Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Debugging Formal Specifications [⋆]

A Practical Approach using Model-Based Diagnosis and Counterstrategies

Robert Könighofer, Georg Hofferek, Roderick Bloem

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

Received: date / Revised version: date

Abstract. Creating a formal specification for a design is an error-prone process. At the same time, debugging incorrect specifications is difficult and time-consuming. In this work, we propose a debugging method for formal specifications that does not require an implementation.

We handle conflicts between a formal specification and the informal design intent using a simulation-based refinement loop, where we reduce the problem of debugging overconstrained specifications to that of debugging unrealizability. We show how model-based diagnosis can be applied to locate an error in an unrealizable specification. The diagnosis algorithm computes properties and signals that can be modified in such a way that the specification becomes realizable, thus pointing out potential error locations. In order to fix the specification, the user must understand the problem. We use counterstrategies to explain conflicts in the specification. Since counterstrategies may be large, we propose several ways to simplify them. First, we compute the counterstrategy not for the original specification but only for an unrealizable core. Second, we use a heuristic to search for a countertrace, i.e., a single input trace which necessarily leads to a specification violation. Finally, we present the countertrace or the counterstrategy as an interactive game against the user, and as a graph summarizing possible plays of this game. We introduce a user-friendly implementation of our debugging method and present experimental results for GR(1) specifications.

1 Introduction

Ideally, a formal specification for a design is written before the design is implemented. This establishes an unambiguous notion of correctness, which can be used as an objective during the implementation phase. It also makes the informal design intent precise, thus preventing misunderstandings between collaborating designers. Clearly, the specification must have the highest possible quality when used as correctness objective for the implementation. The process of deriving an implementation from the formal specification can also be automated using property synthesis techniques [40,31,24,39,17,36]. This yields implementations which are *correct by construction* with respect to the formal specification. In such a synthesis-based design flow, a high quality specification is even more crucial since the synthesized implementation can only be as correct as its specification.

Formal specifications are also created and sold as intellectual property for verification [15]. In this scenario, too, a specification has to be created that reflects exactly the informal design intent without a corresponding implementation being available.

Like other engineering processes, constructing a high quality formal specification is a difficult task [25,38,12,22,11], especially if no implementation is present. Mistakes can lead to various flaws. First, a specification may be incomplete. This means that the specification allows implementations that do not conform to the informal design intent. Second, the specification may be unsound. This is the case if it disallows implementations that are valid with respect to the design intent. As a special case, the specification may be unrealizable. An unrealizable specification does not allow any implementation. In this article we introduce debugging techniques for unrealizability and unsoundness.

It is possible to turn an unsound specification into an unrealizable one by adding properties to enforce the desired behavior that is forbidden by the specification. We use this idea to reduce the problem of debugging unsound specifications to the problem of debugging unrealizable specifications. However, unrealizability is also

[⋆] This work was supported in part by the European Commission through the projects COCONUT (FP7-2007-IST-1-217069) and DIAMOND (FP7-2009-IST-4-248613)

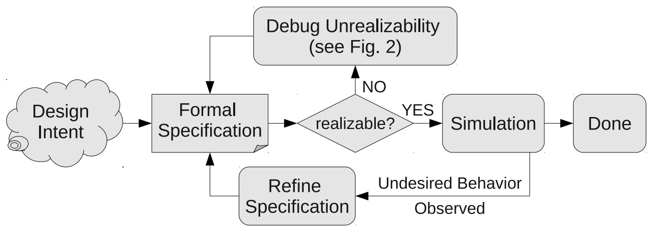


Fig. 1: Refining a specification.

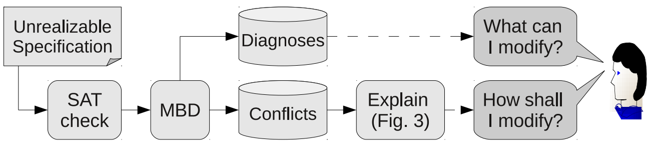


Fig. 2: Our approach to debug unrealizability.

a serious problem of its own. Our experience in writing complete formal specifications (e.g., [3,4]) for property synthesis tools shows that many mistakes during specification development lead to unrealizability. Debugging an unrealizable specification is difficult. Even an enormous piece of software can be executed to track down the error, but this is not possible for an unrealizable specification. It is important to note that, for systems with inputs and outputs, there is a difference between the realizability and the satisfiability of a specification. A specification is satisfiable if one trace of input values and output values fulfilling the specification exists. This is a necessary but not a sufficient condition for realizability. A specification is only realizable if there is a valid output trace for *every* possible input trace. Additionally, the outputs in any particular time step may depend on past and present inputs only. Tools such as RAT [38] explain why a single trace does not conform to the specification. However, this does not suffice to explain unrealizability.

Context

In this work, we address the debugging of specifications for reactive systems. A reactive system is a system which interacts with its environment via inputs and outputs in an infinite execution. Our debugging method is based on the synthesis of reactive systems conforming to a given specification, and on the synthesis of corresponding counterstrategies. We assume specifications to consist of environment assumptions and system guarantees: If the environment fulfills all assumptions, then the specification requires the system to fulfill all guarantees. Our debugging approach can deal with a wide range of such specifications. In addition to a general description, we also explain how it can be applied to Generalized Reactivity(1) [39]. For the latter, synthesis is performed using fixpoint computations over states in a game graph, implemented symbolically using BDDs [6]. Our debugging technique analyzes specifications stand-alone, i.e., it does not require a corresponding implementation.

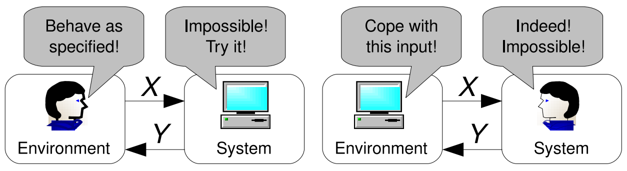
Outline of our Debugging Approach

This article is based on [27], [29], [2], and [28]. We propose the flow depicted in Fig. 1 to create a high quality specification. The user creates an initial formal specification of the design. If the specification is unrealizable, it needs to be debugged. If the specification is realizable, it can be simulated. If undesired behavior (with respect to the informal design intent) is observed during simulation, the specification is refined to exclude this undesired behavior. The refined specification can be realizable or unrealizable again. This is repeated until the user is satisfied with the behavior of the design.

Unrealizability is debugged as illustrated in Fig. 2. First, we check for satisfiability. If the specification is unsatisfiable, existing trace-based debugging methods [38] can be applied. However, since unsatisfiability is a special case of unrealizability, our debugging method works in any case. Next, we use model-based diagnosis [41,26] (MBD) to identify possible error locations in the specification. In order to apply MBD, we take the unrealizable specification as an inherently conflicting model. As diagnoses we get sets of guarantees and outputs that can be weakened in order to make the specification realizable. The computation of diagnoses relies on the computation of minimal conflicts in the specification. Minimal conflicts are unrealizable cores, i.e., minimal subsets of the specification which are unrealizable on their own. Diagnoses indicate possible error locations. However, in order to be able to find the best repair for unrealizability, the user additionally has to understand the problem.

Explaining unrealizability means to explain why no implementation can fulfill the specification. We cannot explain for every possible implementation why it does not conform to the specification. However, the user must have an implementation in his mind when creating the formal specification. We show that this imagined implementation does not conform to the formal specification by swapping the roles between the tool and the user as illustrated in Fig. 3. The tool takes on the role of the environment and the user takes the role of the system. The two parties play a *diagnostic game*: The tool provides inputs and the user tries to provide outputs that conform to the specification. Since the specification is unrealizable, there exists a so-called *counterstrategy* for the environment, which makes the user fail for sure. However, while trying, she will understand why there is no way to fulfill the specification, i.e., why the specification is unrealizable. This knowledge can then be used to correct the specification.

In general, a counterstrategy cannot be presented as a single trace of inputs, since inputs may depend on



(a) Simulating the System (b) Debugging Unrealizability

Fig. 3: Explaining unrealizability by swapping the roles.

previous outputs. Our experience shows that these dependencies can become quite complex, especially for a large unrealizable specification. This makes it difficult for the user to comprehend which environment behavior leads to problems and why the specification is unrealizable. We therefore propose several techniques to simplify counterstrategies. First, we compute the counterstrategy not for the original specification, but for an unrealizable core. Every unrealizable core contains one root cause of unrealizability. Explaining unrealizable cores instead of the entire unrealizable specification allows the user to focus on one problem at a time. When computing unrealizable cores, we not only remove properties but also signals from the specification. This allows the user to focus on those signals that are relevant for the conflict when playing the diagnostic game. Second, we attempt to compute what we call a *countertrace*. A countertrace is a single trace of inputs such that no behavior of the system can fulfill the specification. A countertrace does not always exist. Even if one exists, its computation is often expensive. Hence, we present a heuristic.

This article is organized as follows. Section 2 discusses related work. Section 3 gives definitions and establishes notation. In Section 4, we describe our debugging approach in a generic way; we elaborate it for Generalized Reactivity(1) [39] specifications in Section 5. An implementation in the requirements analysis and synthesis tool RATSY [2] is presented in Section 6. Section 7 presents experimental results and Section 8 concludes.

2 Related Work

Incomplete specifications and coverage metrics have already been addressed before in various ways. There is existing work on completeness checking with respect to a given implementation. This includes the definition of different comparison criteria between the tableau of the specification and the implementation [25], identification of parts of the state space of the implementation which are covered by the specification [23], and checking if modifications introduced into the model of the implementation are detected by the specification [8,9]. On the other hand, there are methods for coverage analysis that do not rely on a particular implementation. A notion of "forgotten cases", which are situations where a certain output in a certain time step is not constrained, is introduced in [12]. Another concept is that of *inherent vacuity* [18]. A specification is inherently vacuous if it can be mutated into a simpler but equivalent specification. Other work checks the specification with respect to a high-level fault model by testing if an implementation which contains stuck-at faults can still conform to the specification [14]. Our debugging approach allows the user to eliminate incompleteness that shows up in simulation. Apart from that, we do not specifically address incomplete specifications. Rather, we focus on specifications which are unsound.

Model-based diagnosis [26,41], which was originally invented to automate debugging of misbehaving physical systems, has already been applied for diagnosis in various settings such as logic programs [13], functional programs [46], VHDL designs [20], Java programs [35], knowledge bases [16], and ontologies [19]. We use model-based diagnosis to perform error localization in unrealizable specifications of reactive systems. This deviates from the standard setting in that there is no observation contradicting a system description, but only an inherently conflicting system description. Our method is similar to [34], where hitting sets are used to compute all unsatisfiable cores in an unsatisfiable constraint system.

Yoshiura [49] addresses the problem of debugging unrealizable specifications by defining several bug localization heuristics based on a classification (into *strong satisfiable*, *stepwise satisfiable*, and *stepwise strong satisfiable*) and the tableau of the specification. Their objective is similar to ours, but there are few similarities in methodology. Cimatti et al. [11] propose to present an unrealizable core as diagnostic aid for unrealizability. Our work extends theirs: We use unrealizable cores for diagnoses computation and to obtain more focused counterstrategies. As an improvement, we not only remove properties but also signals from the specification. Furthermore, we use Delta Debugging [50] as a more advanced and often faster algorithm for unrealizable core computation. While single unrealizable cores help to understand problems in the specification, they are less useful for suggesting locations to fix them. The reason is that many unrealizable cores may exist, and that a repair has to resolve them all. We address this problem with the model-based diagnosis step, which takes *all* unrealizable cores into account.

Counterstrategies have been used or at least mentioned as debugging aids for unrealizability in various settings such as game graphs with controllable and uncontrollable edges [48], Live Sequence Charts [5], timed automata [1], and for model-checking of the modal μ-calculus [45,44] with more efficient algorithms and implementations presented in [32,33,47]. None of these papers mention the simplification of counterstrategies in order to provide the user with helpful diagnostic information. This is one of the main contributions of our work. In particular, we are not aware of any previous work on countertraces. Also, we do not know of any existing publication showing how unsound specifications can be debugged with counterstrategies.

3 Preliminaries

3.1 Model-Based Diagnosis

Model-based diagnosis (MBD) [26,41] is a method to identify possible error locations in a system. In this work, we follow the definitions and notation of [41]. Let SD be a description of the correct behavior of a system, and let OBS be an observation of an erroneous behavior. Both SD and OBS are sets of logical sentences. The system is assumed to consist of a set COMP of components. Every component $c \in \mathsf{COMP}$ can behave abnormally (denoted $\mathsf{AB}(c)$) or normally (denoted $\neg\mathsf{AB}(c)$). The behavior of a component c is described with a logical sentence of the form $\neg\mathsf{AB}(c) \Rightarrow N_c$, where N_c defines the normal behavior of c. That is, if the component behaves abnormally, it can exhibit any behavior. The system description SD consists of component descriptions and a definition of their interconnections or interplay, again in terms of logical sentences. The observation OBS is in contradiction with the system description. That is, if all components behaved normally, it would be impossible to observe OBS. More formally, the set $\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\mathsf{AB}(c) \mid c \in \mathsf{COMP}\}$ of logical sentences is inconsistent, i.e., contains a logical contradiction. We write $\neg\mathsf{consistent}(\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\mathsf{AB}(c) \mid c \in \mathsf{COMP}\})$ to express this.

MBD identifies sets of components that may have caused the erroneous behavior OBS. Such sets are called diagnoses. Formally, a set $\Delta \subseteq \mathsf{COMP}$ is a *diagnosis* iff it is a minimal set of components such that

$$\mathsf{consistent}(\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\mathsf{AB}(c) \mid c \in \mathsf{COMP} \setminus \Delta\}) \quad (1)$$

holds. Minimality means that Eq. 1 must not hold for any subset $\Delta' \subset \Delta$. Thus, the observation would be possible under the assumption that all components in a diagnosis behaved abnormally. Hence, every diagnosis represents a fault candidate. A diagnosis Δ with $|\Delta| = 1$ is called a *single-fault diagnosis.*

Diagnoses can be computed using conflicts. A *conflict* is a set $C \subseteq \mathsf{COMP}$ such that

$$\neg\mathsf{consistent}(\mathsf{SD} \cup \mathsf{OBS} \cup \{\neg\mathsf{AB}(c) \mid c \in C\}) \quad (2)$$

holds. Informally speaking, a conflict is a set of components that cannot all behave normally. If all components of a conflict behaved normally, the observation would be impossible. Again, a conflict C is *minimal* if no subset $C' \subset C$ is a conflict. A diagnosis must explain all conflicts, so it must share at least one element with every conflict. This relation can be formalized using hitting sets. A *hitting set* for a collection $\mathcal{K}$ of sets is a set H such that $\forall K \in \mathcal{K} \,.\, H \cap K \neq \emptyset$ holds. A hitting set H is *minimal* if no subset $H' \subset H$ is a hitting set for that collection. A set $\Delta \subseteq \mathsf{COMP}$ is a diagnosis iff Δ is a minimal hitting set for the collection of all minimal conflicts. Hence, computing diagnoses reduces to computing minimal hitting sets for the collection of minimal conflict sets. Reiter [41] presents an algorithm for minimal hitting set computation which computes conflicts on-the-fly and produces diagnoses in order of increasing cardinality. Diagnoses with a lower cardinality are in general considered as more likely fault candidates than diagnoses with higher cardinality. Thus, if the algorithm is aborted before all diagnoses have been computed (which is often the case), only less likely fault candidates are missed.

3.2 Automata and Machines

A *deterministic and complete Büchi word automaton (a DBW)* is a tuple $\mathcal{A} = (Q, \Sigma, T, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $T : Q \times \Sigma \rightarrow Q$ is a deterministic and complete transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of accepting states. A *run* of the automaton $\mathcal{A}$ on an (infinite) word $\overline{\sigma} = \sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $\overline{r} = q_0q_1q_2 \ldots \in Q^\omega$ such that $q_{i+1} = T(q_i, \sigma_i)$ for all $i \geq 0$. The run is *accepting* iff $\mathsf{inf}(\overline{r}) \cap F \neq \emptyset$, where $\mathsf{inf}(\overline{r})$ denotes the states occurring infinitely often in $\overline{r}$. Given two disjoint sets of Boolean inputs and outputs X and Y, we assume that $\Sigma = \mathcal{X} \times \mathcal{Y}$ is composed of an input alphabet $\mathcal{X} = 2^X$ and an output alphabet $\mathcal{Y} = 2^Y$. Moreover, we assume that $Q = 2^V$ for a set V of state bits. This allows for symbolic representations using BDDs [6]. 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 write $\overline{x}||\overline{y}$ to denote the composition $(x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$.

A *Mealy machine* is a tuple $M^e = (Q, \mathcal{X}, \mathcal{Y}, \delta, q_0, \lambda)$, where Q, $\mathcal{X}$, $\mathcal{Y}$, and q_0 are defined as for DBWs, $\delta : Q \times \mathcal{X} \rightarrow Q$ is a complete transition function, and $\lambda : Q \times \mathcal{X} \rightarrow \mathcal{Y}$ is a complete output function. Given the input trace $\overline{x} = x_0x_1 \ldots \in \mathcal{X}^\omega$, M^e produces the output trace $M^e(\overline{x}) = \lambda(q_0, x_0)\lambda(q_1, x_1) \ldots \in \mathcal{Y}^\omega$, where $q_0q_1 \ldots \in Q^\omega$ is a sequence of states with $q_{i+1} = \delta(q_i, x_i)$ for all $i \geq 0$. We denote the set of words that can be produced by M^e by $L(M^e) = \{\overline{x}||\overline{y} \in (\mathcal{X} \times \mathcal{Y})^\omega \mid M^e(\overline{x}) = \overline{y}\}$.

A *Moore machine* is a tuple $M^o = (Q, \mathcal{Y}, \mathcal{X}, \epsilon, q_0, \kappa)$ with $\epsilon : Q \times \mathcal{Y} \rightarrow Q$ and $\kappa : Q \rightarrow \mathcal{X}$. Given the trace $\overline{y} = y_0y_1 \ldots \in \mathcal{Y}^\omega$, M^o produces the trace $M^o(\overline{y}) = \kappa(q_0)\kappa(q_1) \ldots \in \mathcal{X}^\omega$, where $q_{i+1} = \epsilon(q_i, y_i)$ for all $i \geq 0$. We have that $L(M^o) = \{\overline{x}||\overline{y} \in (\mathcal{X} \times \mathcal{Y})^\omega \mid M^o(\overline{y}) = \overline{x}\}$. Refer to [21] for a more gentle introduction to automata.

3.3 Games

A (finite state, two player) *game* is a five-tuple $\mathcal{G} = (Q, \Sigma, T, q_0, \mathsf{Win})$, where Q, $\Sigma = \mathcal{X} \times \mathcal{Y}$, T, and q_0 are defined as for DBWs, and $\mathsf{Win} : Q^\omega \rightarrow \{\mathsf{false}, \mathsf{true}\}$ is the winning condition. The game is played by two players, the *environment* and the *system*. A *play* $\overline{\pi}$ of $\mathcal{G}$ is an infinite sequence $\overline{\pi} = q_0 q_1 q_2 \ldots \in Q^\omega$ of states with $q_{i+1} = T(q_i, \sigma_i)$ for all $i \geq 0$. In each step, the letter $\sigma_i = (x_i, y_i)$ is chosen by the two players in such a way that the environment first chooses an $x_i \in \mathcal{X}$, after which the system chooses some $y_i \in \mathcal{Y}$. A play is won by the system iff $\mathsf{Win}(\overline{\pi}) = \mathsf{true}$. Otherwise, it is lost for the system and won for the environment. A (finite memory) *strategy* for the environment in game $\mathcal{G}$ is a tuple $\mathcal{S} = (\Gamma, \gamma_0, \rho)$, where Γ is some (finite) set representing the memory, $\gamma_0 \in \Gamma$ is the initial memory content, and $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$ is a relation mapping a state of $\mathcal{G}$ and the memory content to a set of possible next inputs and an updated memory content. A strategy is *deterministic* iff $\forall q, \gamma \,.\, |\{(q, \gamma, x, \gamma') \in \rho\}| \leq 1$.

A *play* $\overline{\pi}$ *conforms to a strategy* $\mathcal{S}$ iff there exist sequences $(x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$ and $\gamma_0 \gamma_1 \ldots \in \Gamma^\omega$ such that, for all $i \geq 0$, $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ and $q_{i+1} = T(q_i, (x_i, y_i))$. Strategy $\mathcal{S}$ is winning for the environment from a state q_s if all plays starting in q_s and conforming to $\mathcal{S}$ are won by the environment. The set $W^{\text{env}} \subseteq Q$ of states from which such a winning strategy exists is called the *winning region* of the environment. A *counterstrategy* is a winning strategy for the environment from q_0. A counterstrategy exists if $q_0 \in W^{\text{env}}$. A *trace* $(x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$ *conforms to a strategy* $\mathcal{S}$ in $\mathcal{G}$ iff there exists a play $\overline{\pi} = q_0 q_1 \ldots$ and a sequence $\gamma_0 \gamma_1 \ldots \in \Gamma^\omega$ such that $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ and $q_{i+1} = T(q_i, (x_i, y_i))$ for all $i \geq 0$. The set of traces that conform to $\mathcal{S}$ in $\mathcal{G}$ is denoted $L(\mathcal{G}, \mathcal{S})$. A Moore machine M^o *implements* a deterministic strategy $\mathcal{S}$ in $\mathcal{G}$ iff $L(M^o) = L(\mathcal{G}, \mathcal{S})$. A construction of M^o is straightforward. A nondeterministic strategy has to be determinized beforehand. See [21] for a more comprehensive introduction.

3.4 μ-Calculus

The (propositional) μ-calculus [30] extends propositional logic with a least fixpoint operator μ and a greatest fixpoint operator ν. We further extend it by two preimage operators MX^s and MX^e and use it to describe fixpoint computations over sets $Q' \subseteq Q$ of states in a game $\mathcal{G} = (Q, \Sigma, T, q_0, \mathsf{Win})$.

Let Var be a set of variables ranging over subsets of Q. The syntax of μ-calculus formulas can be defined inductively as follows: Every variable $Z \in$ Var and every set $Q' \subseteq Q$ of states is a μ-calculus formula. Given that R and S are μ-calculus formulas, then so are $\neg R$, $R \cup S$, and $R \cap S$, with the expected semantics. Finally, for $Z \in$ Var, we have that $\mu Z \,.\, R(Z)$, $\nu Z \,.\, R(Z)$, $\mathsf{MX}^s(R)$, and $\mathsf{MX}^e(R)$ are μ-calculus formulas. These are defined as

$$\mu Z.R(Z) = \bigcup_i Z_i, \text{ with } Z_0 = \emptyset \text{ and } Z_{i+1} = R(Z_i), \quad (3)$$

$$\nu Z.R(Z) = \bigcap_i Z_i, \text{ with } Z_0 = Q \text{ and } Z_{i+1} = R(Z_i),$$

$$\mathsf{MX}^s(R) = \{q \in Q \mid \forall x \in \mathcal{X} \,.\, \exists y \in \mathcal{Y} \,.\, T(q, (x, y)) \in R\},$$

$$\mathsf{MX}^e(R) = \{q \in Q \mid \exists x \in \mathcal{X} \,.\, \forall y \in \mathcal{Y} \,.\, T(q, (x, y)) \in R\}.$$

We will refer to the sets Z_i as the iterates of the fixpoint. The operation $\mathsf{MX}^s(R)$ gives all states from which the system is able to force the play into a state of R in one step. Analogously, $\mathsf{MX}^e(R)$ gives all states from which the environment can enforce a visit to R in one step.

3.5 Specifications for Reactive Systems

A reactive system is a Mealy machine that continuously interacts with its environment via inputs X and outputs Y. The specifications we consider are of the form $\varphi = (A, G)$, where A is a set of environment assumptions and G is a set of system guarantees. Let $\overline{\sigma} \models p$ denote that a trace $\overline{\sigma} \in \Sigma^\omega$ fulfills an assumption or guarantee $p \in A \cup G$. Then $\overline{\sigma}$ fulfills $\varphi = (A, G)$, written $\overline{\sigma} \models \varphi$, iff

$$\big(\forall a \in A \,.\, \overline{\sigma} \models a\big) \text{ implies } \big(\forall g \in G \,.\, \overline{\sigma} \models g\big). \quad (4)$$

We assume that properties $\top$ and $\bot$ can be formulated with $\overline{\sigma} \models \top$ and $\overline{\sigma} \not\models \bot$ for any $\overline{\sigma} \in \Sigma^\omega$. A Mealy machine M^e implements a specification φ iff $\overline{\sigma} \models \varphi$ holds for all $\overline{\sigma} \in L(M^e)$. A specification φ is unrealizable, iff no Mealy machine implements it.

A *Generalized Reactivity(1)* specification [39] (we will write GR(1)) is an instance of a specification of the form (A, G). It consists of m DBWs $\mathcal{A}_i^e = (Q_i^e, \Sigma, T_i^e, q_{0,i}^e, F_i^e)$ representing the environment assumptions, and n DBWs $\mathcal{A}_j^s = (Q_j^s, \Sigma, T_j^s, q_{0,j}^s, F_j^s)$ representing the system guarantees. A trace $\overline{\sigma} \in \Sigma^\omega$ fulfills assumption or guarantee $\mathcal{A}$ if the corresponding run is accepting in $\mathcal{A}$. A game $\mathcal{G}^{\text{GR1}} = (Q, \Sigma, T, q_0, \mathsf{Win})$ can be built with state space $Q = Q_1^e \times \cdots \times Q_m^e \times Q_1^s \times \cdots \times Q_n^s$, transition function $T\left((q_1^e, \ldots, q_n^s), \sigma\right) = \left(T_1^e(q_1^e, \sigma), \ldots, T_n^s(q_n^s, \sigma)\right)$, and initial state $q_0 = (q_{0,1}^e, \ldots, q_{0,n}^s)$. Let $J_i^e = \{(q_1^e, \ldots, q_n^s) \mid q_i^e \in F_i^e\}$ be the set of all states of $\mathcal{G}^{\text{GR1}}$ that are accepting in $\mathcal{A}_i^e$. Similarly, let J_j^s be the set of all states of $\mathcal{G}^{\text{GR1}}$ that are accepting in $\mathcal{A}_j^s$. Then $\mathsf{Win}(\overline{\pi})$ is true iff

$$(\forall i \,.\, \mathsf{inf}(\overline{\pi}) \cap J_i^e \neq \emptyset) \text{ implies } (\forall j \,.\, \mathsf{inf}(\overline{\pi}) \cap J_j^s \neq \emptyset). \quad (5)$$

The winning region $W_{\text{sys}}^{\text{GR1}}$ of the system is [39]

$$W_{\text{sys}}^{\text{GR1}} = \nu Z \,.\, \bigcap_{j=1}^{n} \mu Y \,.\, \bigcup_{i=1}^{m} \nu X \,.$$

$$\big(J_j^s \cap \mathsf{MX}^s(Z)\big) \cup \mathsf{MX}^s(Y) \cup \big(\neg J_i^e \cap \mathsf{MX}^s(X)\big). \quad (6)$$

The result of the greatest fixpoint in X contains all states from which the system can enforce that the play either stays in $\neg J_i^e$ for some i or eventually reaches the set $J_j^s \cap \mathsf{MX}^s(Z) \cup \mathsf{MX}^s(Y)$. Both cases are winning for the system. The former means that an assumption is violated. The latter is winning because the fixpoints in Y and Z ensure that all sets J_j^s of accepting states of the system can be visited infinitely often, thus fulfilling all guarantees. The winning region for the environment is $W_{\text{env}}^{\text{GR1}} = Q \setminus W_{\text{sys}}^{\text{GR1}}$. A GR(1) specification is realizable iff $q_0 \in W_{\text{sys}}^{\text{GR1}}$ in the corresponding game.

3.6 Failure Preserving Minimization Algorithms

We will use two different algorithms to compute minimal unrealizable cores.

Delta Debugging [50] is an algorithm to isolate the trigger of a failure. Given a procedure test and some input C that makes test fail (denoted $\mathsf{test}(C) = \text{✘}$), the algorithm computes a minimal input $\hat{C} = \mathsf{ddmin}^{\mathsf{test}}(C)$ with $\hat{C} \subseteq C$ such that test still fails on $\hat{C}$. We assume that test is monotonic, i.e., that for all $C'' \subseteq C' \subseteq C$ we have that $\mathsf{test}(C'') = \text{✘}$ implies $\mathsf{test}(C') = \text{✘}$. The Delta Debugging algorithm is defined as $\mathsf{ddmin}^{\mathsf{test}}(C) = \mathsf{ddmin}_2^{\mathsf{test}}(C, 2)$, with $\mathsf{ddmin}_2^{\mathsf{test}}(C', n) =$

$$\begin{cases} \mathsf{ddmin}_2^{\mathsf{test}}(C'_i, 2) & \text{if } \exists i \,.\, \mathsf{test}(C'_i) = \text{✘} \\ \mathsf{ddmin}_2^{\mathsf{test}}(\overline{C'_i}, \max(n-1, 2)) & \text{else if } \exists i \,.\, \mathsf{test}(\overline{C'_i}) = \text{✘} \\ \mathsf{ddmin}_2^{\mathsf{test}}(C', \min(|C'|, 2n)) & \text{else if } n < |C'| \\ C' & \text{otherwise.} \end{cases}$$

The sets $C'_1, \dots, C'_n$ form a partition of C' into n (approximately) equally sized parts, and $\overline{C'_i} = C' \setminus C'_i$. The procedure ddmin_2 first tries to find a subset C'_i which still fails the test. If such a subset is found, it is reduced further by a recursive call. If not, the complements $\overline{C'_i}$ are tried. If this does not work either, the granularity n for the search is doubled. If n cannot be increased any further, the current set must be minimal. In the best case, the number of calls to test is logarithmic in $|C|$. In the worst case, it is quadratic in $|C|$.

The second failure preserving input minimization algorithm we will use is simpler. It is defined as $\hat{C} = \mathsf{linMin}^{\mathsf{test}}(C) = \mathsf{linMin}_2^{\mathsf{test}}(C, C)$ with $\mathsf{linMin}_2^{\mathsf{test}}(T, R) =$

$$\begin{cases} R & \text{if } T = \emptyset \\ \mathsf{linMin}_2^{\mathsf{test}}(T \setminus t, R \setminus t) & \text{else if } \mathsf{test}(R \setminus t) = \text{✘} \\ \mathsf{linMin}_2^{\mathsf{test}}(T \setminus t, R) & \text{otherwise,} \end{cases}$$

where t is a randomly chosen element of T. This algorithm, used in [11], attempts to remove one element after the other, thus requiring exactly $|C|$ checks.

4 Debugging Approach

In this section, we introduce our debugging technique in a generic way. We assume that the specification consists of a (possibly empty) set of environment assumptions A and a set of system guarantees G, and that it is possible to add and remove guarantees. Furthermore, procedures realizable and sat deciding realizability and satisfiability of a specification are required. We also assume that the specification can be turned into a game $\mathcal{G}$ and that a finite memory counterstrategy can be computed in the case of unrealizability. Moreover, we assume that output signals can be existentially quantified in guarantees. All these are rather loose restrictions that apply to many common logics such as LTL, PSL, CTL, or S1S, to name only a few.

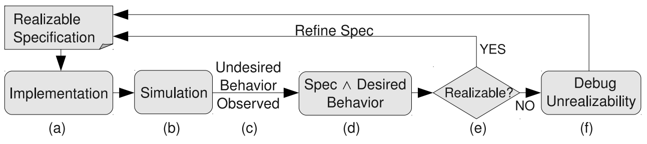


Fig. 4: Our method to fix mismatches with the design intent.

4.1 Debugging Undesired Behavior

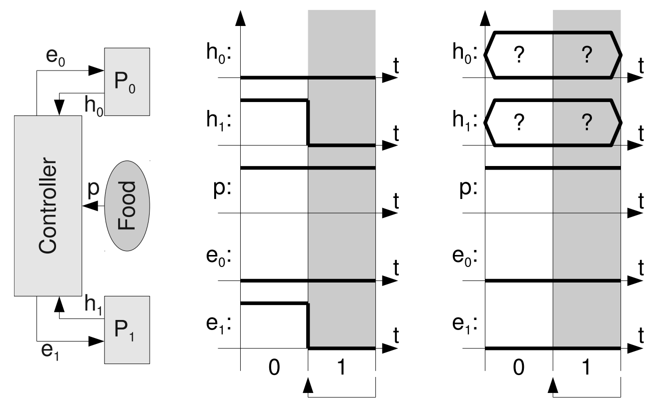
Deviations of the formal specification from the informal design intent often show up when a system correctly implementing the specification is simulated. If undesired behavior, i.e., behavior that should be forbidden according to the design intent, is observed, then this means that the specification is incorrect and needs to be refined.

Fig. 4 illustrates our method to tackle this problem. Suppose that undesired behavior is observed during the simulation of an implementation of the specification (Fig. 4a-4c). The tool first checks whether the trace fulfills the assumptions. If not, a warning is issued because the system does not need to fulfill any guarantee in this case. Otherwise, the user is asked to specify the desired response to the input trace that was used in simulation. The specification is then augmented with a guarantee enforcing this desired behavior (Fig. 4d). Two cases can be distinguished (Fig. 4e).

1. The augmented specification is realizable. That is, the original specification leaves enough freedom to choose either the observed or the desired behavior. It is incomplete and needs to be refined.
2. The augmented specification is unrealizable. That is, the original specification disallows the desired behavior because no system can fulfill both. Hence, the original specification is not sound.[1] We debug unsoundness by debugging the unrealizability of the augmented specification (Fig. 4f).

We propose that the user defines the desired behavior simply by modifying the obtained simulation trace. Traces are infinite in our setting, so we assume some finite representation such as a finite stem followed by a loop. We allow the user to change any signal value in any time step to 0, 1, or **?**, where **?** represents "don't care". Due to the "don't cares", we obtain a trace template $\overline{t}$, consisting of an input part $\overline{t^x}$ and an output part $\overline{t^y}$. The user expresses the desired behavior as follows: Whenever the sequence of inputs matches the input part $\overline{t^x}$ of the template, the outputs have to follow $\overline{t^y}$. The use of "don't cares" in $\overline{t^x}$ and $\overline{t^y}$ allows the definition of rather general requirements. The tool converts $\overline{t}$ into a guarantee g_{new} enforcing $\overline{t^y}$ if the inputs conform to $\overline{t^x}$.

[1] It is also not complete since it allows the undesired behavior that has been observed, but this incompleteness is eliminated by the augmentation.



(a) Example (b) Simulation (c) Design Intent

Fig. 5: Example: Debugging undesired behavior.

More formally, $\overline{t^x}$ can be seen as a sequence $t_0^x t_1^x t_2^x \dots$ of functions $t_i^x : X \to \{0, 1, \textbf{?}\}$ for all $i \geq 0$. Analogously, $\overline{t^y}$ is a sequence of functions $t_i^y : Y \to \{0, 1, \textbf{?}\}$. A trace $\overline{\sigma} = (x_0, y_0)(x_1, y_1) \ldots \in (\mathcal{X} \times \mathcal{Y})$ conforms to $\overline{t^x}$, written $\overline{\sigma} \sqsubseteq \overline{t^x}$, iff for all $i \geq 0$ and $v \in X$ it is true that $t_i^x(v) = 0$ implies $v \notin x_i$ and $t_i^x(v) = 1$ implies $v \in x_i$. Conformance $\overline{\sigma} \sqsubseteq \overline{t^y}$ with the output part of the template is defined analogously. The guarantee g_{new} must be defined in such a way that it accepts a trace $\overline{\sigma}$ iff $\overline{\sigma} \sqsubseteq \overline{t^x}$ implies $\overline{\sigma} \sqsubseteq \overline{t^y}$.[2] How g_{new} is actually constructed depends on the specification language.

Guarantee g_{new} is added to specification $\varphi = (A, G)$ to obtain $\varphi' = (A, G \cup g_{\text{new}})$. If φ' is realizable, then φ is incomplete and φ' is a refinement which makes it more complete. If φ' is unrealizable, then this means that φ is so restrictive that no system can fulfill φ and the desired behavior formalized in g_{new} at the same time. Resolving this conflict between φ and g_{new} is equivalent to resolving unrealizability of φ'. The next sections will explain how unrealizable specifications are debugged.

Example 1. We use a specification of a controller for two dining philosophers P_0 and P_1 to illustrate our debugging method throughout this article. The system is illustrated in Fig. 5a. The inputs h_0 and h_1 signal to the controller that P_0 or P_1 is hungry. Input p indicates whether the food is poisoned. The outputs e_0 and e_1 are set if the respective philosopher is allowed to eat. A first version of the specification could be $\varphi_1 = (\{a_1\}, \{g_1, g_2, g_3\})$ with

$$\begin{aligned}
a_1 &= \mathsf{always}(p) \mathsf{~or~} \mathsf{always}(\mathsf{not}\, p) \\
g_1 &= \mathsf{always}(\mathsf{not}(e_0) \mathsf{~or~} \mathsf{not}(e_1)) \\
g_2 &= \mathsf{always}(\mathsf{eventually}(\mathsf{not}(h_0) \mathsf{~or~} e_0)) \\
g_3 &= \mathsf{always}(\mathsf{eventually}(\mathsf{not}(h_1) \mathsf{~or~} e_1))
\end{aligned}$$

Assumption a_1 states that food is either always poisoned or never. Guarantee g_1 requires that P_0 and P_1 do not eat simultaneously. The guarantees g_2 and g_3 ensure that no philosopher starves forever. Specification φ_1 is realizable and Fig. 5b depicts a possible simulation run. The gray background marks the part of the trace that repeats infinitely often. In this run, P_1 is allowed to eat although the food is poisoned. In order to exclude this undesired behavior, the user corrects the trace as shown in Fig. 5c. The tool transforms this trace into a guarantee

$$g_4 = \mathsf{always}(p) \mathsf{~implies~} \mathsf{always}(\mathsf{not}(e_0) \mathsf{~and~} \mathsf{not}(e_1))$$

and refines φ_1 to $\varphi_2 = (\{a_1\}, \{g_1, g_2, g_3, g_4\})$, which is unrealizable. That is, φ_1 is in conflict with the desired behavior specified in Fig. 5c. Debugging this conflict, carried out by debugging the unrealizability of φ_2, will be explained in the next sections.

4.2 Model-Based Diagnosis for Unrealizability

This section shows how model-based diagnosis (MBD) can be applied to locate the error in an unrealizable specification. In contrast to the standard MBD setting, we do not diagnose a conflict between a system description and an observation but an inherently conflicting system description, namely the unrealizable specification.

4.2.1 Diagnosis of Guarantees

Let $\varphi = (A, G)$ be an unrealizable specification over inputs X and outputs Y. Our goal for this section is to identify sets of components in φ that can be modified in such a way that the specification becomes realizable. The first question that naturally arises is how to define a component in this setting. A first idea could be to make all assumptions and guarantees components, i.e., $\mathsf{COMP}_{A,G} = A \cup G$, because they typically define relatively self contained and independent aspects of the system behavior. However, the following problem arises with this definition.

Proposition 1. *If a diagnosis is defined to be a minimal set $\Delta_{A,G} \subseteq (A \cup G)$ of assumptions and guarantees which can be modified in such a way that the specification becomes realizable, then every set $\{a\}$ for $a \in A$ is a diagnosis.*

This is a direct consequence of Eq. 4, since replacing any assumption with $\bot$ gives a realizable specification. In other words, it does not make sense to search for assumptions that can be modified to obtain a realizable specification, because every assumption can be modified in such a way. However, not every guarantee can be changed to fix unrealizability. Therefore, we use the component definition $\mathsf{COMP}_G = G$, implicitly assuming that all assumptions are as intended by the designer. A complementary approach, which deals with how to find suitable environment assumptions, has been presented by Chatterjee et al. [7].

[2] Note that having g_{new} contain an implication does not break the splitting into assumptions and guarantees. It just reflects the fact that a trace is fulfilled by g_{new} if it either violates the input part or fulfills the output part of the user-given trace template.

We define the system description SD to be the tuple (A, G, X, Y). There is no observation in our setting. While standard MBD attempts to explain logical inconsistencies, we need to explain the unrealizability of a specification. As a notion of consistency we therefore define a function $\mathsf{consistent}_G^{\mathsf{SD}} : 2^{\mathsf{COMP}_G} \to \{\mathsf{true}, \mathsf{false}\}$ as $\mathsf{consistent}_G^{\mathsf{SD}}(G') = \mathsf{realizable}((A, G'))$. Intuitively, this function $\mathsf{consistent}_G^{\mathsf{SD}}$ maps a set $G' \subseteq \mathsf{COMP}_G$ of components to true iff (A, G') is a realizable specification.

In order to bridge the gap between our definition of consistency and that of Section 3.1, which is based on normally and abnormally behaving components, we can transform φ to $\tilde{\varphi}$ such that for every trace $\overline{\sigma} \in \Sigma^\omega$ we have that $\overline{\sigma} \models \tilde{\varphi}$ iff

$$\big(\forall a \in A \,.\, \overline{\sigma} \models a\big) \text{ implies } \big(\forall g \in G \,.\, \mathsf{AB}(g) \vee (\overline{\sigma} \models g)\big).$$

That is, abnormal guarantees do not have to be fulfilled in $\tilde{\varphi}$ (cf. Eq. 4). Consequently, $\mathsf{consistent}_G^{\mathsf{SD}}(G') = \mathsf{true}$ iff $\tilde{\varphi}$ is realizable with $\neg\,\mathsf{AB}(g')$ for all $g' \in G'$. Since guarantees that do not have to be fulfilled can be thought of being removed from the specification, this is equivalent to the simpler definition given above.

Lemma 1. *Function* $\mathsf{consistent}_G^{\mathsf{SD}}$ *is monotonic, i.e., for all* $G'' \subseteq G' \subseteq G$ *we have that* $\neg\,\mathsf{consistent}_G^{\mathsf{SD}}(G'')$ *implies* $\neg\,\mathsf{consistent}_G^{\mathsf{SD}}(G')$.

This is obvious since adding guarantees to an unrealizable specification preserves unrealizability. Similar to Eq. 2, a conflict is now a set $C_G \subseteq \mathsf{COMP}_G$ such that $\mathsf{consistent}_G^{\mathsf{SD}}(C_G) = \mathsf{false}$, i.e., $\mathsf{realizable}((A, C_G)) = \mathsf{false}$. In analogy to Eq. 1, a diagnosis is finally a minimal set $\Delta_G \subseteq \mathsf{COMP}_G$ for which $\mathsf{consistent}_G^{\mathsf{SD}}(\mathsf{COMP}_G \setminus \Delta_G) = \mathsf{true}$ holds.

Observation 1 *Given that* C_G *is a conflict in an unrealizable specification* φ*, at least one guarantee* $g \in C_G$ *must be modified in order to render* φ *realizable.*

If none of the guarantees in a conflict are modified, then the specification is unrealizable, independent of all other guarantees. The reason is that the guarantees in C_G already induce an unrealizable specification, additional guarantees can only make it worse (see Lemma 1).

Observation 2 *A diagnosis* $\Delta_G \subseteq G$ *is a minimal set of guarantees that can be modified in such a way that the specification becomes realizable.*

The guarantees $g \in \Delta_G$ can be modified to $\top$, which gives a realizable specification. By definition, Δ_G is a minimal set for which this is possible.

Example 2. Specification $\varphi_2 = (\{a_1\}, \{g_1, g_2, g_3, g_4\})$ is unrealizable (see Example 1). It contains the minimal conflicts $C_{G,1} = \{g_2, g_4\}$ and $C_{G,2} = \{g_3, g_4\}$. $C_{G,1}$ is a conflict because $(\{a_1\}, \{g_2, g_4\})$ is unrealizable, i.e., no system can implement both

$g_2 = \mathsf{always}(\mathsf{eventually}(\mathsf{not}(h_0)\ \mathsf{or}\ e_0))$ and

$g_4 = \mathsf{always}(p)\ \mathsf{implies}\ \mathsf{always}(\mathsf{not}(e_0)\ \mathsf{and}\ \mathsf{not}(e_1))$.

If h_0 and p are always set, the system must either violate g_4 (if it ever sets e_0) or g_2 (otherwise). $C_{G,1}$ is minimal because $(\{a_1\}, \{g_2\})$ and $(\{a_1\}, \{g_4\})$ are both trivially realizable. Analogously for $C_{G,2}$. There are more conflicts (any superset of $C_{G,1}$ or $C_{G,2}$), but no more minimal ones. The diagnoses for φ_2 are $\{g_4\}$ and $\{g_2, g_3\}$. The set $\{g_2, g_3\}$ is a diagnosis because $(\{a_1\}, \{g_1, \top, \top, g_4\})$, which is equivalent to $(\{a_1\}, \{g_1, g_4\})$, is realizable: An implementation could forbid P_0 and P_1 to ever eat. This also means that g_2 and g_3 can be modified in such a way that the specification becomes realizable, because modifying g_2 and g_3 to $\top$ certainly resolves the unrealizability. There are other, more desirable ways to weaken g_2 and g_3, for instance to $\mathsf{always}(\mathsf{eventually}(p\ \mathsf{or}\ \mathsf{not}(h_0)\ \mathsf{or}\ e_0))$ and $\mathsf{always}(\mathsf{eventually}(p\ \mathsf{or}\ \mathsf{not}(h_1)\ \mathsf{or}\ e_1))$. The set $\{g_4\}$ is a diagnosis because $(\{a_1\}, \{g_1, g_2, g_3\}) = \varphi_1$ is realizable (see Example 1). MBD also tells the user that, for instance, modifying g_1 alone cannot resolve the unrealizability because $\{g_1\}$ is not a diagnosis.

4.2.2 Diagnosis of Output Signals

In the previous section we defined diagnoses to be (sets of) guarantees that can be weakened in order to fix the unrealizability of a specification. In this section we define a formalism to identify output signals that may be overconstrained. We also show how the two approaches can be combined.

Let $\varphi = (A, G)$ be an unrealizable specification of a system with inputs X and outputs Y, and let $\overline{\sigma} = (x_0, y_0)(x_1, y_1) \ldots \in (\mathcal{X} \times \mathcal{Y})^\omega$ be a trace. We define an existential quantification $(A, \exists Y' \,.\, G)$ of the outputs $Y' \subseteq Y$ in the guarantees G of φ in such a way that $\overline{\sigma} \models (A, \exists Y' \,.\, G)$ iff

$$\big(\forall a \in A \,.\, \overline{\sigma} \models a\big) \text{ implies } \big(\forall g \in G \,.\, \overline{\sigma} \models \exists Y' \,.\, g\big).$$

The existential quantification $\exists Y' \,.\, g$ in one single guarantee $g \in G$ is defined as $\overline{\sigma} \models \exists Y' \,.\, g$ iff

$$\exists y_0' y_1' y_2' \ldots \in \left(2^{Y'}\right)^\omega . (x_0, y_0^E)(x_1, y_1^E)(x_2, y_2^E) \ldots \models g,$$

where $y_i^E = (y_i \setminus Y') \cup y_i'$ for all $i \geq 0$.

Informally speaking, a quantification $\exists Y' \,.\, G$ of outputs $Y' \subseteq Y$ in guarantees G removes all restrictions on these outputs. The specification $(A, \exists Y' \,.\, G')$ allows arbitrary values for all signals $y \in Y'$ in all time steps. Also note that the quantification is performed on every single guarantee in isolation, and not for all guarantees simultaneously. With $\mathsf{SD} = (A, G, X, Y)$, $\mathsf{COMP}_Y = Y$ and $Y' \subseteq Y$, we define $\mathsf{consistent}_Y^{\mathsf{SD}}(Y') = \mathsf{realizable}((A, \exists Y \setminus Y' \,.\, G))$. Consequently, a conflict is a set $C_Y \subseteq Y$ of outputs for which $\mathsf{realizable}((A, \exists Y \setminus C_Y \,.\, G)) = \mathsf{false}$ applies. Finally, a diagnosis is a minimal set $\Delta_Y \subseteq Y$ such that $\mathsf{realizable}((A, \exists \Delta_Y \,.\, G)) = \mathsf{true}$. Hence, every diagnosis Δ_Y represents a minimal set of signals that may be overconstrained, because removing restrictions on these signals resolves unrealizability.

An alternative version $\tilde{\exists}$ of the quantification can be defined in such a way that $\overline{\sigma} \models (A, \tilde{\exists} Y' . G)$ iff

$$\big(\forall a \in A . \overline{\sigma} \models a\big) \text{ implies } \overline{\sigma} \models \tilde{\exists} Y' . G,$$

where $\overline{\sigma} \models \tilde{\exists} Y' . G$ iff

$$\exists y_0' y_1' \ldots \in \left(2^{Y'}\right)^{\omega} . \forall g \in G . (x_0, y_0^E)(x_1, y_1^E) \ldots \models g,$$

and y_i^E is again $(y_i \setminus Y') \cup y_i'$ for all $i \geq 0$.

The difference is that $\exists$ is performed on every guarantee separately, while $\tilde{\exists}$ requires the existence of one trace of the quantified signals fulfilling all guarantees.

Example 3. As an illustration, consider the specification $(A, G) = (\emptyset, \{\mathsf{always}(O), \mathsf{always}(\mathsf{not}\, O)\})$, where O is the only output. The specification $(A, \exists\{O\} . G)$ is realizable (it allows all traces), but $(A, \tilde{\exists}\{O\} . G)$ is not. Consequently, $\{O\}$ is a diagnosis when using $\exists$ but not when using $\tilde{\exists}$.

Our approach works for both definitions. However, we decided for $\exists$ because we think that $\{O\}$ should be a diagnosis for the example. After all, if there was no output O, there would also be no conflict. Furthermore, the user can often comprehend what a quantification in one guarantee means. Understanding what a simultaneous quantification in all guarantees means is typically much more difficult, because complex dependencies between the guarantees may exist.

4.2.3 Diagnosis of Guarantees and Outputs

The approaches of diagnosing guarantees and outputs can also be combined by defining $\mathsf{COMP}_{G,Y} = G \cup Y$. For $B \subseteq \mathsf{COMP}_{G,Y}$ we have that $\mathsf{consistent}^{\mathsf{SD}}_{G,Y}(B) = \mathsf{realizable}((A, \exists Y \setminus B . (G \cap B)))$. Moreover, $C_{G,Y} \subseteq (Y \cup G)$ is a conflict iff $\mathsf{realizable}((A, \exists Y \setminus C_{G,Y} . (G \cap C_{G,Y}))) = \mathsf{false}$. A diagnosis is finally a minimal set $\Delta_{G,Y} \subseteq (Y \cup G)$ such that $\mathsf{realizable}((A, \exists (Y \cap \Delta_{G,Y}) . (G \setminus \Delta_{G,Y}))) = \mathsf{true}$. The properties stated in Section 4.2.1 still hold:

Lemma 2. *Function* $\mathsf{consistent}^{\mathsf{SD}}_{G,Y}$ *is monotonic in the sense that, for all* $B'' \subseteq B' \subseteq G \cup Y$*, we have that* $\neg\, \mathsf{consistent}^{\mathsf{SD}}_{G,Y}(B'')$ *implies* $\neg\, \mathsf{consistent}^{\mathsf{SD}}_{G,Y}(B')$*.*

In analogy to the Observations 1 and 2, one has to weaken at least one guarantee or output signal out of every conflict $C_{G,Y}$ in order to obtain a realizable specification. A diagnosis $\Delta_{G,Y}$ is a minimal set of guarantees and output signal definitions that can be modified to obtain a realizable specification.

Theorem 1. *Every diagnosis* Δ_G *and every diagnosis* Δ_Y *for an unrealizable specification* $\varphi = (A, G)$ *is also a diagnosis with respect to the definition of* $\Delta_{G,Y}$*.*

Proof. If $\mathsf{realizable}((A, G \setminus \Delta_G))$ holds, then so does $\mathsf{realizable}((A, \exists (Y \cap \Delta_G) . (G \setminus \Delta_G)))$ because $Y \cap \Delta_G = \emptyset$. Moreover, $\forall \Delta_G' \subset \Delta_G . \neg\, \mathsf{realizable}((A, G \setminus \Delta_G'))$ implies $\forall \Delta_G' \subset \Delta_G . \neg\, \mathsf{realizable}((A, \exists (Y \cap \Delta_G') . (G \setminus \Delta_G')))$ since $Y \cap \Delta_G' = \emptyset$. Hence, any Δ_G is also a diagnosis $\Delta_{G,Y}$. Analogously, since $Y \cap \Delta_Y'' = \Delta_Y''$ and $G \setminus \Delta_Y'' = G$ for all $\Delta_Y'' \subseteq \Delta_Y$, we have that $\mathsf{realizable}((A, \exists \Delta_Y . G))$ implies that $\mathsf{realizable}((A, \exists (Y \cap \Delta_Y) . (G \setminus \Delta_Y)))$ and $\forall \Delta_Y' \subset \Delta_Y . \neg\, \mathsf{realizable}((A, \exists \Delta_Y' . G))$ implies $\forall \Delta_Y' \subset \Delta_Y . \neg\, \mathsf{realizable}((A, \exists (Y \cap \Delta_Y') . (G \setminus \Delta_Y')))$. Therefore, every diagnosis Δ_Y is also a diagnosis $\Delta_{G,Y}$. □

Theorem 1 states that the definition of $\Delta_{G,Y}$ subsumes Δ_G and Δ_Y. Having all diagnoses $\Delta_{G,Y}$, one would not gain further diagnoses by computing Δ_Y and Δ_G. Thus, we will stick to the definition of $\Delta_{G,Y}$ in the following.

Example 4. The minimal conflicts for the unrealizable specification $\varphi_2 = (\{a_1\}, \{g_1, g_2, g_3, g_4\})$ from Example 1 are $\{g_2, g_4, e_0\}$ and $\{g_3, g_4, e_1\}$. Recall that

$$g_2 = \mathsf{always}(\mathsf{eventually}(\mathsf{not}(h_0) \text{ or } e_0)) \text{ and}$$
$$g_4 = \mathsf{always}(p) \text{ implies } \mathsf{always}(\mathsf{not}(e_0) \text{ and } \mathsf{not}(e_1)).$$

The set $\{g_2, g_4, e_0\}$ is a conflict because the specification $(\{a_1\}, \exists e_1 . \{g_2, g_4\})$ is equivalent to

$$(\{a_1\},\ \{\mathsf{always}(\mathsf{eventually}(\mathsf{not}(h_0) \text{ or } e_0)),$$
$$\mathsf{always}(p) \text{ implies } \mathsf{always}(\mathsf{not}\, e_0)\})$$

and thus unrealizable. Similar for $\{g_3, g_4, e_1\}$. The diagnoses are $\{g_4\}$, $\{g_2, g_3\}$, $\{g_2, e_1\}$, $\{e_0, g_3\}$, and $\{e_0, e_1\}$. Note that the diagnoses are exactly the minimal hitting sets for the minimal conflicts, i.e., every diagnosis shares at least one element with every minimal conflict.

4.2.4 Computation of Diagnoses

So far, we have only defined what diagnoses are. This section addresses their computation. We use the algorithm presented by Reiter [41] for the computation of diagnoses. It computes all minimal hitting sets for the collection of conflicts via a hitting set tree. Theorem 2, which is a reformulation of Theorem 4.4 in [41], ensures that this procedure correctly yields all diagnoses.

Theorem 2. *A set* $\Delta_{G,Y} \subseteq G \cup Y$ *is a diagnosis for an unrealizable specification* $\varphi = (A, G)$ *iff it is a minimal hitting set for the collection* $\mathcal{K}$ *of conflicts in* $\varphi = (A, G)$*.*

Proof. ($\Rightarrow$) Let $\Delta_{G,Y}$ be a diagnosis. We have that $\mathsf{consistent}^{\mathsf{SD}}_{G,Y}((G \cup Y) \setminus \Delta_{G,Y})$ holds. Consequently, due to Lemma 2, there is no conflict not containing an element of $\Delta_{G,Y}$. Hence, $\Delta_{G,Y}$ is a hitting set for $\mathcal{K}$. It is a minimal hitting set because for all $\Delta_{G,Y}' \subset \Delta_{G,Y}$ we have that $\neg\, \mathsf{consistent}^{\mathsf{SD}}_{G,Y}((G \cup Y) \setminus \Delta_{G,Y}')$ holds by definition. That is, for all real subsets $\Delta_{G,Y}'$ of $\Delta_{G,Y}$ there exists at least one conflict which does not contain any element of $\Delta_{G,Y}'$. Hence, $\Delta_{G,Y}$ is a minimal hitting set.

($\Leftarrow$) Let $\Delta_{G,Y}$ be a minimal hitting set for $\mathcal{K}$. By definition, $\Delta_{G,Y}$ is also a diagnosis iff $\mathsf{consistent}^{\mathsf{SD}}_{G,Y}((G \cup Y) \setminus \Delta_{G,Y})$ and $\neg\, \mathsf{consistent}^{\mathsf{SD}}_{G,Y}((G \cup Y) \setminus \Delta_{G,Y}')$ hold for all $\Delta'_{G,Y} \subset \Delta_{G,Y}$. Both properties are shown by contradiction. If the former would not hold, there would be a conflict not containing any element of $\Delta_{G,Y}$, so $\Delta_{G,Y}$ would not be a hitting set for $\mathcal{K}$. If the latter would not hold, then $\Delta'_{G,Y} \subset \Delta_{G,Y}$ would a hitting set for $\mathcal{K}$ (see ($\Rightarrow$)), so $\Delta_{G,Y}$ could not be a minimal hitting set. □

Since understanding the hitting set tree algorithm is not vital for the purpose of this article, we refer the reader to [41] for details. Basically, the algorithm only requires a procedure to compute a conflict not containing a certain set of components, if such a conflict exists. This can be implemented with one single realizability check. However, the algorithm performs better if the computed conflicts are minimal. Such a procedure can be defined as $\mathsf{cNot}^{\mathsf{SD}}(B) =$

$$\begin{cases} \text{None} & \text{if } \mathsf{consistent}^{\mathsf{SD}}_{G,Y}(\mathsf{COMP}_{G,Y} \setminus B) \\ \mathsf{min}^{\mathsf{SD}}(\mathsf{COMP}_{G,Y} \setminus B) & \text{otherwise} \end{cases}$$

for $B \subseteq \mathsf{COMP}_{G,Y}$. The function $\mathsf{min}^{\mathsf{SD}}$ computes the minimal conflict. That is, given a set $M \subseteq \mathsf{COMP}_{G,Y}$ such that $\mathsf{consistent}^{\mathsf{SD}}_{G,Y}(M) = \mathsf{false}$, it computes a set $\hat{M} = \mathsf{min}^{\mathsf{SD}}(M)$ such that $\mathsf{consistent}^{\mathsf{SD}}_{G,Y}(\hat{M}) = \mathsf{false}$, and for all $M' \subset \hat{M}$ it holds that $\mathsf{consistent}^{\mathsf{SD}}_{G,Y}(M') = \mathsf{true}$. Function $\mathsf{min}^{\mathsf{SD}}$ can be implemented as $\mathsf{min}^{\mathsf{SD}}(M) = \mathsf{ddmin}^{\mathsf{test}^{\mathsf{SD}}}(M)$ or $\mathsf{min}^{\mathsf{SD}}(M) = \mathsf{linMin}^{\mathsf{test}^{\mathsf{SD}}}(M)$, in which $\mathsf{test}^{\mathsf{SD}}(M') = \text{✘}$ iff $\neg\,\mathsf{consistent}^{\mathsf{SD}}_{G,Y}(M')$.

4.2.5 Performance Optimizations

Model-based diagnosis for unrealizability requires the computation of many unrealizable cores, which in turn requires many realizability checks. Hence, it is important that these operations are implemented efficiently.

The first factor which influences performance is the minimization algorithm that is used for unrealizable core computation. We propose to use Delta Debugging, as it performs much better than linMin in our experiments. (See Section 7.) Furthermore, it is important to exploit the monotonicity of test (cf. Lemma 2) to speed up the minimization as suggested in [50]: all sets M' for which $\mathsf{test}^{\mathsf{SD}}(M')$ returned ✘ are stored. Whenever a subset M'' of a stored set M' is tested, $\mathsf{test}^{\mathsf{SD}}(M'') \neq$ ✘ can be concluded without actually invoking the realizability check.

As a second performance optimization, we propose to use approximations of realizability. We call a procedure $\mathsf{realizable}_O$ an over-approximation of realizability iff for all specifications φ we have that $\mathsf{realizable}(\varphi)$ implies $\mathsf{realizable}_O(\varphi)$. Similarly, $\mathsf{realizable}_U$ is an under-approximation of realizability if $\mathsf{realizable}_U(\varphi)$ implies $\mathsf{realizable}(\varphi)$ for all φ. If the specification can be transformed into a game $\mathcal{G} = (Q, \Sigma, T, q_0, \mathsf{Win})$, then realizability can be decided by computing the winning region $W_{\text{sys}} \subseteq Q$ of the system and testing whether $q_0 \in W_{\text{sys}}$. An over-approximation for realizability can be defined by computing an over-approximation W_O of W_{sys} (such that $W_O \supseteq W_{\text{sys}}$), and checking whether $q_0 \in W_O$. An under-approximation can be defined analogously. Suppose that we are able to find approximations of realizability which are both fast to compute and close to the exact definition of realizability. Then we can use them to define

$$\mathsf{realizable}_E(\varphi) = \begin{cases} \mathsf{true} & \text{if } \mathsf{realizable}_U(\varphi) \\ \mathsf{false} & \text{else if } \neg\,\mathsf{realizable}_O(\varphi) \\ \mathsf{realizable}(\varphi) & \text{otherwise} \end{cases}$$

as an often faster decision procedure for realizability. Different approximations can be applied in different order.

An over-approximation of realizability can also be used to compute unrealizable cores in a two step approach. First, the unrealizable specification is minimized with $\mathsf{realizable}_O$. This gives an over-approximation of a minimal unrealizable core. In a second step, the approximate minimal core is further minimized with the exact definition of realizability, resulting in an exact minimal core. This two-step approach can be faster because ideally the expensive exact checks are performed on relatively small subsets of the specification only. More formally, we suggest to compute

$$\hat{M} = \mathsf{min}^{\mathsf{SD}}(M) = \mathsf{linMin}^{\mathsf{test}^{\mathsf{SD}}_2}\left(\mathsf{ddmin}^{\mathsf{test}^{\mathsf{SD}}_1}(M)\right) \text{ with}$$

$\mathsf{test}^{\mathsf{SD}}_1(M') = \text{✘}$ iff $\neg\,\mathsf{realizable}_O((A, \exists(Y \setminus M') \,.\, G \cap M'))$, $\mathsf{test}^{\mathsf{SD}}_2(M') = \text{✘}$ iff $\neg\,\mathsf{realizable}_E((A, \exists(Y \setminus M') \,.\, G \cap M'))$, and $M' \subseteq M \subseteq G \cup Y$. We use linMin as minimization algorithm for the second step, because ddmin does not perform well when executed on an almost minimal set: In the early phase, all attempts to remove big chunks of the set fail, so a lot of checks are wasted until the granularity is high enough. With high granularity, ddmin behaves similarly to linMin.

4.3 Explaining Conflicts with Counterstrategies

In the previous section, we discussed how model-based diagnosis can be used to identify possible error locations by identifying output signals and guarantees that can be weakened in order to make the specification unrealizable. However, there may be many such diagnoses and there are also many ways to weaken guarantees and restrictions on outputs. In order to find the best suitable fix, the user has to understand the problem in the specification. We assist the user in achieving this by explaining conflicts, as computed by the diagnosis algorithm, with counterstrategies.

4.3.1 Debugging Flow

Fig. 6 depicts the flow of our method to explain conflicts, and thereby the root causes of unrealizability. The focus is on obtaining *simple* explanations. Minimal conflicts as computed by our diagnosis algorithm are unrealizable

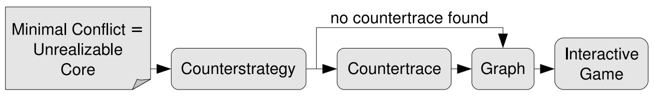


Fig. 6: Our procedure to explain conflicts in a specification.

cores, i.e., parts of the specification which are unrealizable on their own. First, a counterstrategy is computed not for the original specification but for an unrealizable core. Next, we use a heuristic algorithm to further simplify the counterstrategy to a countertrace, i.e., a single trace of inputs that suffices to illustrate unrealizability. Finally, we use this countertrace (or the counterstrategy, in case our heuristic fails) in an interactive diagnostic game against the user, and to compute a graph which summarizes all plays that are possible in this game. The next subsections will detail these steps.

4.3.2 Unrealizable Cores

Understanding the contradiction in a large unrealizable specification may be difficult, but often only a small part of the specification is responsible for the contradiction. Removing the rest gives a specification which still contains the contradiction, but is much smaller and thus easier to debug. A part of the specification which is unrealizable on its own right is called an unrealizable core [11]. For an unrealizable specification $\varphi = (A, G)$ with inputs X and outputs Y, a minimal conflict $C_{G,Y}$, as computed in our diagnosis approach, represents the unrealizable core $(A, \exists Y \setminus C_{G,Y} \,.(G \cap C_{G,Y}))$. There are two important differences to unrealizable cores as defined in [11].

First, we do not remove environment assumptions. The reason is not only that Prop. 1 hinders us from defining assumptions to be components. Removing environment assumptions would also confuse the user in subsequent steps of our debugging process: the counterstrategy, computed to explain the unrealizable core, could exhibit behavior which the user originally forbade. This would allow cheating by the environment in the diagnostic game, at least from the user's point of view.

Second, using existential quantification, we not only remove properties but also output signals from the specification. The meaning is that the core is still unrealizable, even though the system can set all existentially quantified outputs arbitrarily. Again, the reason for removing outputs is not only that we wanted to identify overconstrained signals when doing diagnosis. Removing outputs also allows the user to focus on the remaining signals, which makes the diagnostic game much simpler. The quantified outputs are not even included in the game, because they can be set arbitrarily. In fact, we found that minimizing only guarantees often makes the diagnostic game more difficult. This is due to the fact that removing guarantees gives the system more freedom. Thus, there are more possible moves and more plays to explore in the diagnostic game. Removing outputs counteracts by providing the user with choices for relevant signals only.

Theorem 3. *Let $\varphi = (A, G)$ be an unrealizable specification, and let $\varphi' = (A, \exists Y' . G')$, with $Y' \subseteq Y$ and $G' \subseteq G$, be an unrealizable core thereof. Any implementation M^o of a counterstrategy for φ' implements a counterstrategy for φ as well.*

Proof. Since M^o implements a counterstrategy for φ', $\overline{\sigma} \not\models \varphi'$ holds for all $\overline{\sigma} \in L(M^o)$. Clearly, φ is stricter than φ', i.e., $\overline{\sigma} \not\models \varphi'$ implies $\overline{\sigma} \not\models \varphi$ for all $\overline{\sigma} \in \Sigma^\omega$. Therefore, $\overline{\sigma} \not\models \varphi$ for all $\overline{\sigma} \in L(M^o)$, which means that M^o implements a counterstrategy for φ. □

Theorem 3 states that computing a counterstrategy for an unrealizable core is indeed useful, because any implementation of the counterstrategy can also explain the unrealizability of the original specification.

Example 5. The minimal conflict $\{g_2, g_4, e_0\}$ (see Example 4) in specification $\varphi_2 = (\{a_1\}, \{g_1, g_2, g_3, g_4\})$ represents the unrealizable core $\varphi' = (\{a_1\}, \exists e_1 \,.\{g_2, g_4\})$. This means that φ_2 is unrealizable even if g_1 and g_3 do not have to be fulfilled and e_1 can be chosen completely arbitrarily in all time steps. In the diagnostic game, the user can focus on setting e_0 in such a way that

$$\varphi' = (\{a_1\},\ \{\mathsf{always}(\mathsf{eventually}(\mathsf{not}(h_0)\ \mathsf{or}\ e_0)),\ \mathsf{always}(p)\ \mathsf{implies}\ \mathsf{always}(\mathsf{not}\ e_0)\})$$

is fulfilled, i.e., she can focus on properties and signals that actually contribute to the conflict. Nevertheless, she will not win the game because a counterstrategy could set p and h_0 in all steps. This is also a valid counterstrategy for φ_2. The unrealizable core just makes the task of the user easier.

4.3.3 Countertraces

One of the main sources for complexity in the diagnostic game is that the environment behavior, defined by the counterstrategy, depends on the previous behavior of the system. The user would prefer one single trace $\overline{x} \in \mathcal{X}^\omega$ of inputs such that no output trace $\overline{y} \in \mathcal{Y}^\omega$ can make $(\overline{x}||\overline{y})$ fulfill the specification. We call such a trace $\overline{x}$ a *countertrace*. A countertrace significantly reduces the effort of understanding which environment behavior causes problems.

Unfortunately, a countertrace does not always exist. Consider the specification $(\emptyset, \{\mathsf{always}(y\ \mathsf{iff}\ \mathsf{next}\, x)\})$, where x and y are Boolean inputs and outputs, respectively. The specification is unrealizable, because the system would have to look into the future to comply with it. However, for every input trace there is also an output trace such that the specification is fulfilled (namely the input trace shifted by one step into the future) [42, 37]. As a second difficulty, computing countertraces is expensive. In general, we can compute a countertrace by existentially quantifying outputs from the game automaton, and complementing the resulting automaton. Every trace in the language of the complemented automaton is a valid countertrace. However, complementation would cause an exponential blow-up since the automaton may be non-deterministic after quantification. We consider this unacceptable, and therefore define a heuristic algorithm. It does not always find a countertrace, even if one exists. However, our experiments show that it is fast and often successful.

Our heuristic takes a counterstrategy $\mathcal{S} = (\mathit{\Gamma}, \gamma_0, \rho)$ with $\rho \subseteq (Q\times\mathit{\Gamma}\times\mathcal{X}\times\mathit{\Gamma})$ for game $\mathcal{G} = (Q, \mathit{\Sigma}, T, q_0, \mathsf{Win})$, constructed from an unrealizable specification φ. The construction of these elements depends on the specification language used. For every state $q \in Q$ and memory content $\gamma \in \mathit{\Gamma}$, the counterstrategy defines possible inputs $x \in \mathcal{X}$ and next memory contents $\gamma' \in \mathit{\Gamma}$ by enforcing that $(q, \gamma, x, \gamma') \in \rho$. After the selection of such an input, the output is chosen by the system. The idea of our heuristic is that we choose from the inputs allowed by the counterstrategy in such a way that the next choice of an input is independent of the choice of the output.

We compute the countertrace $\overline{\tau} = \tau_0\tau_1 \ldots \in \mathcal{X}^\omega$ and a sequence $S_0S_1 \ldots$ of sets $S_i \subseteq (Q \times \mathit{\Gamma})$ in parallel. Every S_i contains all pairs of state and memory content which are possible after the finite input sequence $\tau_0 \ldots \tau_{i-1}$. The computation starts with $S_0 = \{(q_0, \gamma_0)\}$ and proceeds with

$$S_{i+1} = \big\{(q', \gamma') \mid \exists(q, \gamma) \in S_i, y \in \mathcal{Y} \,. \\ q' = T(q, (\tau_i, y)) \wedge (q, \gamma, \tau_i, \gamma') \in \rho\big\},$$

where τ_i is chosen arbitrarily from the set $T_i = \{\tau \in \mathcal{X} \mid \forall(q, \gamma) \in S_i \,.\, \exists\gamma' \in \mathit{\Gamma} \,.(q, \gamma, \tau, \gamma') \in \rho\}$. The set T_i contains all inputs which conform to the counterstrategy, independent of the current state and memory content $(q, \gamma) \in S_i$. We choose such an input $\tau_i \in T_i$ for the countertrace, compute with S_{i+1} all state-memory pairs in which we might end up in the next step, and repeat the procedure. This gives an input trace $\overline{\tau} = \tau_0\tau_1 \ldots$ which conforms to the counterstrategy, independent of the outputs chosen by the system.

If $T_i = \emptyset$ for any i, our heuristic fails. We can stop the computation successfully at index k if $S_k \subseteq S_j$ for some $j < k$. This is because $S_k \subseteq S_j$ implies $T_k \supseteq T_j$, and thus, τ_k can be set to τ_j. Since the definition of S_{i+1} is monotonic in S_i (when used with the same τ_i), we have that $S_{k+1} \subseteq S_{j+1}$, which implies $T_{k+1} \supseteq T_{j+1}$ and allows $\tau_{k+1} = \tau_{j+1}$. This can be repeated. We obtain a lasso-shaped countertrace with finite stem $\tau_0 \ldots \tau_{j-1}$ followed by infinitely many repetitions of $\tau_j \ldots \tau_{k-1}$. A symbolic implementation (e.g., using BDDs) of the computation is straightforward if $\mathcal{G}$ and $\mathcal{S}$ are represented symbolically.

The number of iterations is equal to the length (stem plus loop) of the computed countertrace. In the worst case, our heuristic requires $2^{|Q\times\mathit{\Gamma}|} - 1$ iterations. However, in all our experiments the length of the countertrace was below 10. The intuitive explanation of the good performance in practice is that existing synthesis algorithms often yield strategies that respond within short time, and the computed countertrace can only exhibit behavior allowed by the counterstrategy.

Even if a countertrace exists, our algorithm may be unable to find one. There are two reasons. First, it may fail due to a bad choice of an input letter $\tau_i \in T_i$. This problem can be solved with backtracking. The second reason is that the counterstrategy from which the countertrace is created may not contain all possible ways to force the system to violate the specification.

A countertrace with finite stem $\tau_0 \ldots \tau_{j-1}$ and infinite loop $\tau_j \ldots \tau_{k-1}$ can be interpreted as a strategy $\mathcal{S}_{\overline{\tau}} = (\mathit{\Gamma}_{\overline{\tau}}, 0, \rho_{\overline{\tau}})$ with $\mathit{\Gamma}_{\overline{\tau}} = \{0, \ldots, k-1\}$ and

$$\rho_{\overline{\tau}} = \big\{(q, \gamma, \tau_\gamma, next(\gamma)) \in (Q, \mathit{\Gamma}_{\overline{\tau}}, \mathcal{X}, \mathit{\Gamma}_{\overline{\tau}})\big\}, \text{ where}$$
$$next(\gamma) = \begin{cases} j & \text{if } \gamma = k-1, \text{ and} \\ \gamma+1 & \text{otherwise.} \end{cases}$$

Theorem 4. *Every play $\overline{\pi}$ that conforms to the countertrace $\overline{\tau}$, i.e., which conforms to the strategy $\mathcal{S}_{\overline{\tau}} = (\mathit{\Gamma}_{\overline{\tau}}, 0, \rho_{\overline{\tau}})$ is won by the environment.*

Proof. The inputs τ_i dictated by countertrace $\overline{\tau}$ and also by $\rho_{\overline{\tau}}$ are (singleton) subsets of the inputs that are allowed by the counterstrategy $\mathcal{S} = (\mathit{\Gamma}, \gamma_0, \rho)$. This follows trivially from the construction of $\overline{\tau}$ and $\mathcal{S}_{\overline{\tau}}$. Since all plays conforming to $\mathcal{S}$ are won by the environment, so are all plays conforming to $\overline{\tau}$ and $\mathcal{S}_{\overline{\tau}}$. □

Example 6. We apply our algorithm to compute a countertrace for the unrealizable core $(\{a_1\}, \exists e_1 \,.\{g_2, g_4\}) =$

$$(\{a_1\},\ \{\mathsf{always}(\mathsf{eventually}(\mathsf{not}(h_0) \text{ or } e_0)), \\ \mathsf{always}(p) \text{ implies } \mathsf{always}(\mathsf{not}\, e_0)\}),$$

which was introduced in Example 5. Due to space constraints, we neither provide a full game definition $\mathcal{G}$ nor a full counterstrategy definition $\mathcal{S}$. Instead, we present parts of these elements in a way that best serves the explanation. Our algorithm is illustrated in Fig. 7. It starts with $S_0 = \{(q_0, \gamma_0)\}$. From there, the counterstrategy only allows the input letter $(h_0, h_1, p) = (0, 0, 1)$. Hence, $T_0 = \{(0, 0, 1)\}$ and τ_0 can only be set to $(0, 0, 1)$. With input $(h_0, h_1, p) = (0, 0, 1)$, the system has two possibilities. If it sets $e_0 = 0$, the play will get to state-memory pair (q_2, γ_1). With $e_0 = 1$ it goes to (q_1, γ_1). After one step, the play is either in (q_2, γ_1) or (q_1, γ_1), so $S_1 = \{(q_2, \gamma_1), (q_1, \gamma_1)\}$. From (q_2, γ_1), the counterstrategy allows two input letters, namely $(h_0, h_1, p) = (1, 1, 1)$ and $(1, 0, 1)$. From (q_1, γ_1) only $(1, 0, 1)$ is allowed. T_1 consists of input letters that are allowed from *all* elements of S_1, so $T_1 = \{(1, 0, 1)\}$. Consequently, there is no other possibility than setting $\tau_1 = (1, 0, 1)$. With input $(1, 0, 1)$ in Step 1, (q_1, γ_1) and (q_2, γ_1) are the only possible successors. Hence, $S_2 = \{(q_1, \gamma_1), (q_2, \gamma_1)\}$. Since

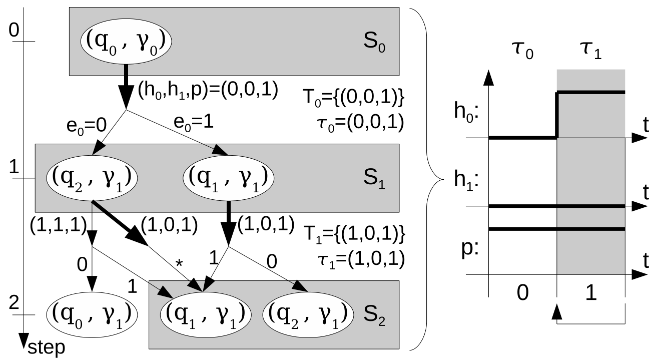


Fig. 7: An example illustrating countertrace computation.

$S_2 = S_1$, the algorithm terminates successfully. This is possible because all τ_i with $i \geq 2$ could be set to τ_1, with the consequence that all encountered sets S_i with $i > 2$ would be equal to S_1. Therefore, the countertrace is τ_0 followed by infinitely many repetitions of τ_1. This is depicted on the right-hand side of Fig. 7.

4.4 Interactive Diagnostic Games and Graphs

We allow the user to play an interactive diagnostic game against the counterstrategy. In every step, the counterstrategy provides inputs and the user responds with outputs. The user attempts to fulfill the specification but fails for sure. Playing the game (and losing repeatedly) will help the user understand why she cannot win, i.e., why the specification is unrealizable. If a countertrace is found, the user plays against this trace. The entire trace is shown right from the beginning to clearly set out the problematic environment behavior.

In order to speed up the learning process, we also provide the user with a graph $\mathbb{G}$ which summarizes all plays that are possible against the counterstrategy. Every node in the graph corresponds to a reachable state-memory pair $(q, \gamma) \in (Q, \Gamma)$. There is a special start node (q_0, γ_0). Directed edges between nodes are created if the corresponding transition is possible. Edges are labeled with the inputs and outputs that enable the transition. The graph can serve as a "cheat sheet" for the diagnostic game. The user can see already in advance how the counterstrategy will react to her moves. Hence, she might discard some moves without trying them in the game. This reduces the time necessary to understand the cause for unrealizability.

5 Debugging GR(1) Specifications

In this section, we instantiate our generic debugging approach for the class of Generalized Reactivity(1) [39] (GR(1)) specifications. A GR(1) specification defines assumptions and guarantees using DBWs. All the prerequisites for applying our debugging approach are satisfied: Guarantees can be added and removed by adding or removing DBWs, realizability can be decided as shown in [39], a GR(1) specification can be turned into a game as described in Section 3.5, and outputs can be existentially quantified by quantifying them in the symbolic representation of the DBWs. Our approach for debugging undesired behavior furthermore requires that the modified simulation trace representing the desired behavior can be turned into a guarantee. Creating a DBW for such a trace is straightforward [29]. There is, to our knowledge, no prior work on how to compute a counterstrategy for an unrealizable GR(1) specification. Hence, we will detail this below. Furthermore, we discuss some GR(1) specific aspects of the interactive diagnostic game as well as the performance optimizations using approximations of realizability.

5.1 Counterstrategies for GR(1) specifications

Piterman et al. [39] propose to transform a GR(1) specification into a game $\mathcal{G}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathsf{Win})$. They then show how to derive a winning strategy for the system from intermediate results in the computation of the winning region $W^{\mathrm{GR1}}_{\mathrm{sys}}$. We follow this approach and derive a counterstrategy from intermediate results in the computation of $W^{\mathrm{GR1}}_{\mathrm{env}} = Q \setminus W^{\mathrm{GR1}}_{\mathrm{sys}}$. Complementing Eq. 6, we obtain

$$W^{\mathrm{GR1}}_{\mathrm{env}} = \mu Z \,.\, \bigcup_{j=1}^{n} \nu Y \,.\, \bigcap_{i=1}^{m} \mu X \,.$$
$$\big(\neg J^s_j \cup \mathsf{MX}^e(Z)\big) \cap \mathsf{MX}^e(Y) \cap \big(J^e_i \cup \mathsf{MX}^e(X)\big)\,. \quad (7)$$

As intermediate results for counterstrategy construction we define Z_a to be the a-th iterate (according to Eq. 3) of the fixpoint in Z. Furthermore, we define $Y_{a,j}$ as

$$\nu Y \,.\, \bigcap_{i=1}^{m} \mu X \,.$$
$$\big(\neg J^s_j \cup \mathsf{MX}^e(Z_{a-1})\big) \cap \mathsf{MX}^e(Y) \cap \big(J^e_i \cup \mathsf{MX}^e(X)\big)\,.$$

Finally, $X_{a,j,i,c}$ denotes the c-th iterate of X in

$$\mu X \,.\, \big(\neg J^s_j \cup \mathsf{MX}^e(Z_{a-1})\big) \cap \mathsf{MX}^e(Y_{a,j}) \cap \big(J^e_i \cup \mathsf{MX}^e(X)\big)\,.$$

To simplify notation, we introduce $Z^{\mathrm{new}}_a = Z_a \setminus Z_{a-1}$, $X^{\mathrm{new}}_{a,j,i,c} = X_{a,j,i,c} \setminus X_{a,j,i,c-1}$, and $i \oplus 1 = (i \mod m) + 1$. For $x \in \mathcal{X}$ and $P \subseteq Q$ we also define $\mathsf{MX}^e_x(P) = \{q \in Q \mid \forall y \in \mathcal{Y} \,.\, T(q,(x,y)) \in P\}$ to denote the set of all states from which the environment can force the play into a state of P with input x.

We use $\Gamma = \mathcal{I} \times \mathcal{J}$ as memory for the counterstrategy (Γ, γ_0, ρ). The set $\mathcal{I} = \{1, \ldots, m\}$ is used to store the index of the set J^e_i of accepting states of the environment that the counterstrategy attempts to reach next. The set $\mathcal{J} = \{0, 1, \ldots, n\}$ is for storing the index of the set J^s_j of accepting states of the system that the counterstrategy tries to evade. The special value 0 indicates that the counterstrategy has not (yet) selected such a set. The initial memory content is $\gamma_0 = (1, 0)$.

We compose the counterstrategy's relation ρ of four parts $\rho_1, \rho_2, \rho_3, \rho_4 \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$.

Sub-strategy ρ_1 is defined as $\rho_1 =$

$$\big\{(q, (i, j), x, (i, 0)) \bigm| \exists a \geq 2 \,.\, q \in Z^{\text{new}}_a \cap \mathsf{MX}^e_x(Z_{a-1})\big\}.$$

It forces the play into a smaller iterate of Z. It cannot know which J^s_j can be evaded in the next step because the next move of the system can influence to which $Y_{a-1,j}$ the play proceeds. Thus, the counterstrategy cannot set the memory $j \in \mathcal{J}$ to a suitable value. In order to remember to do so in the next step, j is set to 0.

Sub-strategy ρ_2 is applied whenever $j = 0$ and ρ_1 is not applicable. It sets j to a suitable value:

$$\begin{aligned}\rho_2 =\big\{(q, (i, 0), x, (i, j')) \bigm| \exists a \geq 1 \,.\\ q \in Z^{\text{new}}_a \cap \mathsf{MX}^e_x(Y_{a,j'}) \setminus \mathsf{MX}^e(Z_{a-1})\big\}\end{aligned}$$

Sub-strategy ρ_3 is applied if the play is in a state of J^e_i. The next goal is to reach a state of $J^e_{i\oplus 1}$, so the memory content of i is updated:

$$\begin{aligned}\rho_3 =\big\{(q, (i, j), x, (i \oplus 1, j)) \bigm| j \neq 0 \wedge q \in J^e_i \wedge\\ \exists a \geq 1 \,.\, q \in Z^{\text{new}}_a \cap \mathsf{MX}^e_x(Y_{a,j}) \setminus \mathsf{MX}^e(Z_{a-1})\big\}\end{aligned}$$

Sub-strategy ρ_4 is used when the set J^e_i is not yet reached. It is an attractor strategy forcing the play ever closer to J^e_i:

$$\begin{aligned}\rho_4 =\big\{(q, (i, j), x, (i, j)) \bigm| j \neq 0 \wedge \exists a \geq 1, c \geq 2 \,.\\ q \in Z^{\text{new}}_a \cap X^{\text{new}}_{a,j,i,c} \cap \mathsf{MX}^e_x(X_{a,j,i,c-1}) \setminus \mathsf{MX}^e(Z_{a-1})\big\}\end{aligned}$$

Theorem 5. *The tuple* $\mathcal{S}^{\text{GR1}} = \big(\mathcal{I} \times \mathcal{J}, (1, 0), \rho^{\text{GR1}}\big)$, *where* $\rho^{\text{GR1}} = \rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4$, *is a counterstrategy in the GR(1) game* $\mathcal{G}^{\text{GR1}}$.

Proof. We need to shown that every play conforming to $\mathcal{S}^{\text{GR1}}$ is won by the environment. This is done inductively. As the base case, we show that any play conforming to $\mathcal{S}^{\text{GR1}}$ is won if it ever reaches Z_1. As inductive step, we show that any play conforming to $\mathcal{S}^{\text{GR1}}$ is won by the environment if it ever reaches a state of Z_a with $a > 1$, assuming that it is won if it ever reaches Z_{a-1}.

Base Case: The set Z_1 is the union of all $Y_{1,j}$, so every state of Z_1 is also in some set $Y_{1,j}$. We have that

$$Y_{1,j} = \bigcap_{i=1}^{m} \bigcup_{c} X_{1,j,i,c}.$$

Hence, for every i, every state of every set $Y_{1,j}$ is also in some iterate $X_{1,j,i,c}$. Since $Z_0 = \emptyset$ we have that $X_{1,j,i,c}$ is the c-th iterate of the fixpoint $\mu X \,.\, \neg J^s_j \cap \mathsf{MX}^e(Y_{1,j}) \cap (J^e_i \cup \mathsf{MX}^e(X))$. That is, the iterates $X_{1,j,i,c}$ are constructed in such a way that a state of J^e_i can be reached from a state of $X_{1,j,i,c}$ in at most $c-2$ steps while never visiting a state of J^s_j. It is possible to choose inputs in such a way that the play proceeds to the next lower iterate in every step, and this is exactly what ρ_4 does by requiring that $q \in \mathsf{MX}^e_x(X_{a,j,i,c-1})$. The prerequisite is that memory j is set such that the play is always in $Y_{1,j}$. This is done by ρ_2 initially. As explained below, ρ_3 and ρ_4 ensure that $Y_{1,j}$ is never left, so they keep j unchanged. Eventually $X_{1,j,i,1} = \neg J^s_j \cap \mathsf{MX}^e(Y_{1,j}) \cap J^e_i$ is reached. The term $\mathsf{MX}^e(Y_{1,j})$ ensures that the environment can force the play into a state of $Y_{1,j}$ again. This is done by ρ_3, which also updates i to $i \oplus 1$. This update ensures that ρ_4 will be able to force the play into a state of $\neg J^s_j \cap \mathsf{MX}^e(Y_{1,j}) \cap J^e_{i\oplus 1}$ in a finite number of steps. As time goes by, all sets J^e_i will be visited repeatedly, while never visiting J^s_j. Hence, ρ_3 and ρ_4 ensure that if a play ever enters Z_1, it is won.[3]

Inductive Step: Suppose the play has reached Z_a with $a > 1$. The only difference to the base case is that $Z_{a-1} \neq \emptyset$. Thus, in comparison to the iterates $X_{1,j,i,c}$, the iterates $X_{a,j,i,c}$ may contain J^s_j states, but only if they are also in $\mathsf{MX}^e(Z_{a-1})$. Hence, it is either possible to apply ρ_3 and ρ_4 forever, thereby avoiding one set J^s_j and visiting all sets J^e_i repeatedly as described for the base case, or to take the play into Z_{a-1} with ρ_1 eventually. From Z_{a-1}, the play is won by induction hypothesis. This concludes the proof. □

5.2 Interactive Diagnostic Games and Graphs

The purpose of the diagnostic games and graphs is to illustrate unrealizability to the user. In the case of a GR(1) specification, in addition to signal values, we also present the current memory content $(i, j) \in (\mathcal{I} \times \mathcal{J})$ of the counterstrategy. This information allows the user to play more effective against the counterstrategy, and to gain additional insights. If the user knows the index i of the set J^e_i which the environment tries to reach, she can try to get around this set. Knowing the index j of the set J^s_j which the environment evades, the user can focus on reaching this set. (The user might indeed reach it. However, the counterstrategy will then force the play into a smaller iterate of Z.) We also print the iterate of Z in which the play currently is. Being in Z^{new}_a, the user knows that memory j will change at most $a-1$ times (not counting changes to the special value 0) in the future, and that she will be able to reach the set J^s_j, which the environment evades, at most $a-1$ times. In particular, she cannot reach J^s_j in Z_1.

The memory content of the counterstrategy can be presented even if the user plays against the countertrace:

[3] Whatever happened in the finite period before entering Z_1 is irrelevant for the winning condition. In particular, every set J^s_j may have been visited a finite number of times.

The entire countertrace is printed right from the beginning. In every step, the counterstrategy is applied to obtain its next memory content. From the different moves allowed by the counterstrategy, the one following the countertrace is selected. This is always possible since the countertrace is constructed in such a way that the inputs dictated by the countertrace are (singleton) subsets of the inputs that are allowed by the counterstrategy.

5.3 Performance Optimizations

In this section, we define approximations of realizability in order to speed up model-based diagnosis as explained in Section 4.2.5. Recall that a GR(1) specification is realizable iff $q_0 \in W^{\mathrm{GR1}}_{\mathrm{sys}}$ in the corresponding game $\mathcal{G}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathsf{Win})$. In order to obtain approximations to this procedure, we define the following subsets of $W^{\mathrm{GR1}}_{\mathrm{sys}}$ or $W^{\mathrm{GR1}}_{\mathrm{env}}$:

$$\begin{aligned}
A^{\mathrm{GR1}}_{\mathrm{sys}} &= \mu X \,.\, A' \cup \mathsf{MX}^s(X) \subseteq W^{\mathrm{GR1}}_{\mathrm{sys}} \quad \text{with} \\
A' &= \bigcup_{i=1}^{m} \nu Y \,.\, \neg J^e_i \cap \mathsf{MX}^s(Y), \\
B^{\mathrm{GR1}}_{\mathrm{sys}} &= \mu X \,.\, B' \cup A^{\mathrm{GR1}}_{\mathrm{sys}} \cup \mathsf{MX}^s(X) \subseteq W^{\mathrm{GR1}}_{\mathrm{sys}} \quad \text{with} \\
B' &= \nu Y \,.\, \mathsf{MX}^s(Y) \cap \bigcap_{j=1}^{n} J^s_j \text{ , and} \\
C^{\mathrm{GR1}}_{\mathrm{env}} &= \mu X \,.\, C' \cup \mathsf{MX}^e(X) \subseteq W^{\mathrm{GR1}}_{\mathrm{env}} \quad \text{with} \\
C' &= \bigcup_{j=1}^{n} \nu Y \,.\, \mathsf{MX}^e(Y) \cap \neg J^s_j \cap \bigcap_{i=1}^{m} J^e_i .
\end{aligned}$$

From all states of A', the system can enforce that the set J^e_i is never visited for some i. The set $A^{\mathrm{GR1}}_{\mathrm{sys}} \subseteq W^{\mathrm{GR1}}_{\mathrm{sys}}$ contains all states from which a state of A' can be reached in a finite number of steps, i.e., all states from which the system can enforce the violation of an environment assumption. The set B' consists of all states from which the system can enforce to fulfill all guarantees by staying in the intersection of all J^s_j forever. $B^{\mathrm{GR1}}_{\mathrm{sys}} \subseteq W^{\mathrm{GR1}}_{\mathrm{sys}}$ comprises all states from which the system can ensure to reach a state of $B' \cup A^{\mathrm{GR1}}_{\mathrm{sys}}$ in a finite number of steps. With C' we compute all states from which the environment can keep the play in the intersection of all J^e_i but outside one particular J^s_j. This is sufficient to enforce a violation of the specification. The set $C^{\mathrm{GR1}}_{\mathrm{env}} \subseteq W^{\mathrm{GR1}}_{\mathrm{env}}$ contains the states from which the environment can enforce to reach C'. We conducted experiments with several more sets, but the listed ones turned out to be both fast to compute and close to $W^{\mathrm{GR1}}_{\mathrm{sys}}$ or $W^{\mathrm{GR1}}_{\mathrm{env}}$.

With $\mathsf{realizable}^{\mathrm{GR1}}(\varphi) = \mathsf{true}$ iff $q_0 \in W^{\mathrm{GR1}}_{\mathrm{sys}}$ in $\mathcal{G}^{\mathrm{GR1}}$, we define a more efficient implementation as

$$\mathsf{realizable}^{\mathrm{GR1}}_E(\varphi) = \begin{cases} \mathsf{true} & \text{if } q_0 \in A^{\mathrm{GR1}}_{\mathrm{sys}} \\ \mathsf{true} & \text{else if } q_0 \in B^{\mathrm{GR1}}_{\mathrm{sys}} \\ \mathsf{false} & \text{else if } q_0 \in C^{\mathrm{GR1}}_{\mathrm{env}} \\ \mathsf{realizable}^{\mathrm{GR1}}(\varphi) & \text{otherwise.} \end{cases}$$

As an over-approximation we define

$$\mathsf{realizable}^{\mathrm{GR1}}_O(\varphi) = \mathsf{true} \text{ iff } q_0 \notin C^{\mathrm{GR1}}_{\mathrm{env}}.$$

This over-approximation as well as the more efficient implementation of the realizability check can be used to speed up computations as described in Section 4.2.5.

6 Implementation

We have implemented our debugging method for GR(1) specifications in the *Requirements Analysis Tool with Synthesis* RATSY [2], a successor of RAT [38]. RATSY is available for download[4] and use under the LGPL[5] open source license. This description refers to version 2.1.0.

The user can enter properties in PSL syntax or by drawing DBWs. A DBW can contain template parameters, allowing the user to instantiate it several times in different form. We think that DBWs are often easier to understand and maintain than PSL formulas, especially for non-experts. Additionally, they can be used to visualize state information while simulating or debugging.

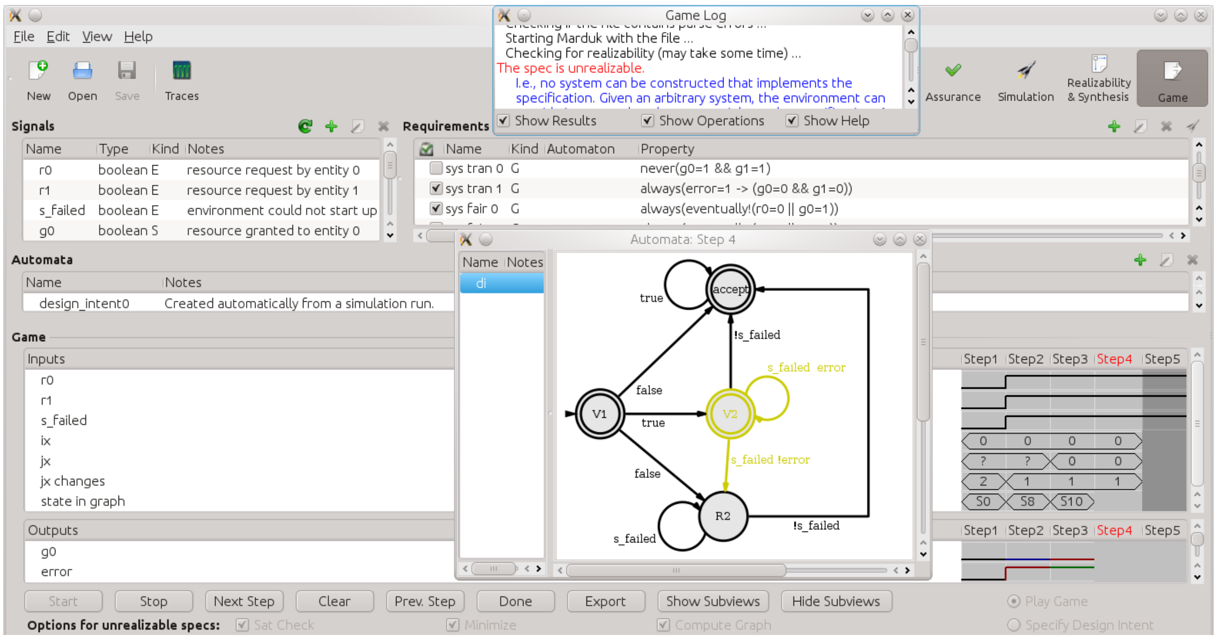
Fig. 8 shows a snapshot of RATSY's GUI when playing a diagnostic game against a countertrace for an unrealizable specification. The entire history of the play is shown with signal waveforms. Different colors indicate different origins of signal values (chosen by the user, arbitrarily chosen by the tool, no other possibility, or a consequence of a choice of some other signal value). The colors remind the user of her choices and their consequences. When analyzing a lost play, this allows her to see where she might be better off with a different choice. The current time step is highlighted in red. Also highlighted is the current state of the play in all DBWs of the specification, as well as state transitions which are still possible. The user has two possibilities for setting signal values. First, she can change signal values directly in the waveforms. Second, she can click on state transitions in the DBWs. The restrictions on signal values associated with this transition will then be enforced. This allows the user to navigate through DBWs, a natural way to play the diagnostic game. A log window informs the user about what the tool did, about results, and it also provides help for interacting with the tool. When a play of the diagnostic game is finished, an explanation why the user has lost is given in this log window, too.

If the specification is realizable, the user can simulate it. Here, an additional option is available: The user can switch into a mode where the simulation trace can be corrected to match the design intent. A DBW enforcing this design intent is then created automatically. In fact, the DBW shown in Fig. 8 has been created in this way.

RATSY is implemented in Python. It uses the decision diagram package CUDD [43] for the implementation of all symbolic algorithms. Furthermore, it uses the

[4] `http://rat.fbk.eu/ratsy/`

[5] `http://www.gnu.org/licenses/lgpl.html`


File Edit View Help
New Open Save Traces
Assurance Simulation Realizability & Synthesis Game
Show Results Show Operations Show Help
Automata: Step 4
true accept !s_failed s_failed error V1 false true V2 s_failed !error false R2 s_failed !s_failed
Step1 Step2 Step3 Step4 Step5
Start Stop Next Step Clear Prev. Step Done Export Show Subviews Hide Subviews
Options for unrealizable specs: Sat Check Minimize Compute Graph
Play Game Specify Design Intent

Game Log

Starting Marduk with the file ...
Checking for realizability (may take some time) ...
The spec is unrealizable.
I.e., no system can be constructed that implements the specification. Given an arbitrary system, the environment can

Signals

Name	Type	Kind	Notes
r0	boolean	E	resource request by entity 0
r1	boolean	E	resource request by entity 1
s_failed	boolean	E	environment could not start up
g0	boolean	S	resource granted to entity 0

Requirements

	Name	Kind	Automaton	Property
	sys tran 0	G		never(g0=1 && g1=1)
✓	sys tran 1	G		always(error=1 -> (g0=0 && g1=0))
✓	sys fair 0	G		always(eventually!(r0=0 || g0=1))

Automata

Name	Notes
design_intent0	Created automatically from a simulation run.

Game

Inputs: r0, r1, s_failed, ix, jx, jx changes, state in graph

Outputs: g0, error

Fig. 8: A snapshot of RATSY's GUI.

model-checker NuSMV [10] to create BDDs from PSL formulas. If a formula is not GR(1), NuSMV is often able to bring it into the desired format using syntactical transformations. It also encodes multi-valued variables using Boolean signals.

7 Experimental Results

For our evaluation, we used mutants of two different specifications, both parameterized. The first one defines an arbiter for ARM's AMBA AHB bus [3], parameterized with the number of masters it can serve. Mutants are denoted A*nei*, where n is the number of masters, e characterizes the error we introduced artificially (`woef` for a removed fairness assumption, `wsf` for an added fairness guarantee, and `wst` for an added safety guarantee), and i is a running index to distinguish different mutations of the same kind. The second specification, denoted G*nei* with the same syntax, defines a generalized buffer [4] connecting n senders to two receivers. All specification mutants are satisfiable but unrealizable. Their size is between 90 properties over 22 signals (`A2woef1`) and 6004 properties over 218 signals (`G100wst1`). All signals are Boolean control signals. Most properties refer to only a couple of signals. Only one property in the G*nei* mutants contains up to 100 signals, but this property is simple in structure. Note that the mutants A*n*`woef`*i* and G*n*`woef`*i* do not match the fault model we assume for diagnosis, namely that assumptions are always correct. Nevertheless, our approach is able to compute diagnoses. Our method to explain unrealizability using counterstrategies does not rely on this assumption. It leaves it up to the user to conclude whether guarantees are too strong or assumptions are too weak.

7.1 Performance Results

Table 1 summarizes performance results of RATSY. The scripts to reproduce them are contained in the distribution of RATSY. All experiments were performed on an Intel Core Duo P7350 processor with 2×2.0 GHz and 3 GB RAM, running a 32-bit Linux.

The Columns 1 to 4 show results for applying model-based diagnosis (MBD). Column 1 gives the time for single-fault diagnoses computation as described in Section 4.2.4 with Delta Debugging as a minimization algorithm. Column 2 contains the time for the same computation with the performance optimization using approximations of realizability as described in Sections 4.2.5 and 5.3. The corresponding speed-up factor is listed in Column 3. Column 4 contains the number of single-fault diagnoses (potentially faulty guarantees or outputs) that have been computed.

The Columns 5 to 7 summarize results when a counterstrategy is computed for the entire specification. The overall time for computing the counterstrategy (the winning region followed by the counterstrategy's relation) is given in Column 5. Column 6 shows the number of nodes in the graph $\mathbb{G}$ that summarizes all plays that are possible against this counterstrategy. Graph computation was aborted as soon as 1 000 nodes were reached. Such entries are marked with '>1 k'. A '?' in Column 6 indicates a time-out without reaching 1 000 nodes. Column 7 indicates if our heuristic found a countertrace.

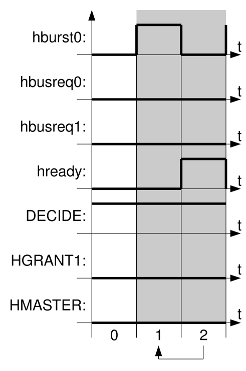
Table 1: Performance Results.

Column	1	2	3	4	5	6	7	8	9	10	11	12
	Computing Diagnoses				Entire Spec			Unrealizable Core				
	Time: $\{\Delta_1\}$	Time: optimized	Speed-Up Factor	$\lvert\{\Delta_1 \mid \lvert\Delta_1\rvert = 1\}\rvert$	Time: $W_{\text{env}}^{\text{GR1}}$ $+\rho_{\text{env}}^{\text{GR1}}$	# Vertices in $\mathbb{G}$	$\mathbb{T}$ found	Time: ddmin	Speed-Up Factor vs. linMin [11]	Size of Core	# Vertices in $\mathbb{G}$	$\mathbb{T}$ found
	[s]	[s]	[-]	[-]	[s]	[-]	[-]	[s]	[-]	[-]	[-]	[-]
`A2woef1`	2.6	1.5	1.8	7	0.8	27	yes	1.3	5.8	9	5	yes
`A4woef1`	311	232	1.3	8	48	75	yes	3.5	47	11	13	yes
`A5woef1`	2.5 k	960	2.6	9	595	139	yes	5.3	499	12	5	yes
`A2wsf1`	5	1.4	3.5	6	0.6	59	yes	1.2	9.2	8	5	yes
`A4wsf1`	213	62	3.4	7	64	171	yes	3.5	176	9	5	yes
`A5wsf1`	4.9 k	1.1 k	4.7	7	493	683	yes	8	2.2 k	9	5	yes
`A2wsf2`	7.4	1.7	4.3	6	0.7	9	yes	2.7	4.1	12	7	yes
`A4wsf2`	444	59	7.6	7	60	35	yes	40	16	20	13	yes
`A5wsf2`	8.0 k	1.7 k	4.7	12	550	9	yes	225	73	12	17	yes
`A2wst1`	3.6	1.3	2.8	8	0.7	51	yes	1.4	4.3	9	7	yes
`A4wst1`	93	47	2	9	8	139	yes	4.3	28	11	19	yes
`A5wst1`	1.6 k	336	4.8	10	96	>1 k	yes	12	92	12	43	yes
`A2wst2`	3.4	1.6	2.1	8	0.3	7	yes	1.9	3.6	10	7	yes
`A4wst2`	115	50	2.3	9	15	7	yes	5.2	15	12	19	yes
`A5wst2`	449	221	2	10	26	7	yes	6.8	24	13	63	yes
`G5woef1`	3.3	3	1.1	10	0.5	192	no	2.4	1.5	15	52	no
`G20woef1`	24	23	1	10	4.8	>1 k	no	14	4.4	15	52	no
`G100woef1`	1.2 k	1143	1.1	10	317	?	no	253	236	15	52	no
`G5wsf1`	6.2	6.1	1	14	0.2	249	no	5	0.7	19	3	yes
`G20wsf1`	787	1.0 k	0.8	44	1.4	789	no	409	0.2	49	3	yes
`G100wsf1`	>40 k	>40 k	?	?	41	?	no	>40 k	?	?	?	?
`G5wsf2`	1.1	1.1	1	2	0.2	>1 k	no	0.8	2.6	7	36	no
`G20wsf2`	8.1	9.2	0.9	2	1.6	>1 k	no	4.7	7.9	14	58	no
`G100wsf2`	404	407	1	2	54	?	no	181	309	7	36	no
`G5wst1`	1.1	0.7	1.5	1	0.2	7	yes	1.1	2.4	7	4	yes
`G20wst1`	6.3	5.3	1.2	1	1.4	22	yes	5.7	8.8	7	4	yes
`G100wst1`	305	302	1	1	52	46	yes	238	240	7	4	yes
`G5wst2`	1.6	1.3	1.2	2	0.2	37	no	1.7	1.7	9	4	yes
`G20wst2`	8.2	6.9	1.2	2	1.4	118	no	7.5	6.8	9	4	yes
`G100wst2`	402	368	1.1	2	65	262	no	263	215	9	4	yes
total	22 k	8.0 k	2.7	226	2.5 k		60%	1.7 k	157	358	549	80%

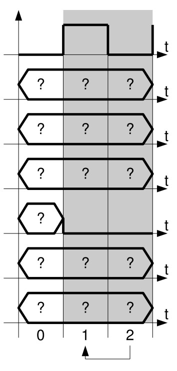
The Columns 8 to 12 contain results when counterstrategies are combined with unrealizable cores. The time for unrealizable core computation using Delta Debugging is listed in Column 8. Column 9 gives the speed-up factor of Delta Debugging compared to linMin, which is used for core computation in [11]. Column 10 shows the size of the core. Finally, the last two columns list the size of the graph $\mathbb{G}$, and whether a countertrace could be found. Entries preceded with '>' indicate time-outs. A '?' marks a missing value due to a time-out.

In our experiments, we observed the following.

– **MBD achieves a more precise error localization than single unrealizable cores.** Every unrealizable core element can be seen as a fault candidate. In our experiments, MBD yields 37% fewer single-fault candidates (Column 4 vs. 10).
– **The performance optimizations discussed in Section 4.2.5 are effective.** We achieve a speed-up factor of 2.7 in the computation of single-fault diagnoses (Column 3). However, single-fault diagnosis computation is still slower than unrealizable core computation by a factor of 4.7 (Column 1 vs. 8).
– **Our heuristic algorithm for countertrace computation performs well.** A countertrace is found in 60% of the cases, and in 80% when using an unrealizable core (Columns 7 and 12). In our experience, countertraces are much easier to understand than counterstrategies. Computation time is insignificant (fractions of a second) if a counterstrategy is

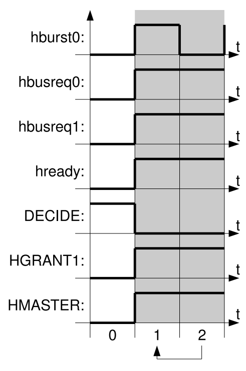


(a) Simulation

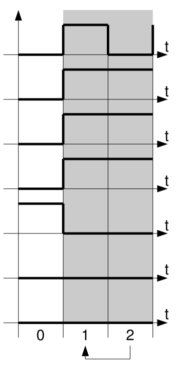


(b) Design Intent

Fig. 9: Debugging a mismatch with the design intent.



(a) HMASTER remains 1



(b) HMASTER=0

Fig. 10: Two possible plays against the countertrace.

available. Hence, the heuristic provides much simpler explanations at very little additional costs.

– **Computing at least one unrealizable core is very beneficial.** The size of the specification is reduced by 90% on average, which makes it much easier to debug. We found that the size of the graph $\mathbb{G}$ is a good indicator how difficult it is for the user to get meaningful insights during the diagnostic game. This size is reduced dramatically due to unrealizable core computation (Column 6 vs. 11)[6]. It also increases the chances for finding a countertrace. Moreover, it often even speeds up the entire computation, because once a core is available, the time for counterstrategy computation is negligible (Column 5 vs. 8).
– **Delta Debugging is often much faster than linMin.** In total, we achieve a speed-up factor of 157, with higher speed-ups for larger specifications (Column 9). Only for the mutants G*n*wsf1, Delta Debugging is slower.

7.2 Example

In this section, we show how to apply our debugging approach to resolve a mismatch between a fictitious design intent and the unmodified specification of the AMBA bus arbiter [3] for two masters. We use lower case letters for input signals and upper case letters for outputs. The output HMASTER is set to 0 whenever the bus is owned by Master 0, and set to 1 when owned by Master 1. If the input hready is set, then HGRANT1=1 signals that HMASTER=1 in the next step. Analogously for HGRANT0. The signals HGRANT*i* can only change when the output DECIDE is set. Finally, the bus can be requested by Master 0 and Master 1 with the inputs hbusreq0 and hbusreq1. The meaning of all other signals is irrelevant for this example.

[6] There is no guarantee that the size of the graph $\mathbb{G}$ decreases, and in one case it actually increases. In fact, removing only guarantees increases the graph size because it gives the system more freedom. Removing output signals counteracts this effect.

A possible simulation trace is depicted in Fig. 9a. The gray background marks the part that repeats infinitely often. Suppose now that the user is not satisfied with this behavior. Suppose that the design intent was that DECIDE=0 as long as input hburst0 keeps changing. Our tool allows the user to modify the obtained simulation trace so that it matches this expectation. The result is shown in Fig. 9b. Next, following our debugging approach, the tool adds a guarantee to the specification which enforces this desired behavior. The resulting specification is now unrealizable. This means that the design intent is in contradiction with the rest of the specification. This contradiction is debugged by debugging the unrealizability of the extended specification.

Our debugging procedure for unrealizability starts with MBD computing unrealizable cores and diagnoses. Besides the guarantee enforcing the design intent, an unrealizable core contains the guarantees

```
HMASTER=HGRANT1=A0=A1=LK=0 and DECIDE=1,                  (8)
always(((LK=0 or hburst0=1 or hburst1=1)
        and A0=A1=0) implies next(A0=A1=0)),              (9)
always(hready=1 implies
        (next(HMASTER=1) iff HGRANT1=1)),                 (10)
always(DECIDE=0 implies
        (next(HGRANT1=0) iff HGRANT1=0)),                 (11)
always(eventually(HMASTER=0 or hbusreq0=0)), and          (12)
always(eventually(HMASTER=1 or hbusreq1=0)).              (13)
```

Although this unrealizable core is rather short, it is not easy to manually derive an explanation. Our tool helps. It computes a counterstrategy and it is able to find a countertrace. This countertrace is depicted in Fig. 10, together with two different responses in the diagnostic game. It explains the unrealizability of the core as follows: Since hburst0 keeps changing, DECIDE must be 0 after the first tick according to the design intent. Output HGRANT1 cannot change as long as DECIDE=0 (Eq. 11). Hence, HGRANT1 can only change its value in Step 1, then it remains constant. Since `hready` is set to 1 in the countertrace, `HMASTER` cannot change either (Eq. 10). If it remains 1 (Fig. 10a), then the fairness guarantee stated in Eq. 12 cannot be fulfilled because `hbusreq0=1` forever. If it remains 0 (Fig. 10b), then Eq. 13 cannot be fulfilled because `hbusreq1=1` forever. Hence, no matter how the system behaves, it cannot fulfill all guarantees. The countertrace fulfills all assumptions, so the specification cannot be fulfilled. The guarantee in Eq. 9 is only in the core because without it the system could enforce a violation of an assumption.

There are many ways to resolve the illustrated problem. By removing Eq. 11 the user could allow `HGRANT1` to change even if `DECIDE=0`. This would resolve the presented core, but it would not make the entire specification realizable. The reason is that an analogous conflict exists with `HGRANT0`. Similarly, Eq. 10 and Eq. 12 cannot be modified in such a way that the specification becomes realizable. MBD helps the user to find a possible location for the fix. It identifies Eq. 9 and Eq. 13 as the only guarantees being single-fault diagnoses. If the fix should affect only one guarantee, then it must be one of these.

8 Summary and Conclusion

In this work, we addressed the problem of debugging an incorrect formal specification of a reactive system in absence of a corresponding implementation. We showed how debugging mismatches between the informal design intent and the formal specification can be reduced to debugging unrealizability. Our method for debugging unrealizability is based on two pillars. First, we use model-based diagnosis for error localization. Second, we explain problems in the specification using counterstrategies. In order to obtain helpful explanations, we discussed techniques to keep counterstrategies simple. Our generic debugging method can be applied to many logics. Experimental results for GR(1) specifications show that the proposed debugging method is both feasible and useful.

Model-based diagnosis for unrealizability is computationally expensive but it yields more precise information about the error location than single unrealizable cores. In order to explain unrealizability in a simple way, it is absolutely vital to compute a counterstrategy not for the original specification but for an unrealizable core. This greatly reduces the complexity of the specification and allows the user to focus on one problem at a time. A significant part of the simplification can be attributed to the fact that we do not only remove properties but also signals from the specification when computing unrealizable cores. Model-based diagnosis produces such unrealizable cores as a side-product, which makes the techniques fit together nicely. Countertraces are much easier to understand than counterstrategies, because they are independent of the moves of the opponent. We presented a heuristic for the computation of such countertraces. Although not complete, it is able to find countertraces in many cases. It avoids an exponential blow-up of the state space and it is very fast in practice.

Acknowledgments
We thank Amir Pnueli for suggesting the use of counterstrategies. We also thank Viktor Schuppan and the anonymous reviewers for their valuable comments.

References

1. G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-Tiga: Time for playing games! In *Proc. Computer Aided Verification (CAV'07)*, pages 121–125, 2007.
2. R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Koenighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY — a new requirements analysis tool with synthesis. In *Proc. Computer Aided Verification*, pages 425–429, 2010. LNCS 6174.
3. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *In Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.
4. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware form PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, 2007. Electronic Notes in Theoretical Computer Science.
5. Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamamenta Informaticae*, 62(2):139–169, 2004.
6. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
7. K. Chatterjee, T. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *International Conference on Concurrency Theory (CONCUR)*, pages 147–161, 2008.
8. H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. A practical approach to coverage in model checking. In G. Berry, H. Comon, and A. Finkel, editors, *Thirteenth Conference on Computer Aided Verification (CAV'01)*, pages 66–78. Springer, 2001. LNCS 2102.
9. H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. In *Tools and algorithms for the construction and analysis of systems (TACAS)*, pages 528–542. Springer, 2001. LNCS 2031.
10. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV'02)*, 2002.
11. A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'08)*, pages 52–67, 2008.

12. K. Claessen. A coverage analysis for safety property lists. In *Proc. Formal Methods in Computer Aided Design*, pages 139–145, 2007.
13. L. Console, G. Friedrich, and D. Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1494–1499. Morgan-Kaufmann, 1993.
14. S. Das, A. Banerjee, P. Basu, P. Dasgupta, P. P. Chakrabarti, C. R. Mohan, and L. Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In *VLSI Design*, pages 201–206, 2005.
15. S. Dellacherie. Automatic bus-protocol verification using assertions. In *Global Signal Processing Expo Conference (GSPx)*, 2004.
16. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152:213–234, 2004.
17. E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *Proc. Computer Aided Verification*, pages 263–277, 2009.
18. D. Fisman, O. Kupferman, S. Seinvald, and M. Y. Vardi. A framework for inverent vacuity. In *Proc. Haifa Verification Conference (HVC)*, 2008.
19. G. Friedrich and K. M. Shchekotykhin. A general diagnosis method for ontologies. In *International Semantic Web Conference*, pages 232–246, 2005.
20. G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *Artif. Intell.*, 111(1-2):3–39, 1999.
21. Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*. Springer, 2002. LNCS 2500.
22. D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE'07)*, pages 1176–1181, 2007.
23. Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proc. Design Automation Conference*, pages 300–305, 1999.
24. B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.
25. S. Katz, O. Grumberg, and D. Geist. "Have I written enough properties?" — A method of comparison between specification and implementation. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 280–297. Springer, 1999. LNCS 1703.
26. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.
27. R. Koenighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *Proc. Formal Methods in Computer Aided Design (FMCAD'09)*, pages 152–159, 2009.
28. R. Koenighofer, G. Hofferek, and R. Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Proc. Haifa Verification Conference (HVC)*, pages 29–45. Springer, 2010. LNCS 6504.
29. R. Könighofer. Debugging formal specifications with simplified counterstrategies. Master's thesis, IAIK, Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria, 2009.
30. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
31. O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science*, pages 531–542, 2005.
32. M. Leucker. Model checking games for the alternation-free μ-calculus and alternating automata. In *Proc. Int. Conf. on Logic Programming and Automated Reasoning (LPAR'99)*, pages 77–91. Springer, 1999.
33. M. Leucker and T. Noll. Truth/SLC - a parallel verification platform for concurrent systems. In *Computer Aided Verification*, pages 255–259. Springer, 2001.
34. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
35. C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Model-based debugging of java programs. In *AADEBUG*, 2000.
36. A. Morgenstern and K. Schneider. Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. *CoRR*, abs/1006.1408, 2010.
37. R. Mori and N. Yonezaki. Several Realizability Concepts in Reactive Objects. *Information Modeling and Knowledge Bases*, 1993.
38. I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *Design Automation Conference*, pages 821–826, 2006.
39. N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.
40. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.
41. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.
42. R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.
43. F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.
44. P. Stevens and C. Stirling. Practical model-checking using games. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998. LNCS 1384.
45. C. Stirling. Local model checking games. In *Proc. Concurrency Theory*, pages 1–11. Springer-Verlag, 1995.
46. M. Stumptner and F. Wotawa. Debugging functional programs. In *Proceedings on the 16th International Joint Conference on Artificial Intelligence*, 1999.
47. L. Tan. PlayGame: A platform for diagnostic games. In *Computer Aided Verification*, pages 492–495. Springer, 2004. LNCS 3114.
48. S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *World Congress on Formal Methods*, pages 233–252, 1999.
49. N. Yoshiura. Finding the causes of unrealizability of reactive system formal specifications. In *Proc. Software Engineering and Formal Methods (SEFM'04)*, pages 34–43, 2004.
50. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.