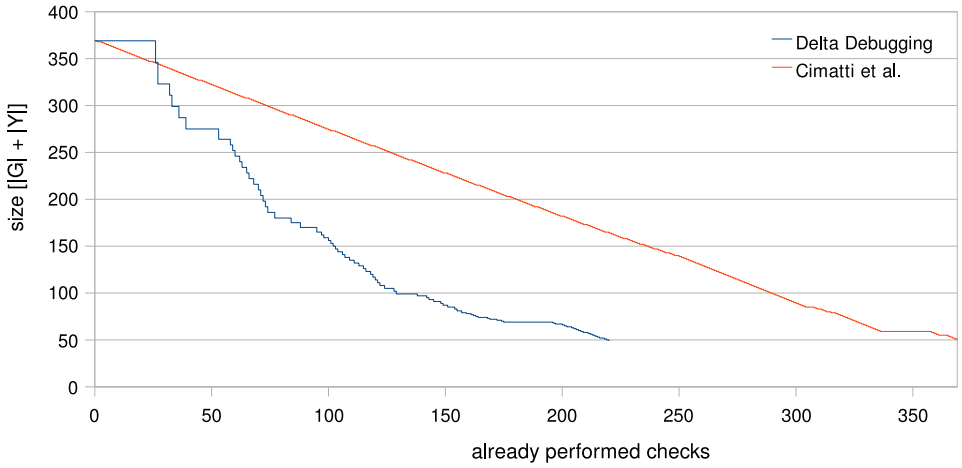
Figure 6.2: The reduction in the graph size due to minimization as a scatter plot. The graph size slightly increases only in a few cases while it is dramatically reduced in the most cases. Note that the axes are scaled logarithmically.

Discorrelation between the Number of Checks and the Time needed for Minimization

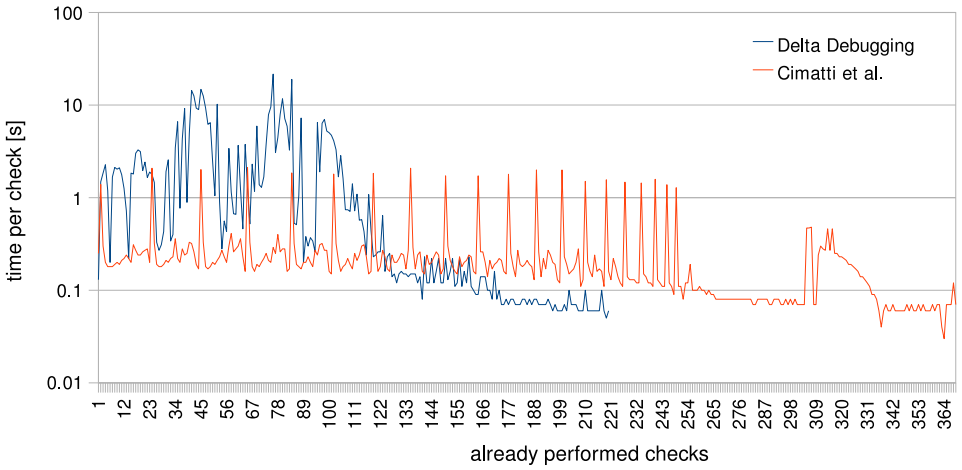
This section analyzes the gap between the reduction in the number of realizability checks and the reduction in the time needed for minimization due to Delta Debugging. This is done by examining the minimization process on the two specification variants `G20wsf1` and `G20wst1`. The first one is examined because for this specification, Delta Debugging needs fewer realizability checks but more time. This is surprising and needs to be investigated. For the second specification mutant, Delta Debugging performs well. This mutant therefore serves as object of comparison in our analysis.

Figure 6.3 gives details to the minimization process in case of `G20wsf1` for both, the Delta Debugging algorithm as well as for the simple minimization algorithm of Cimatti et al. [21]. Figure 6.3a shows how the size of the set of guarantees and output signals decreases with the realizability checks. Figure 6.3b contains the time per realizability check. Note that the ordinate is scaled logarithmically. Figure 6.3c finally illustrates how the size of the set to be minimized decreases with the elapsed time. Figure 6.4 illustrates the same information for the specification `G20wst1`. Two effects can be observed in these figures.

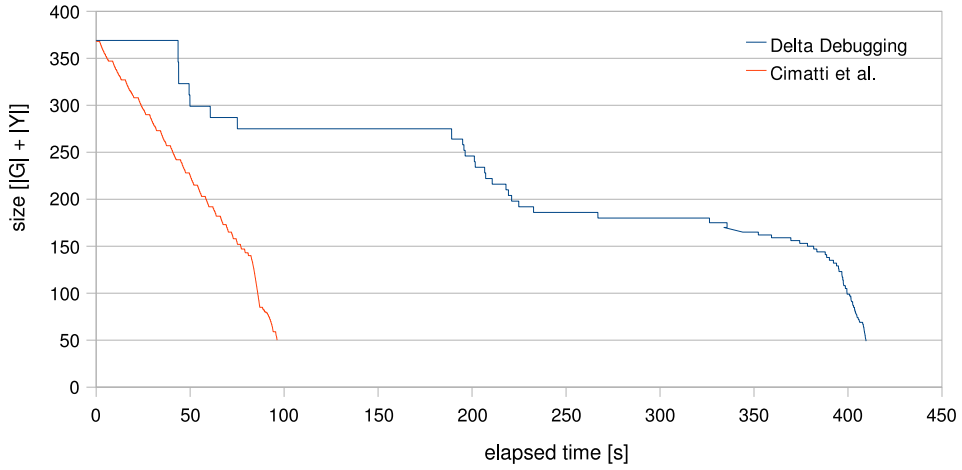
First, the time per realizability check tends to decrease with decreasing size of the specification (see Figure 6.3b and Figure 6.4b), which is not surprising. This explains why the time savings due to Delta Debugging are often slightly higher than the savings in the number of checks. As it can be seen in Figure 6.4a, significant reductions of the size often occur early when using Delta Debugging, so that



(a) The reduction in the size of the set of guarantees and output signals over the already undertaken amount of realizability checks.

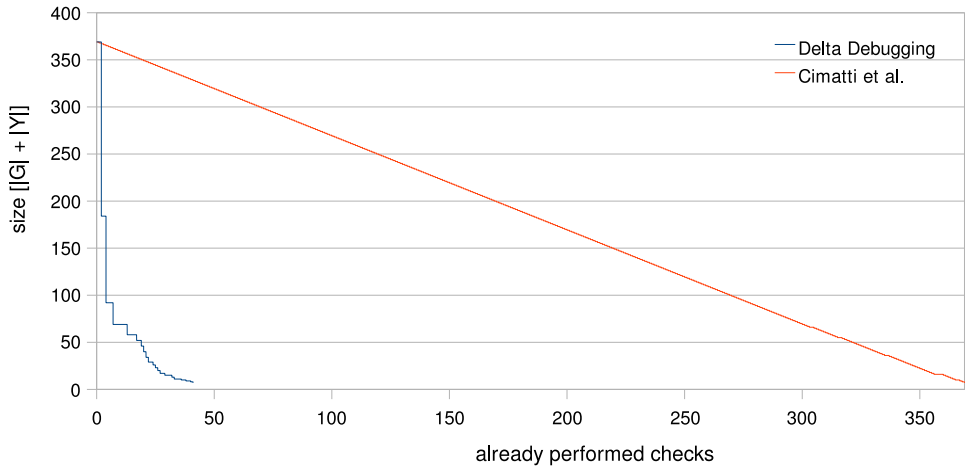


(b) The times needed per realizability check.

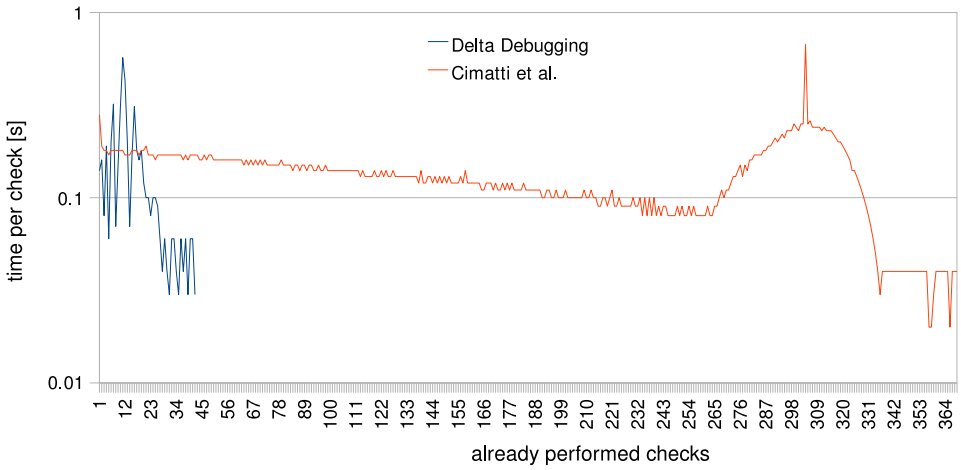


(c) The reduction in the size of the set of guarantees and output signals over the elapsed time.

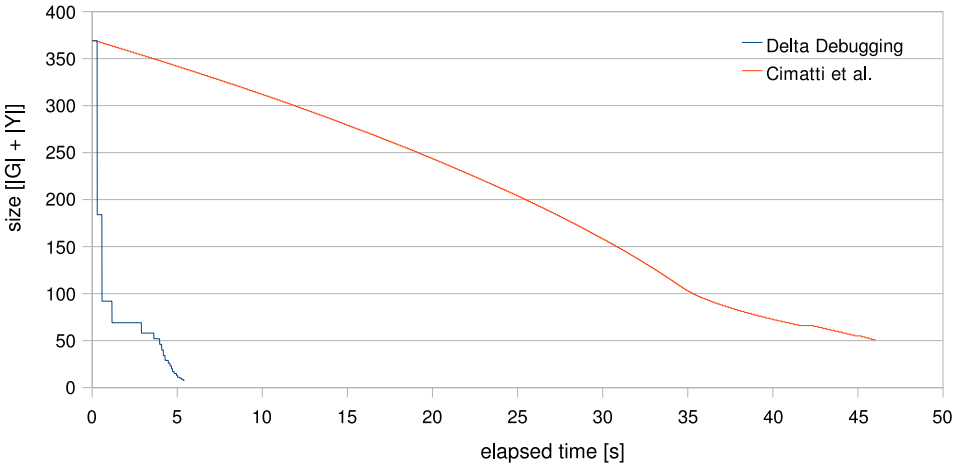
Figure 6.3: Analysis of the minimization process in case of `G20wsf1`.



(a) The reduction in the size of the set of guarantees and output signals over the already undertaken amount of realizability checks.



(b) The times needed per realizability check.



(c) The reduction in the size of the set of guarantees and output signals over the elapsed time.

Figure 6.4: Analysis of the minimization process in case of `G20wst1`.

the algorithm mostly triggers checks on rather small and simple specifications. The correlation between the time per realizability check and the size of the specification does, however, not explain the bad performance of the Delta Debugging algorithm on `G20wsf1`. Figure 6.3a shows that, except for the first 26 checks, the simple minimization algorithm has to deal with larger specifications throughout the minimization process.

Second, the time needed for one realizability check is often significantly higher when the specification is realizable, compared to the cases where the specification is unrealizable. Every spike in Figure 6.3a corresponds to a case where the examined specification is realizable. When the simple minimization algorithm is applied to `G20wsf1`, only 50 of the 369 checks give the verdict of having a realizable specification. In case of Delta Debugging, 134 of the 220 checks are performed on a realizable specification. That is, although the overall number of checks is smaller when using Delta Debugging, the amount of checks on realizable specifications is higher. Considering the higher time needed for checking a realizable specification, this explains the higher computation time needed by Delta Debugging in case of `G20wsf1`.

Performing a realizability check on an unrealizable specification is often faster because of the performance improvement (see Section 4.2) applied to our implementation: The algorithm computing the winning region W_{sys} for the system aborts, signaling that the specification is unrealizable, if the initial state is not contained in some iterate of the outermost fixpoint. If the specification is realizable, the computation of the winning region has to be performed completely. Unfortunately, this situation would not change if realizability was decided by computing the winning region W_{env} of the environment and checking if $q_0 \notin W_{\text{env}}$ instead of checking if $q_0 \in W_{\text{sys}}$. Just as the computation of W_{sys}, the computation of W_{env} could only be aborted before completion in case of unrealizability, but not in case of realizability.

The simple minimization algorithm requires exactly $|\hat{G} \cup \hat{Y}|$ checks on realizable specifications. No minimization algorithm can require fewer checks on realizable specifications, since the minimality of the solution can only be guaranteed if there has been an attempt to take every single element out of the set. In fact, given an unrealizable core, its minimality cannot even be verified with fewer than $|\hat{G} \cup \hat{Y}|$ checks on realizable specifications. Hence, Delta Debugging cannot outperform the simple minimization algorithm regarding the amount of the more expensive checks on realizable specifications. Delta Debugging can only outperform the simple minimization algorithm by reducing the overall number of checks.

For `G20wst1`, Delta Debugging encounters much more unrealizable intermediate specifications than the simple minimization algorithm as well. However, Delta Debugging is able to remove big parts of the specification early (see Figure 6.4a and Figure 6.4c), thus requiring only a few checks for realizability at all. This compensates the higher amount of the (often more expensive) checks on realizable specifications. Also, the differences between the times needed for testing realizable and unrealizable specifications are quite low for `G20wst1`. A manual inspection of some other mutants suggests that this is also the case for many other specification variants. The big differences between the times for checking realizable and unrealizable specification that occur in the specification variants `Gnwsf1` seem to be the exception.

6.2 Evaluation of the given Explanations

This section evaluates the usefulness of the diagnostic information given by our tool. It investigates an unrealizable specification as well as a specification that is in conflict with the design intent. It illustrates that the given explanations are indeed easy to understand and of great help to find fixes, even for industrial-size specifications.

6.2.1 Debugging Unrealizability

We investigate the explanations given for unrealizability by debugging the unrealizable specification `G5wst2`. This specification defines a generalized buffer, which is used by 5 senders and 2 receivers. Some signals and properties of the specification, which are relevant for this example, will now be explained briefly. The inputs will be written with lower case letters and the outputs will be typed with upper case letters in the following. With the inputs `stob_reqi`, where $0 \leq i < 5$, the senders can signal a request to send data. The outputs `BTOS_ACKi`, again with $0 \leq i < 5$, are used to acknowledge to the senders that they can send data. The generalized buffer does not only communicate with the senders, but also with a FIFO (First In, First Out) storage unit. The output `ENQ` is set whenever data should be enqueued in this storage unit. The specification `G5wst2` contains an additional guarantee $\mathsf{G}(\texttt{ENQ=0})$ which forbids the buffer to put any data into this storage unit. This guarantee makes the specification unrealizable, as the buffer is no longer able to handle requests of senders.

SAT-check and Minimization

In our approach (see Figure 3.2), a check for satisfiability is performed first. It shows that the unrealizable specification `G5wst2` is still satisfiable. Thus, we cannot use SAT-solving techniques to explain the problem. Next, the specification is minimized in order to obtain an unrealizable core. From the 67 formulas specifying the system, the following remain:

$$\texttt{BTOS\_ACK4=0} \wedge \texttt{ENQ=0} \wedge \texttt{DEQ=0} \tag{6.1}$$

$$\mathsf{G}((\texttt{BTOS\_ACK4=0} \wedge \mathsf{X}\,\texttt{BTOS\_ACK4=1}) \Rightarrow \mathsf{X}\,\texttt{ENQ=1}) \tag{6.2}$$

$$\mathsf{G}(\texttt{ENQ=0}) \tag{6.3}$$

$$\mathsf{G}\,\mathsf{F}(\texttt{stob\_req4=1} \Leftrightarrow \texttt{BTOS\_ACK4=1}) \tag{6.4}$$

$$\mathsf{G}((\texttt{rtob\_ack0=1} \wedge \mathsf{X}\,\texttt{rtob\_ack0=0}) \Rightarrow \mathsf{X}\,\texttt{DEQ=1}) \tag{6.5}$$

$$\mathsf{G}((\texttt{rtob\_ack1=1} \wedge \mathsf{X}\,\texttt{rtob\_ack1=0}) \Rightarrow \mathsf{X}\,\texttt{DEQ=1}) \tag{6.6}$$

$$\mathsf{G}(\texttt{empty=1} \Rightarrow \texttt{DEQ=0}) \tag{6.7}$$

Furthermore, the minimization algorithm keeps only 3 of the 15 output signals, namely

`BTOS_ACK4`, `ENQ`, and `DEQ`.

All other formulas specifying the behavior of the system, and all other outputs, are irrelevant for the unrealizability problem. That is, even if the specification would require the system to fulfill only the enumerated guarantees with respect to the listed outputs, the system would not be able to behave conforming to this specification.

The minimization result is so simple that the problem can be seen at one glance: The output signal `BTOS_ACK4` is initially set to 0 (Equation 6.1). It cannot change its value from 0 to 1 without `ENQ` being set (Equation 6.2). The output `ENQ` must not be set ever (Equation 6.3, which was added to make the specification unrealizable), so `BTOS_ACK4` can never be set, i.e., Sender 4 will never receive an acknowledgement to be allowed to send data. When Sender 4 forever requests to send data by setting `stob_req4=1`, the guarantee in Equation 6.4 cannot be fulfilled. This guarantee states that every request of Sender 4 to send data must finally be acknowledged by the generalized buffer.

The guarantees stated in the Equations 6.5, 6.6, and 6.7 are part of the unrealizable core, because they are necessary for the environment to fulfill all assumptions. Without these guarantees, the system could enforce a situation from which the environment could not meet all assumptions. We omit the details of how the system could do so, because this information would also be of little interest for the user, who only wants to find out why the specification is unrealizable.

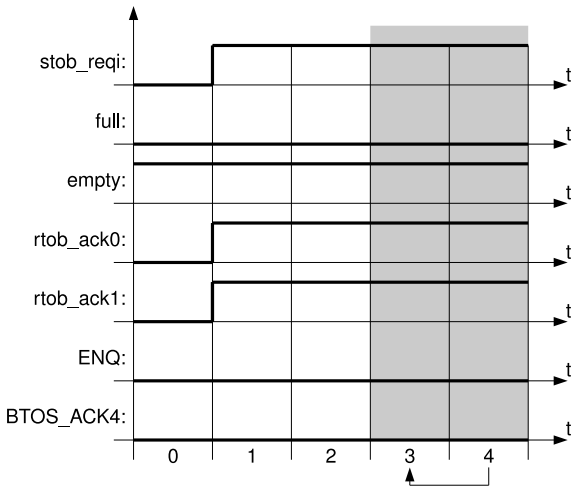


Figure 6.5: The countertrace for the specification `G5wst2`. The infinite loop is marked in gray. The traces for `stob_req0` to `stob_req4` are equal and thus printed only once in the waveform labeled with `stob_reqi`. The only possible value sequences for the output signals `ENQ` and `BTOS_ACK4` are included in this figure as well.

By now, we can also comprehend why the specification is still satisfiable. Remember that a specification is satisfiable if one trace of inputs and outputs exists, so that the specification is fulfilled. In our example, we have such traces. One possibility is that none of the senders requests to send data. In this case, the system does not have to give any acknowledgements.

Counterstrategy and Countertrace

Next, following our debugging approach, a counterstrategy is computed. This counterstrategy is then used to heuristically search for a countertrace. Our heuristic is able to find a countertrace which exploits the discussed problem. It is depicted in Figure 6.5 together with the only possible trace (conforming to the safety guarantees in Equations 6.1, 6.2, and 6.3) for the output signals `ENQ` and `BTOS_ACK4`. The infinite loop is marked with gray background. The relevant part of the countertrace is that the input `stob_req4` is set to 1 forever. To comply with the safety constraints, the system must set `ENQ=0` and `BTOS_ACK4=0` forever. The fairness constraint stated in Equation 6.4 is not fulfilled.

Interactive Game and Summarizing Graph

In order to explore possible responses to the countertrace, the user can play the interactive game against the environment. Figure 6.6 shows the graph $\mathbb{G}$ that summarizes all plays which are possible in this game when the environment uses the countertrace of Figure 6.5. Refer to Section 5.2.3 for a detailed description of the elements of the graph.

The graph is very simple. There is no way for the user to prevent the play from getting stuck in `S2` and `S3`. The states of these vertices are not part of the set J_0^s of accepting state of the system[1]. The set J_0^s represents all states in which the condition of Equation 6.4 is fulfilled. Hence, there is no way to fulfill this guarantee when the countertrace of Figure 6.5 is used as input.

[1] Remember that the indices start at 0 in our implementation while they start at 1 in our theoretical framework.

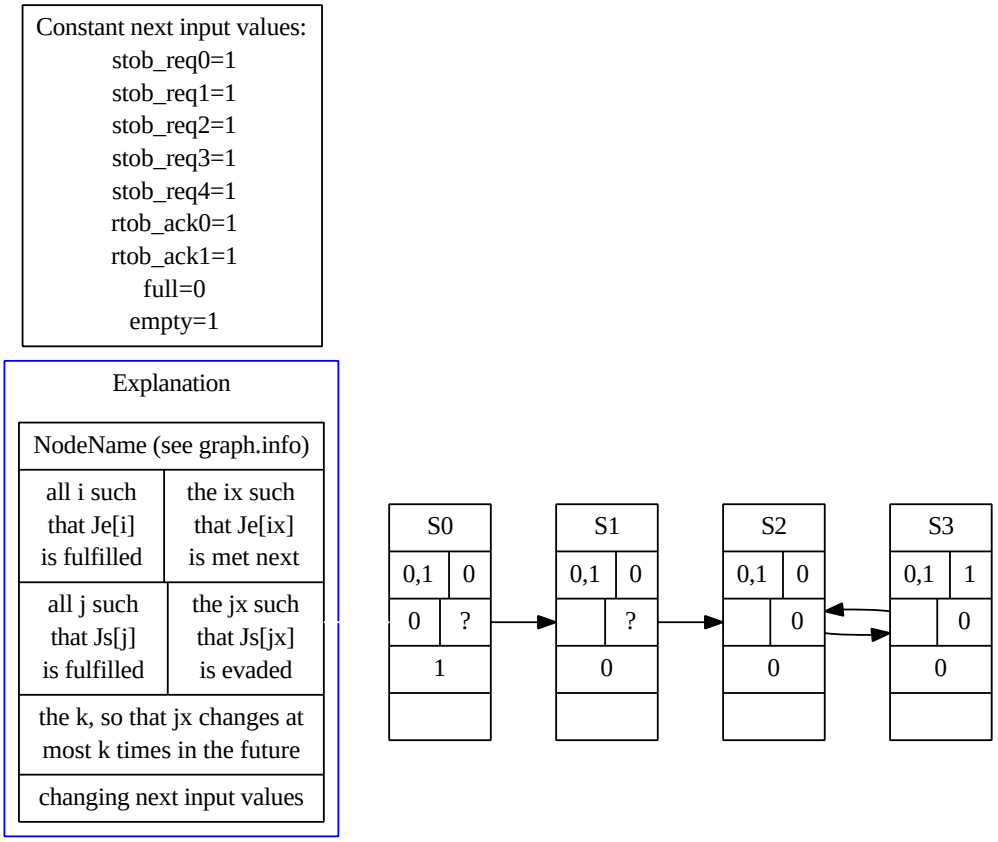


Figure 6.6: The graph $\mathbb{G}$ computed for the specification `G5wst2` after minimization. It summarizes all plays that are possible when the environment uses the countertrace. Vertices correspond to situations in the game, edges to choices of the user. All plays start in `S0`. Refer to Section 5.2.3 for a more detailed description.

Solution

Once the user has understood the conflict that makes the specification unrealizable, it is up to her to resolve it. There are multiple ways of doing so. The user could add environment assumptions that prevent the requests from the senders to be pending indefinitely. Another possibility would be to remove one of the guarantees which are remaining after minimization. (Remember that we added the guarantee stated in Equation 6.3 in order to make the specification unrealizable.) Selecting the best solution is up to the user.

6.2.2 Debugging Undesired Behavior

In this section, our approach for debugging undesired behavior will be investigated on an industrial-size example. This is done by debugging the unmodified bus arbiter specification `A2` for two masters with respect to a fictive design intent. Again, we use lower case letters for input signals and upper case letters for output signals. The relevant signals for this example are briefly introduced as follows. The output signal `HMASTER` states which master currently owns the bus. It is set to 0 whenever the bus is owned by Master 0, and set to 1 whenever the bus is occupied by Master 1. The output signal `START` must be raised when the bus ownership changes. The inputs `hbusreq0` and `hbusreq1` are used to request the bus by Master 0 and Master 1, respectively.

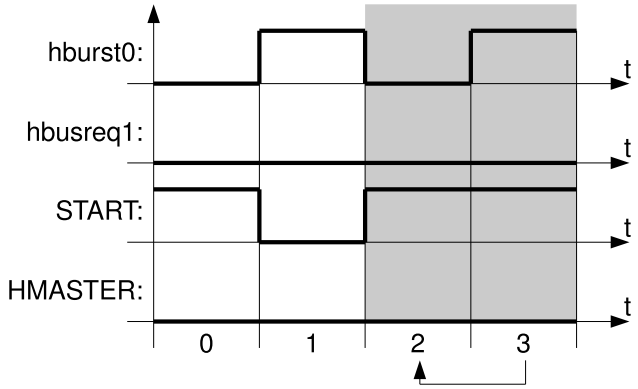


Figure 6.7: A possible simulation trace of the unmodified bus arbiter specification `A2`. It is restricted to signals that are relevant for this example. The infinite loop is colored in gray.

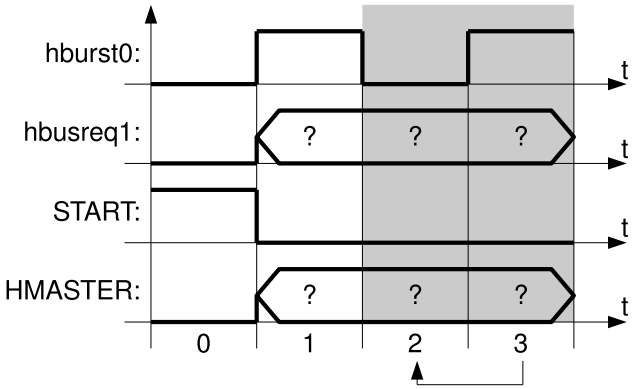


Figure 6.8: The desired behavior as specified by the user: Start must be 0 as long as the input keeps on changing. Question marks stand for the special value "don't care". The infinite loop is colored in gray.

Excluding Undesired Behavior

Figure 6.7 depicts a possible simulation run. It contains only signals which are of interest for this example. The time steps in the infinite loop have a gray background. Suppose the design intent was that `START=0` as long as the input `hburst0` keeps on changing. This behavior cannot be observed in the simulation run shown in Figure 6.7. In such a situation, our debugging approach (see Figure 3.7) requires the user to specify the design intent. This is done by changing signal values in the simulation trace to the desired values. Figure 6.8 shows the result. This trace, specifying the desired behavior, is turned into a guarantee by the tool. The guarantee is then added to the specification. The result is the specification `A2wst1`, which is unrealizable. This means that no system can implement the original specification and at the same time show the desired behavior of Figure 6.8. The desired behavior is in conflict with the rest of the specification.

SAT-check and Minimization

In order to explain this conflict, our approach proceeds as depicted in Figure 3.2. The SAT-check reports that the specification is satisfiable. As the next step, an unrealizable core is computed. From the 65 formulas that specify the system, only the following ones are in conflict with the design intent of

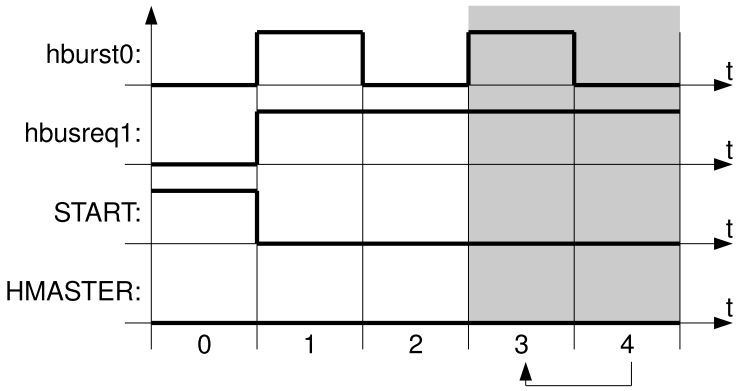


Figure 6.9: The countertrace for the specification `A2wst1`. It keeps `hburst0` changing and sets `hbusreq1=1` forever. The only possible value sequences for the output signals `START` and `HMASTER` are included in this figure as well. The infinite loop is marked in gray.

Figure 6.8:

$$\texttt{A0=0} \wedge \texttt{A1=0} \wedge \texttt{START=1} \wedge \texttt{HMLOCK=0} \wedge \texttt{HMASTER=0} \tag{6.8}$$

$$\mathsf{G}((\texttt{A0=0} \wedge \texttt{A1=0} \wedge (\texttt{HMLOCK=0} \vee \texttt{hburst0=1} \vee \texttt{hburst1=1})) \Rightarrow \mathsf{X}(\texttt{A0=0} \wedge \texttt{A1=0})) \tag{6.9}$$

$$\mathsf{G}((\mathsf{X}\,\texttt{START=0}) \Rightarrow (\texttt{HMASTER=1} \Leftrightarrow \mathsf{X}\,\texttt{HMASTER=1})) \tag{6.10}$$

$$\mathsf{G}\,\mathsf{F}(\texttt{HMASTER=1} \vee \texttt{hbusreq1=0}) \tag{6.11}$$

From the 15 output signals, only the 5 signals

`HMASTER`, `HMLOCK`, `START`, `A0`, and `A1`

are involved in the conflict with the design intent. Analyzing this minimization result, the conflict can be explained as following. With `HMASTER=0`, the bus is initially granted to Master 0 (Equation 6.8). According to the design intent specified in Figure 6.8, the output `START` cannot be raised as long as the input `hburst0` keeps on changing. The guarantee in Equation 6.10 ensures that the bus ownership cannot change without `START` being raised. Hence, the bus will remain granted to Master 0 as long as `hburst0` keeps on changing. When the bus is additionally requested by Master 1 (`hbusreq1=1`) forever, the guarantee stated in Equation 6.11 is not fulfilled. This guarantee affirms that every request of Master 1 for the bus is finally acknowledged.

Equation 6.9 is necessary for the specification to be unrealizable, because there is an environment assumption $\mathsf{G}\,\mathsf{F}(\texttt{A0=0} \wedge \texttt{A1=0})$. This assumption requires the environment to visit the state encoded with `A0=0` and `A1=0` infinitely often. The environment needs Equation 6.9 in order to be able to enforce that.

Counterstrategy and Countertrace

The tool computes a counterstrategy and therefrom a countertrace. The resulting countertrace is illustrated in Figure 6.9. It again contains only the inputs that are relevant for this example. It exploits the already discussed problem by keeping the input `hburst0` changing, and by setting the input `hbusreq1=1` forever. Figure 6.9 also shows the only sequence of the outputs `START` and `HMASTER` which fulfills the remaining safety guarantees, i.e., the Equations 6.8 to 6.10. The fairness guarantee of Equation 6.11 is not fulfilled.

Interactive Game and Summarizing Graph

If neither the minimization result nor the countertrace make the conflict between the specification and the design intent clear for the user, the interactive game can be played. Figure 6.10 shows the graph that summarizes all plays that are possible when the environment uses the countertrace. It is organized as described in Section 5.2.3. There is only one set J_0^s of accepting states for the system. This set contains all states that fulfill the condition in Equation 6.11. There are two sets J_0^e and J_1^e of accepting states of the environment. After two steps, the play is in one of the vertices `S2` to `S5` of the graph. This set of vertices cannot be left any more. The states in this set are all elements of J_0^e and J_1^e, but no elements of J_0^s. Hence, all assumptions are fulfilled but the system cannot fulfill all guarantees when the environment uses the countertrace.

Solution

Resolving the conflict is again up to the user. The user might allow the bus ownership to change without raising the signal `START`, she might remove the guarantee that every request for the bus is finally acknowledged, etc. Which solution is most suitable cannot be decided automatically.

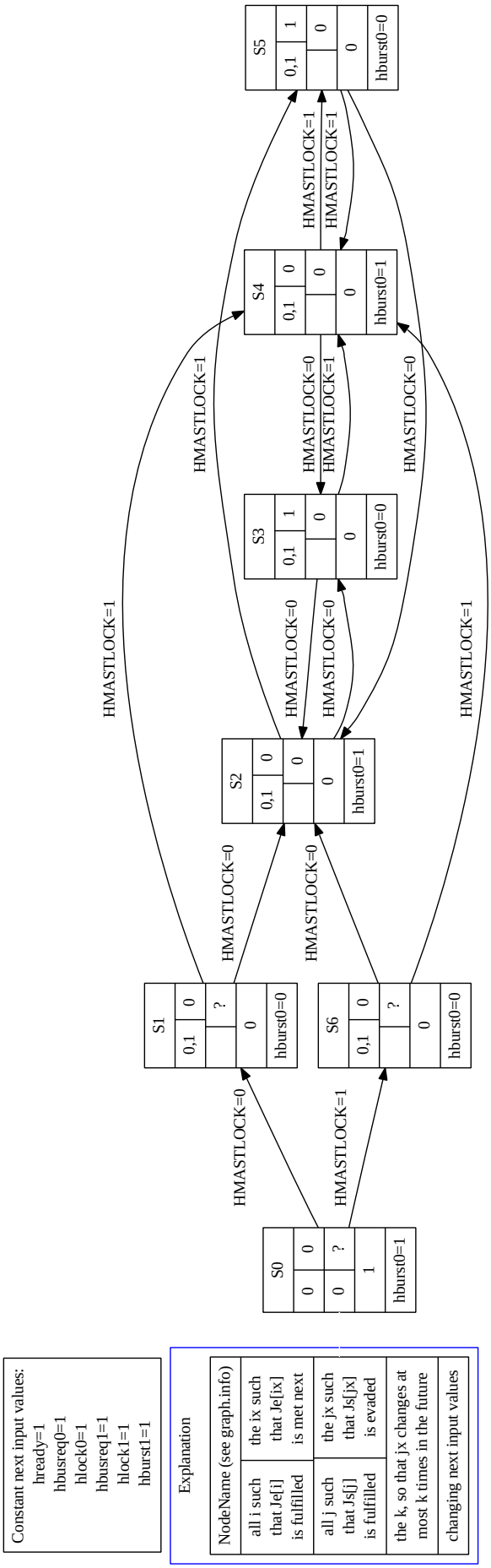


Figure 6.10: The graph $\mathbb{G}$ computed for the specification `A2wst1` after minimization. It summarizes all plays that are possible when the environment uses the countertrace. Vertices correspond to situations in the game, edges to choices of the user. All plays start in `S0`. Refer to Section 5.2.3 for a more detailed description.

7 Related Work

7.1 Debugging Incomplete Specifications

The matters of incomplete specifications and coverage measures to detect incompleteness have been addressed before in various ways.

On the one hand, there is work that checks the completeness of a specification regarding a given implementation. Katz et al. [60] define different comparison criteria, each revealing a certain dissimilarity between the implementation and the specification. The criteria are based on comparisons between the implementation and the tableau of the specification. Hoskote et al. [56] propose a coverage metric that identifies the part of the state space of the implementation that is covered by the specification. In this metric, a state is covered with respect to a certain signal if modifying the value of the signal in that state violates the specification. The idea of introducing modifications into the model of the implementation and checking if they violate the specification is also used by Chockler et al. [17; 18]. All these papers are strongly motivated by model-checking as there is always an implementation available in such a setting.

On the other hand, there are coverage measures that do not rely on any particular implementation. Claessen [23] introduces a notion of "forgotten cases", which are situations where a certain output at a certain point in time is not constrained. They also define a concept of "freeness" that allows the user to distinguish between intentionally and unintentionally underspecified signals. The specification is assumed to be given as a list of safety properties. Fisman et al. [45] try to detect what they call "inherent vacuity". They define a specification to be inherently vacuous if it can be mutated into a simpler equivalent specification. This is in turn the case if the specification is satisfied vacuously in all systems. The method works for sets of linear temporal properties. Das et al. [34] propose to test the specification against a high-level fault model. They use a single stuck-at fault model on inputs and outputs. The idea is to check whether an implementation that contains a stuck-at fault can still conform to the specification. If so, then the signal that is allowed to be stuck is underconstrained.

The focus of our work is not on detecting incompleteness in a specification. We only show how a specification that is incomplete can be distinguished from a specification that is in conflict with the design intent, given a situation where undesired behavior has been observed during the simulation of a system. In addition, we show how the specification can be refined in case of incompleteness.

7.2 Debugging Specifications which are not Sound

7.2.1 Counterstrategies as Debugging Aids

The idea of using counterstrategies as aids for debugging unrealizable specifications has already been used in various settings.

Tripakis et al. [96] consider game graphs with controllable and uncontrollable edges with respect to invariance and reachability properties. They attempt to compute a strategy and a counterstrategy simultaneously, mentioning that the counterstrategy can be used as diagnostics if no winning strategy could be found.

Bontemps et al. [10] present an algorithm to compute a counterstrategy for an unrealizable specification of a reactive system that is given as a Live Sequence Chart [32]. They also mention that a user could utilize this counterstrategy in a play-out engine to illustrate the flaws in the specification.

Behrmann et al. [5] present the tool UPPAAL-Tiga, which works with a network of timed automata that defines a game. The objective in the game can be given as a safety property or as a liveness property. If no controller can achieve the objective, a counterstrategy is computed. The counterstrategy can be output as a decision graph, and it can be used in a game against the user.

Stevens and Stirling [92; 91] consider the problem of model-checking the modal μ-calculus. If the μ-calculus formula does not hold, they use a counterstrategy to explain the reason. They assume that the user has some path through the system in his mind that makes the formula hold. Just like we do, they use the counterstrategy in an interactive game to demonstrate that the imagined path does not fulfill the formula. Counterstrategy computation as well as the interactive game have been implemented in the Edinburg Concurrency Workbench [74]. Even some usability issues are addressed, but they are all related to the (textual) user interface. Leucker and Noll extend the work of Stevens and Stirling by presenting a more efficient algorithm [70] and by integrating it into the tool Truth/SLC [71], where the diagnostic game can be played utilizing a graphical user interface. Tan [93] introduces another tool called PlayGame, implementing the same approach. The core feature of PlayGame compared to Truth/SLC is that it implements diagnostic games independent of a specific model checker.

Although all of these papers at least mention the use of counterstrategies for diagnostic purposes, none addresses the simplification of a counterstrategy in order to present helpful information to the user. This simplification is the main contribution of our work. In particular, we are not aware of any previous work on finding countertraces. Finally, the use of counterstrategies to explain conflicts between the formal specification and the informal design intent is also new to the best of our knowledge.

7.2.2 Other Debugging Techniques

Yoshiura [100] addresses the problem of explaining unrealizability. He proposes several heuristics for the localization of the cause of unrealizability in a temporal specification of a reactive system. The heuristics are based on the tableau of the specification and on a classification of the specification regarding three properties a specification might have. These three properties are *strong satisfiability*, *stepwise satisfiability*, and *stepwise strong satisfiability*, as introduced by Mori and Yoshiura [75].

Cimatti et al. [21] suggest to explain unrealizability by presenting an unrealizable core. They also state that such an unrealizable core can be used to obtain a more focused counterstrategy. To the best of our knowledge, this is the only work that is concerned with giving *simple* explanations. Our work utilizes their ideas with severals improvements. First, we do not minimize environment assumptions because this step is computationally expensive and because it would confuse the user in the game. Second, we remove not only properties but also signals from the specification. Third, we use a more advanced minimization algorithm that is faster in most of our experiments.

8 Conclusion and Outlook

8.1 Summary

In this work, we presented aids for debugging specifications that are unrealizable or in conflict with the design intent. We introduced a generic approach and concretized it for the class of GR(1) specifications. We presented an implementation of our method for this class of specifications and finally provided experimental results.

8.1.1 Debugging Approach

In our approach for debugging conflicts with the design intent, the user has to specify the desired behavior by modifying a simulation trace that surfaces undesired behavior. The tool then augments the specification with a guarantee that enforces the desired behavior. If the augmented specification is realizable, the original specification has been incomplete and the augmented specification is a valid fix. If the augmented specification is unrealizable, no system can implement the specification and at the same time exhibit the desired behavior. The desired behavior is in conflict with the rest of the specification. Explaining this conflict is done by explaining the unrealizability of the augmented specification.

Our approach for explaining unrealizability is based on the presentation of a counterstrategy. When the environment adheres to it, no behavior of the system can fulfill the specification. The counterstrategy is presented in two ways. First, the user can play an interactive game in the role of the system against the counterstrategy. The goal of the user in this game is to fulfill the specification. Failing to do so, she will learn where the specification is too restrictive to be realizable. Second, the counterstrategy is presented as a graph that summarizes all plays that are possible in the interactive game when the environment adheres the counterstrategy.

8.1.2 Simplification of Counterstrategies

Counterstrategies may become complex and thus hard to understand for the user, so we presented two methods to simplify them. These simplification methods are the main contribution of this work.

First, we simplify the specification itself, by computing an unrealizable core as suggested by Cimatti et al. [21]. This gives a simpler specification that still contains the conflict. We improve the work of Cimatti et al. in three points: (1) we also minimize the output signals, (2) we do not minimize environment assumptions as this step is computationally expensive, and (3) we use Delta Debugging as a (more advanced) minimization algorithm.

Second, we suggest to present a countertrace instead of a counterstrategy. The inputs dictated by the counterstrategy depend on the previous output values. To the contrary, a countertrace is a *fixed* sequence of inputs so that no behavior of the system can fulfill the specification. A countertrace does not always exist. Additionally, its computation is expensive. We therefore presented a heuristic algorithm that searches for a countertrace. It does not always find a countertrace, even if one exists, but it performs well in our experiments.

8.2 Discussion

8.2.1 Debugging Undesired Behavior

Our approach and the corresponding implementation to debug undesired behavior let the user eliminate faulty behavior of the system with a high usability. The feature that the user can simply change signal

values in the faulty trace obtained during simulation in order to define the desired behavior is very convenient. It is easy to use even for non-experts and for users which are not familiar with the underlying specification language. As a shortcoming, one could mention that the manual inspection and correction of simulation traces becomes the more laborious the larger the specification is. For specifications with hundreds or thousands of signals, one would have to think of ways to automate parts of these processes.

8.2.2 Countertraces

Judging from experience, we consider the countertraces as the most useful information given by our tool in order to make the user understand conflicts within the specification or between the specification and the design intent. Countertraces are independent of the system's moves and thus much easier to understand than conventional counterstrategies. Additionally, the user knows in advance how the environment will behave in the interactive game when given the countertrace used by the environment. This makes it easier for the user to localize the problem. Sometimes, a glimpse onto the countertrace is enough to understand the problem, and the interactive game even does not have to be played at all. Think of a situation where the user has simply forgotten to exclude the environment behavior dictated by the countertrace with environment assumptions, for example. The CPU time needed for the computation of a countertrace is negligible when using our heuristic. Thus, countertraces give simpler explanations at low costs in many cases. To the best of our knowledge, countertraces have not been mentioned before in the literature.

8.2.3 Summarizing Graphs

The graph that summarizes all possible plays of the interactive game is useful if no countertrace could be found. In such a situation, this graph allows the user to see already in advance how the environment will react to her choices on the output values. In a sense, it is therefore something like a "cheat sheet" for the interactive game. The graph does not provide more information than the interactive game, but it provides information in a more dense form by showing all possible plays simultaneously. As a downside, it can become huge and thus impossible to analyze by the user.

8.2.4 Minimization

Minimization greatly reduces not only the size of the specification but also the complexity of the corresponding game structure. In order to achieve a simple game structure, it is important that not only guarantees but also output signals are removed. Carried out this way, minimization decreases the size of the graph $\mathbb{G}$, which summarizes all possible plays, significantly. Remember that the size of this graph is also a good indicator of how simple it is for the user to understand the conflict when playing the interactive game. Furthermore, minimization increases the chance to find a countertrace with our heuristic. The amount of time needed to perform the minimization step is often much lower than the time needed for the computation of a counterstrategy for the original specification. The times needed for all subsequent steps are negligible when minimization has been performed. Hence, minimization does not only lead to simpler explanations, it often even speeds up the whole computation.

8.3 Future Work

Although we think that we have defined a quite useful approach for debugging unrealizability and conflicts with design intents, there is still a lot of work to be done in the future.

8.3.1 Evaluation

In this work, the introduced debugging approach has been concretized and evaluated for the class of GR(1) specifications only. This could be done for other kinds of specification languages as well. It would be interesting to see for which other languages the approach works better or worse.

In Section 6, we evaluated our debugging approach on artificially constructed bugs, inserted into two parametrized GR(1) specifications. More significant results would be obtained if the evaluation was done on bugs that occurred in real specification development processes. This would require some sort of database of buggy specifications. The expressiveness of the evaluation could also be increased if further specifications were used.

8.3.2 Graph Computation

As can be seen from Table 6.2, a lot of time is often spent on the computation of the graph $\mathbb{G}$ until this computation aborts as the number of graph nodes exceeds some threshold value. To overcome that, the graph nodes could be computed symbolically in a first step. Remember that the graph nodes are simply the pairs of state and memory content that might occur during a play when the environment adheres to the counterstrategy. Thus, the symbolic computation of the graph can be done similar to Listing 4.4, in which the set of reachable states is computed symbolically. Checking if the number of graph nodes is larger than the threshold value could be done using the symbolic representation of the set of graph nodes. Finally, the computation of the explicit representation of the graph $\mathbb{G}$ is only performed if the graph is not too large. The symbolic computation of the set of graph nodes is likely to be very fast compared to an explicit computation. Hence, the high graph computation times could be avoided in cases where this computation is aborted anyway.

The graph $\mathbb{G}$ often contains nodes which are equivalent in the sense that exactly the same moves are possible from these nodes. For example, in the graph depicted in Figure 6.10, `S1` is equivalent to `S6`, `S2` is equivalent to `S4`, and `S3` is equivalent to `S5` in this sense. Such equivalent nodes could be merged into one node. This would reduce the size of the graph, making it easier for the user to analyze it. On the other hand, one graph node would then no longer represent one state-memory pair of the play, but a set of such pairs. Hence, the direct connection between a graph node and the states in the automata of the game would be lost. Thus, computing the graphs in two versions, one in which equivalent nodes are merged and one in which they are not, would probably be the best solution.

8.3.3 Countertraces

The countertraces produced by our heuristic are often not as short in their representation as they could be. For instance, in Figure 6.5, the steps 1, 2, 3, and 4 contain exactly the same values for all inputs. The steps 1 to 3 could be removed to obtain a shorter representation of the same trace. As a general solution, the infinite loop could be analyzed for patterns that repeat, at first. After the repetitions have been removed in the loop, all occurrences of the loop pattern at the end of the finite stem could be removed from the stem. This gives a shorter representation of the same countertrace.

The heuristic that searches for countertraces itself could be improved as well. Currently, an input letter τ_i is chosen arbitrarily from the set T_i. If $T_i = \emptyset$ for some i, our heuristic aborts without success. The sets S_i and T_i depend on all previously chosen input letters τ_j with $j < i$. Hence, when encountering an empty set T_i in some step i, the heuristic could pick a different input letter $\tau_j \in T_j$ in some previous time step j, in the hope that this letter does not lead to an empty set. In the question of how often the heuristic should pick a different τ_j until it aborts without success, one would have to find a trade-off between the computation time and the success rate. Another direction into which one could go in order to improve the heuristic would be to pick the input letters $\tau_i \in T_i$ not arbitrarily but following some metric that reflects the chances for success. A first idea would be to choose τ_i such that S_{i+1} has the lowest possible cardinality.

If no countertrace could be found from the initial state, the tool could use a combination of a counterstrategy with countertraces during the interactive game. It could start the game using a counterstrategy. From every state-memory pair that is encountered during a play, it could then on-the-fly try to find a countertrace using our heuristic. If a countertrace is found from a certain situation, this countertrace is used instead of the counterstrategy from that point on.

8.3.4 Minimization

The minimization step would need further investigation concerning its performance as well. In the current version of the implementation, output signals are minimized simultaneously with guarantees. Another possibility would be to minimize only the guarantees in a first step, and the output signals in a second step, or vice versa. Maybe this improves the performance in the average case.

It could also improve the performance if realizability was decided by checking if $q_0 \notin W_{\text{env}}$ instead of checking if $q_0 \in W_{\text{sys}}$. Both methods are equivalent since W_{env} and W_{sys} are complementary sets of states, i.e., $W_{\text{env}} = Q \setminus W_{\text{sys}}$. An optimization similar to the one mentioned in Section 4.2 could be applied: the computation of W_{env} could be aborted, signaling that the specification is unrealizable, as soon as $q_0 \in Z_a$ for some iterate Z_a of the outermost fixpoint according to Equation 4.1. It is not to be expected that the optimization has a greater impact on the performance in case of computing W_{env} but maybe the performance increases for some other reason.

A maybe more promising approach to reduce the effort for deciding realizability would be to define criteria that are on the one hand sufficient for a specification to be unrealizable or realizable, and on the other hand easy to check. Realizability would then have to be computed only if none of the criteria applies. The challenge is to find criteria that apply to as many cases as possible while still being efficient to check.

As it can be seen from the Tables 6.3 and 6.4, Delta Debugging performs much better than the simple minimization algorithm of Cimatti et al. [21] in most cases. However, there are also cases in which Delta Debugging is worse. One could try to define a heuristic that picks one of the algorithms based on some properties of the specification which are easy to observe. One could also try to combine the algorithms, e.g, by starting with Delta Debugging and switching to the simple minimization algorithm if the performance of Delta Debugging goes below a certain mean reduction-per-check value. Finally, other minimization algorithms could be tried as well.

All in all, we can conclude that writing correct formal specifications is hard, and that it is difficult to provide the user with meaningful diagnostic information in case of an incorrect specification. The debugging approach presented in this work supports the user in the task of localizing bugs in a formal specification, but there is still a lot of work to be done in the future.

Appendices

Appendix A: A session with Marduk

```
1  ------------------- Satisfiability -------------------
2  GF(guarantee[j]) can be satisfied for all j
3  The specification IS satisfiable (took 0.01 seconds)
4  --------------------- Minimization ---------------------
5  Applying DD to find a smaller spec that is still unrealizable ...
6    guarantees: 5 formulas --reduced to--> 3 formulas
7    outputs: 6 formulas --reduced to--> 4 formulas
8    All in all: 11 formulas --reduced to--> 7 formulas
9    83 checks for realizability had to be done
10   64 checks could be omitted, because a superset was already realizable
11   -> only 19 checks were actually carried out (took 0.14 seconds)
12  The minimal spec that is still unrealizable was written to ./delta.xml.
13  A log of what the Delta Debugger did was written to ./delta.log.
14  FOR ALL FURTHER ANALYSIS, I WILL USE THE MINIMIZED SPECIFICATION!
15 ------------------- Counterstrategy -------------------
16 Calculating the winning region for the environment ... (0.01 seconds)
17 Calculating the counterstrategy ... (took 0.01 seconds)
18 --------------------- Countertrace ---------------------
19 Searching heuristically for a countertrace ... (took 0.01 seconds)
20  Countertrace FOUND (length loop: 1   length stem: 4)
21 Printing the trace to 'trace.txt' ...
22  FOR ALL FURTHER ANALYSIS, I WILL USE THIS COUNTERTRACE!!!
23 ------------------- Summarizing Graph -------------------
24 Computing a graph that summarizes all possible plays ... (0.07 seconds)
25 Nr of states in graph: 13
26 Graphs were written to 'graph.dot' and 'graph_with_signals.dot'.
27    You can produce pictures of the graphs by typing for example:
28       'dot -Tpdf -o ./graph.pdf ./graph.dot'
29    Detailed information to the graphs was written to 'graph.info'.
30 ------------------- Interactive Game -------------------
31 Lets play a game. I am the environment and you are the system. I will
32 give you inputs, you have to choose outputs. I will help you by writing
33 possible output values in brackets. A log of all variable values in all
34 time steps will be printed to log.txt after you quit with 'Q'. You will
35 see that I will force you to violate your specification!
36 [some things snipped]
37 current req0 is: 1
38 current req1 is: 1
39 current startup_failed is: 1
40 current grant0 is: 0
41 current error is: 1
42 current di_state0 is: 1
43 current di_state1 is: 0
44 The environment tries to fulfill fairness condition nr 0 next
45 I try to keep the system from fulfilling fairness condition nr: 0
46 I reserve the right to change my opinion on that at most 1 times from
47 now
48 next req0 is: 1
49 next req1 is: 1
50 next startup_failed is: 1
51 enter next grant0 (0,1): q
```

Listing A.1: A session with Marduk using its textual user interface. The output was slightly shortened to fit onto a page.

Bibliography

[1] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1989. (Cited on page 5.)

[2] Rajeev Alur and Thomas A. Henzinger, editors. *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*. Springer, 1996. (Cited on pages 89 and 92.)

[3] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for LTL fragments. In Joseph Halpern, editor, *LICS*, pages 291–302. IEEE Computer Society, 2001. (Cited on page 2.)

[4] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. RuleBase: An industry-oriented formal verification tool. In *DAC*, pages 655–660, 1996. (Cited on page 1.)

[5] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. UPPAAL-Tiga: Time for playing games! In Damm and Hermanns [33], pages 121–125. (Cited on pages 7 and 81.)

[6] Gérard Berry, Hubert Comon, and Alain Finkel, editors. *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001. (Cited on pages 90 and 93.)

[7] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In Rudy Lauwereins and Jan Madsen, editors, *DATE*, pages 1188–1193. ACM, 2007. (Cited on pages 3, 6, 63, 64, 65, 66 and 67.)

[8] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007. (Cited on pages 3, 6 and 63.)

[9] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Efficient decision procedures for model checking of linear time logic properties. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 1999. (Cited on page 1.)

[10] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, 62(2):139–169, 2004. (Cited on pages 3, 7 and 81.)

[11] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. Vis: A system for verification and synthesis. In Alur and Henzinger [2], pages 428–432. (Cited on page 1.)

[12] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. (Cited on pages 1 and 11.)

[13] J. Richard Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the 1960 International Congress of Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford Univ. Press, 1962. (Cited on page 2.)

[14] Julius R. Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, April 1969. (Cited on page 2.)

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, volume 1 of *Wiley Series in Software Design Patterns*. John Wiley & Sons, 1996. (Cited on page 59.)

[16] Roberto Cavada. Model-view-controller and observer patterns for pygtk (version 1.99.0). Available from `http://sourceforge.net/projects/pygtkmvc/files/`. Last visit in October of 2009. (Cited on page 61.)

[17] Hana Chockler, Orna Kupferman, Robert P. Kurshan, and Moshe Y. Vardi. A practical approach to coverage in model checking. In Berry et al. [6], pages 66–78. (Cited on pages 3 and 81.)

[18] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for temporal logic model checking. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 528–542. Springer, 2001. (Cited on pages 3 and 81.)

[19] Alonzo Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians (Stockholm, 1962)*, pages 23–35. Institut Mittag-Leffler, Djursholm, 1963. (Cited on page 2.)

[20] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002. (Cited on page 1.)

[21] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Andrei Tchaltsev. Diagnostic information for realizability. In Francesco Logozzo, Doron Peled, and Lenore D. Zuck, editors, *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2008. (Cited on pages 7, 20, 21, 22, 58, 63, 66, 67, 68, 70, 82, 83 and 86.)

[22] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Stefano Tonetta. Boolean abstraction for temporal logic satisfiability. In Damm and Hermanns [33], pages 532–546. (Cited on page 20.)

[23] Koen Claessen. A coverage analysis for safety property lists. In *FMCAD*, pages 139–145. IEEE Computer Society, 2007. (Cited on pages 3 and 81.)

[24] Edmund M. Clarke. The birth of model checking. In Grumberg and Veith [50], pages 1–26. (Cited on page 1.)

[25] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 1988. (Cited on page 1.)

[26] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. (Cited on page 1.)

[27] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983. (Cited on page 1.)

[28] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986. (Cited on page 1.)

[29] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 1999. (Cited on page 1.)

[30] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. (Cited on page 1.)

[31] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *AADEBUG*, 2000. (Cited on page 17.)

[32] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. (Cited on pages 3 and 81.)

[33] Werner Damm and Holger Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007. (Cited on pages 89, 90, 92 and 93.)

[34] Sayantan Das, Ansuman Banerjee, Prasenjit Basu, Pallab Dasgupta, P. P. Chakrabarti, Chunduri Rama Mohan, and Limor Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In *VLSI Design*, pages 201–206. IEEE Computer Society, 2005. (Cited on pages 3 and 81.)

[35] Pallab Dasgupta. *A Roadmap for Formal Property Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. (Cited on page 9.)

[36] Mark Dowson. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84, 1997. (Cited on page 1.)

[37] Samuel Eilenberg. *Automata, Languages, and Machines*. Academic Press, Inc., Orlando, FL, USA, 1974. (Cited on page 10.)

[38] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990. (Cited on page 1.)

[39] E. Allen Emerson. Model checking and the mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1996. (Cited on pages 13 and 15.)

[40] E. Allen Emerson. The beginning of model checking: A personal perspective. In Grumberg and Veith [50], pages 27–45. (Cited on page 1.)

[41] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *LICS*, pages 267–278. IEEE Computer Society, 1986. (Cited on pages 36 and 37.)

[42] E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, 1984. (Cited on page 2.)

[43] Berndt Farwer. omega-automata. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, *Automata, Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2001. (Cited on page 9.)

[44] Andrea Ferrara, Paolo Liberatore, and Marco Schaerf. Model checking and preprocessing. In Roberto Basili and Maria Teresa Pazienza, editors, *AI*IA*, volume 4733 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2007. (Cited on page 1.)

[45] Dana Fisman, Orna Kupferman, Sarai Sheinvald-Faragy, and Moshe Y. Vardi. A framework for inherent vacuity. In Hana Chockler and Alan J. Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2008. (Cited on pages 3 and 81.)

[46] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers: a survey. *Software Testing, Verification & Reliability*, 19(3):215–261, 2009. (Cited on page 3.)

[47] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995. (Cited on pages 58 and 61.)

[48] Simson Garfinkel. History's worst software bugs. *Byte Magazine*, November 2005. Available from `http://www.wired.com/software/coolapps/news/2005/11/69355` (Last visit in October of 2009). (Cited on page 1.)

[49] Karin Greimel, Roderick Bloem, Barbara Jobstmann, and Moshe Y. Vardi. Open implication. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 361–372. Springer, 2008. (Cited on page 5.)

[50] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008. (Cited on pages 90, 91 and 95.)

[51] Sankar Gurumurthy, Orna Kupferman, Fabio Somenzi, and Moshe Y. Vardi. On complementing nondeterministic Büchi automata. In Daniel Geist and Enrico Tronci, editors, *CHARME*, volume 2860 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2003. (Cited on page 2.)

[52] Ronald H. Hardin, Zvi Har'El, and Robert P. Kurshan. Cospan. In Alur and Henzinger [2], pages 423–427. (Cited on page 1.)

[53] David Harel and Amir Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985. (Cited on page 4.)

[54] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. (Cited on page 1.)

[55] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. (Cited on page 11.)

[56] Yatin Vasant Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. Coverage estimation for symbolic model checking. In *DAC*, pages 300–305, 1999. (Cited on pages 3 and 81.)

[57] IEEE Standards Department. *IEEE Standard for Verilog Hardware Description Language*. Number 1364-2001 in IEEE Standards. IEEE, 2001. (Cited on page 50.)

[58] Barbara Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *FMCAD*, pages 117–124. IEEE Computer Society, 2006. (Cited on page 2.)

[59] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In Damm and Hermanns [33], pages 258–262. (Cited on pages 3, 5, 6, 8 and 48.)

[60] Sagi Katz, Orna Grumberg, and Daniel Geist. "Have i written enough properties?" - a method of comparison between specification and implementation. In Laurence Pierre and Thomas Kropf, editors, *CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 280–297. Springer, 1999. (Cited on pages 3 and 81.)

[61] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD*. IEEE Computer Society, 2009. To appear. (Cited on page 3.)

[62] Eleftherios Koutsofios and Stephen C. North. Drawing graphs with Dot. Technical Report 910904-59113-08TM, AT&T Bell Laboratories, 1991. (Cited on page 31.)

[63] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990. (Cited on page 3.)

[64] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983. (Cited on page 13.)

[65] Thomas Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. (Cited on page 1.)

[66] Hillel Kugler, Cory Plock, and Amir Pnueli. Controller synthesis from LSC requirements. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2009. (Cited on page 3.)

[67] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Safraless compositional synthesis. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006. (Cited on page 2.)

[68] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542. IEEE Computer Society, 2005. (Cited on page 2.)

[69] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006. (Cited on page 3.)

[70] Martin Leucker. Model checking games for the alternation-free μ-calculus and alternating automata. In Harald Ganzinger, David A. McAllester, and Andrei Voronkov, editors, *LPAR*, volume 1705 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 1999. (Cited on page 82.)

[71] Martin Leucker and Thomas Noll. Truth/SLC - A parallel verification platform for concurrent systems. In Berry et al. [6], pages 255–259. (Cited on pages 7 and 82.)

[72] Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In Damm and Hermanns [33], pages 95–107. (Cited on page 3.)

[73] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. (Cited on page 1.)

[74] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from `http://homepages.inf.ed.ac.uk/perdita/cwb/`. Last visit in October of 2009. (Cited on page 82.)

[75] Ryosei Mori and Naoki Yonezaki. Several realizability concepts in reactive objects. *Information Modeling and Knowledge Bases*, 1993. (Cited on pages 26 and 82.)

[76] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In Agostino Cortesi and Gilberto Filé, editors, *SAS*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1999. (Cited on page 1.)

[77] Glenford J. Myers. *The art of software testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. (Cited on page 1.)

[78] Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997. (Cited on page 1.)

[79] Ingo Pill, Simone Semprini, Roberto Cavada, Marco Roveri, Roderick Bloem, and Alessandro Cimatti. Formal analysis of hardware requirements. In Ellen Sentovich, editor, *DAC*, pages 821–826. ACM, 2006. (Cited on pages 3, 6, 20 and 48.)

[80] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006. (Cited on pages 3, 6, 10, 11, 12, 13, 14, 15, 17, 36, 37, 38, 39 and 50.)

[81] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977. (Cited on pages 1 and 9.)

[82] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989. (Cited on pages 2, 5 and 26.)

[83] Vaughan R. Pratt. Anatomy of the pentium bug. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer, 1995. (Cited on page 1.)

[84] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969. (Cited on page 2.)

[85] Michael Oser Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, Boston, MA, USA, 1972. (Cited on page 2.)

[86] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992. (Cited on page 2.)

[87] Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 149–167. Springer, 2007. (Cited on page 36.)

[88] Shmuel Safra. On the complexity of omega-automata. In *FOCS*, pages 319–327. IEEE, 1988. (Cited on page 2.)

[89] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985. (Cited on page 1.)

[90] Aravinda Prasad Sistla. *Theoretical issues in the design and verification of distributed systems*. PhD thesis, Harvard University, Cambridge, MA, USA, 1983. (Cited on page 9.)

[91] Perdita Stevens and Colin Stirling. Practical model-checking using games. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 85–101. Springer, 1998. (Cited on pages 7 and 82.)

[92] Colin Stirling. Lokal model checking games. In Insup Lee and Scott A. Smolka, editors, *CONCUR*, volume 962 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 1995. (Cited on pages 7 and 82.)

[93] Li Tan. Playgame: A platform for diagnostic games. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 492–495. Springer, 2004. (Cited on pages 7 and 82.)

[94] Wolfgang Thomas. Languages, automata and logic. In Arto Salomaa and Grzegorz Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer-Verlag, Berlin, 1997. (Cited on pages 2, 10 and 11.)

[95] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *ICCAD*, pages 130–133, 1990. (Cited on page 44.)

[96] Stavros Tripakis and Karine Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 233–252. Springer, 1999. (Cited on pages 7 and 81.)

[97] Moshe Y. Vardi. From church and prior to PSL. In Grumberg and Veith [50], pages 150–171. (Cited on page 9.)

[98] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2):183–221, 1986. (Cited on page 2.)

[99] Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In Oscar H. Ibarra and Zhe Dang, editors, *CIAA*, volume 2759 of *Lecture Notes in Computer Science*, pages 11–22. Springer, 2003. (Cited on page 3.)

[100] Noriaki Yoshiura. Finding the causes of unrealizability of reactive system formal specifications. In *SEFM*, pages 34–43. IEEE Computer Society, 2004. (Cited on page 82.)

[101] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999. (Cited on page 17.)

[102] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. (Cited on pages 17, 18, 20, 22 and 23.)