Debugging Formal Specifications Using Simple Counterstrategies

Robert Könighofer, Georg Hofferek, and Roderick Bloem
Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology

***Abstract*—Deriving a formal specification from an informal design intent is an error-prone process. The resulting specification may be incomplete, unrealizable, or in conflict with the design intent. We propose a debugging method for incorrect specifications that does not need an implementation.**

We show that we can explain conflicts with the design intent by explaining unrealizability. Our approach for explaining unrealizability is based on counterstrategies. Since counterstrategies may be large, we propose several ways to simplify them. First, we simplify the specification itself by removing both requirements and variables that do not contribute to the problem. Second, we heuristically search for a countertrace, i.e., a single input trace that suffices to demonstrate unrealizability. Finally, we present the countertrace or the counterstrategy to the user in extensive form as a graph and implicitly as an interactive game. We present experimental results for specifications given as GR(1) formulas.

I. Introduction

Ideally, a formal specification is written before the implementation. The specification can then be implemented either manually or automatically [20], [14], [10], [19]. In this scenario, the specification must have the highest quality possible. The same holds if the specification is written as independent verification IP. This scenario occurs quite frequently, for instance for protocols [7]. Creating a formal specification is an error-prone process and it is hard to achieve a high quality [12], [18], [6], [9], [5]. First, a specification may be incomplete. Second, it may be unrealizable, i.e., there may not be any implementation fulfilling it. Third, the specification might define something different from the engineer's original (informal) design intent. The aim of this paper is to debug specifications in the absence of an implementation.

Incomplete specifications have been addressed before. Katz et al. [12] analyze the completeness of a specification with respect to a given implementation. Claessen [6] gives a coverage analysis of a list of safety properties, introducing the notion of "forgotten cases" in which the system is underspecified. Fisman et al. [8] propose to check if a specification can be mutated into a simpler equivalent one in order to detect "inherent vacuity". The approaches of [6] and [8] are independent of an actual implementation.

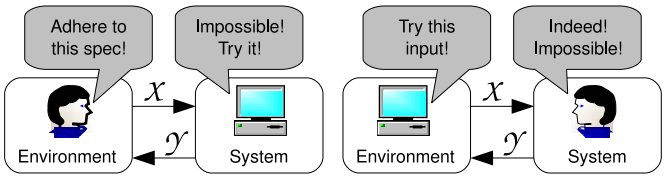
The first contribution of this paper is a method to tackle inconsistencies between the specification and the design intent. Suppose there is an implementation that was either automatically synthesized from the formal specification or implemented manually. Suppose further that this system exhibits some behavior that differs from the designer's original intent, although the system conforms to the specification. In that case, either the specification is incomplete, or there is an inconsistency between the (informal) design intent and the specification. We will show that explaining such inconsistencies can be reduced to explaining unrealizability of specifications.

Unrealizability is a problem of its own. Our experience with the synthesis tools Lily [10] and Anzu [11] shows that mistakes during specification development often lead to unrealizability. Explaining unrealizability is difficult. There is no way to execute or simulate an unrealizable specification to track down the error, like one would do with erroneous implementations. Tools like RAT [18] explain why single traces do not fulfill the specification, but this does not suffice to explain unrealizability. Note that realizability is not the same as satisfiability. A specification is satisfiable if there is one input/output trace that satisfies the specification. In contrast, realizability requires that for each input trace we can construct a correct output trace step by step. Our case study shows that many unrealizable specifications are still satisfiable. Thus, known techniques from SAT solvers cannot be used to find the cause of unrealizability.

We present an interactive approach for explaining unrealizability, based on the following idea: When a user learns that her specification is unrealizable, she will be puzzled, since she must have imagined an implementation. In order to show that the imagined implementation is flawed, the debugging tool takes on the role of the environment, while the user takes on the role of the system. (See Fig. 1.) The tool provides inputs and the user tries to provide outputs conforming to the specification. The tool uses a counterstrategy to find inputs in such a way that there is no response of the system that fulfills the specification. Hence, the user will fail. However, while trying, she gains insights into why there is no way for her to comply with the specification, i.e., why the specification is unrealizable. She can subsequently use this knowledge to correct the specification.

Our experience shows that just presenting a counterstrategy does not suffice to explain unrealizability of larger specifications. The counterstrategy can be so complex that the user is unable to learn where the specification is too restrictive to be realizable. Thus, we present several simplifications. We adopt the idea of Cimatti et al. [5] to compute an unrealizable core. As the second contribution of this paper, we improve

This work was supported in part by the European Commission through project COCONUT (FP7-2007-IST-1-217069).



(a) Simulating the System (b) Debugging Unrealizability

Fig. 1. Swapping the roles to gain insight into the cause of unrealizability.

over [5] by removing unnecessary signals and using delta debugging [26], an efficient minimization algorithm. Removing signals is crucial: just computing an unrealizable core leads to a specification that is harder to explain, not easier. Our third contribution is to use *countertraces* where possible. A countertrace is a fixed input trace for which there is no output trace fulfilling the specification. Focusing on this single trace makes it much easier for the user to localize the problem. A countertrace does not always exist and even if one exists, its computation is expensive. Hence, we present a heuristic.

After introducing our (generic) debugging approach, as a final contribution we show how it can be applied to specifications given in *Generalized Reactivity(1)* (GR(1), for short) [19]. GR(1) is a subset of LTL that has enough expressive power to be used for real world problems [2], [3] while still offering efficient symbolic algorithms [19]. We have evaluated our concepts for this class of specifications by integrating them into the synthesis tool Anzu [11].

Counterstrategies as debugging aids were previously used in the context of restricted specifications for timed systems [25], [1], in the context of Live Sequence Charts [4], and as witnesses or counterexamples to branching-time logic formulas [23], [22]. More efficient algorithms for the latter are presented in [15], [16], [24]. These papers mention simplification only peripherally, by discussing user interface and usability issues. We focus on simplifying the counterstrategy itself in order to convey meaningful information to the user, which is not done in the papers mentioned. (But see [5], as discussed above.) We are not aware of any previous work on finding countertraces, which we consider the most practical tool towards understanding why a specification is unrealizable. Also, to the best of our knowledge, counterstrategies have not been used before to explain conflicts between formal specifications and informal design intents.

The rest of the paper is organized as follows. Section II will revisit definitions and establish some notation. Section III introduces our debugging approach and Section IV concretizes it for GR(1) specifications. Section V presents our evaluation results for GR(1) and Section VI concludes the article.

II. Preliminaries

A. Automata

A *(deterministic and complete) automaton* is a tuple $\mathcal{A} = (Q, \Sigma, T, q_0, \mathrm{Acc})$, where Q is a finite set of states, Σ is a finite alphabet, $T : Q \times \Sigma \to Q$ is a deterministic and complete transition function, $q_0 \in Q$ is the initial state, and $\mathrm{Acc} : Q^\omega \to \{\mathsf{false}, \mathsf{true}\}$ is the acceptance condition. A *run* of the automaton 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 $\mathrm{Acc}(\overline{r}) = \mathsf{true}$.

A *deterministic and complete Büchi word automaton (DBW)* is an automaton in which Acc is given as a set of states $F \subseteq Q$ such that $\mathrm{Acc}(\overline{r}) = \mathsf{true}$ iff $\inf(\overline{r}) \cap F \neq \emptyset$, where $\inf(\overline{r})$ is the set of states that occur infinitely often in $\overline{r}$.

In the following, we assume that $Q = 2^V$ for a set V of state bits, and that $\Sigma = 2^X \times 2^Y$ for a set X of Boolean input signals and a set Y of Boolean output signals. We write $\mathcal{X} = 2^X$ and $\mathcal{Y} = 2^Y$. For $\overline{x} = x_0x_1 \ldots \in \mathcal{X}^\omega$ and $\overline{y} = y_0y_1 \ldots \in \mathcal{Y}^\omega$ we define $\overline{x}||\overline{y} = (x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$ as their combination.

B. Generalized Reactivity

Generalized Reactivity (1) specifications form a subset of Linear Temporal Logic (LTL) [19]. They specify the interaction between an environment (controlling the input variables X) and a system (controlling the output variables Y) and consist of two parts: *assumptions* and *guarantees*. The specification states that the system must fulfill all guarantees whenever the environment fulfills all assumptions.

A GR(1) specification consists of $m+n$ DBWs representing m environment assumptions and n system guarantees [19]. With $\mathcal{A}^e_i = (Q^e_i, \Sigma, T^e_i, q^e_{0,i}, F^e_i)$ we denote the DBWs representing environment assumptions. The DBWs representing system guarantees are denoted $\mathcal{A}^s_j = (Q^s_j, \Sigma, T^s_j, q^s_{0,j}, F^s_j)$. All DBWs share the alphabet Σ. Let $\mathcal{A}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathrm{Acc})$ be the product of all DBWs $\mathcal{A}^e_i$ and $\mathcal{A}^s_j$, where the state space is $Q = Q^e_1 \times \cdots \times Q^e_m \times Q^s_1 \times \cdots \times Q^s_n$, the transition function is $T\left((q^e_1, \ldots, q^s_n), \sigma\right) = \left(T^e_1\left(q^e_1, \sigma\right), \ldots, T^s_n\left(q^s_n, \sigma\right)\right)$, and the initial state is $q_0 = (q^e_{0,1}, \ldots, q^s_{0,n})$. Let $J^e_i = \{(q^e_1, \ldots, q^s_n) \mid q^e_i \in F^e_i\}$ be the set of all states of the product automaton $\mathcal{A}^{\mathrm{GR1}}$ that are accepting in $\mathcal{A}^e_i$. Similarly, let J^s_j be the set of all states of $\mathcal{A}^{\mathrm{GR1}}$ that are accepting in $\mathcal{A}^s_j$. The acceptance condition Acc is

$$\mathrm{Acc}(\overline{r}) \Leftrightarrow (\forall i : \inf(\overline{r}) \cap J^e_i \neq \emptyset) \rightarrow (\forall j : \inf(\overline{r}) \cap J^s_j \neq \emptyset).$$

Thus, a run of $\mathcal{A}^{\mathrm{GR1}}$ is accepting iff all sets J^s_j of accepting states of the system are visited infinitely often, or some set J^e_i of accepting states of the environment is visited only finitely often.

C. Games and Strategies

A *game* is a tuple $\mathcal{G} = (Q, \Sigma, T, q_0, \mathrm{Win})$, where Q, T, and q_0 are defined as for DBWs and $\mathrm{Win} : Q^\omega \to \{\mathsf{false}, \mathsf{true}\}$. The game is played by two players. A *play* of $\mathcal{G}$ is an infinite sequence of states $\overline{\pi} = q_0q_1q_2 \ldots \in Q^\omega$, where $q_{i+1} = T(q_i, \sigma_i)$ for $i \geq 0$. The letters $\sigma_i = (x_i, y_i)$ are successively chosen by the players: In each step Player 1 first chooses x_i, after which Player 2 chooses y_i. A play $\overline{\pi}$ is won by Player 1 iff $\mathrm{Win}(\overline{\pi}) = \mathsf{true}$. Otherwise it is lost for Player 1 and won for Player 2. Note that Player 1 cannot react

to Player 2 and thus acts like a Moore machine. In contrast, Player 2 acts like a Mealy machine.

For GR(1) games, finite memory strategies suffice [19]. A *(finite memory) strategy* for Player 1 in the game $\mathcal{G}$ is a tuple (Γ, γ_0, ρ), where $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$, Γ is some (finite) set representing the memory, and $\gamma_0 \in \Gamma$ is the initial memory content. The relation ρ maps a state of the game and the memory content to a set of possible choices for the inputs and an updated memory content. We require that ρ is complete, i.e., $\forall q, \gamma\, \exists x, \gamma' : (q, \gamma, x, \gamma') \in \rho$. A play $\overline{\pi} = q_0 q_1 \ldots$ *conforms to a strategy* (Γ, γ_0, ρ), iff there is a sequence $(x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$ and a sequence $\gamma_0 \gamma_1 \ldots \in \Gamma^\omega$ such that for all $i \geq 0$, $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ and $q_{i+1} = T(q_i, (x_i, y_i))$. A strategy is *winning* from a state $q \in Q$ iff all plays starting from q and conforming to the strategy are won by Player 1. The *winning region* $W \subseteq Q$ of Player 1 is the set of states for which a winning strategy for Player 1 exists. A *counterstrategy* is a winning strategy for Player 1 from q_0.

A *co-GR(1) game* $\mathcal{G}_{\text{env}}^{\text{GR1}}$ (where Player 1 is the environment and Player 2 is the system) can be constructed from the automaton $\mathcal{A}^{\text{GR1}}$. The winning condition of $\mathcal{G}_{\text{env}}^{\text{GR1}}$ is the complement of the GR(1) acceptance condition:

$$\text{Win}(\overline{r}) \Leftrightarrow (\forall i : \inf(\overline{r}) \cap J_i^e \neq \emptyset) \wedge (\exists j : \inf(\overline{r}) \cap J_j^s = \emptyset) \quad (1)$$

D. μ-Calculus

We will use the propositional μ-calculus [13] extended with a mixed-preimage operator MX, defined on sets of states of a game. Let Var be a set of variables each representing a specific subset of Q. The syntax of μ-calculus formulas is defined recursively: Every subset $S \subseteq Q$ and every variable $Y \in \text{Var}$ is a μ-calculus formula. If P, Q are μ-calculus formulas, so are $\neg P$, $P \cup Q$, and $P \cap Q$, with the expected semantics. Furthermore, for $Y \in \text{Var}$, $\mu Y\,.\,P(Y)$, $\nu Y\,.\,P(Y)$, and $\mathsf{MX}(Y)$ are μ-calculus formulas defined as

$$\mu Y.P(Y) = \bigcup_i Y_i, \quad \text{where } Y_0 = \emptyset \text{ and } Y_{i+1} = P(Y_i), \quad (2)$$

$$\nu Y.P(Y) = \bigcap_i Y_i, \quad \text{where } Y_0 = Q \text{ and } Y_{i+1} = P(Y_i),$$

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

The expression $\mathsf{MX}(P)$ denotes the set of states from which the environment can force a play into a state of P in one step. The order of existential and universal quantification corresponds to the fact that Player 1 moves first. We also define $\mathsf{MX}_x(P) = \{q \in Q \mid \forall y \in \mathcal{Y} : T(q, (x, y)) \in P\}$, i.e., the set of all states from which the games moves to P if Player 1 chooses input x.

E. Delta Debugging

Delta debugging [26] is an algorithm for minimization problems. Let S be a set that fails some test, denoted by $test(S) = \text{✘}$. We assume that some subset of S is "responsible" for the failing test and that $test$ is monotonic: $(test(S) = \text{✘}) \rightarrow (\forall S'' \supseteq S : test(S'') = \text{✘})$. The algorithm finds a minimal subset $S' \subseteq S$ so that $test(S') = \text{✘}$. The algorithm is defined recursively: $ddmin(c) = d(c, 2)$, and

$$d(c, n) = \begin{cases} d(c_i, 2) & \text{if } \exists i : test(c_i) = \text{✘} \\ d(\overline{c_i}, \max(n\text{-}1, 2)) & \text{else if } \exists i : test(\overline{c_i}) = \text{✘} \\ d(c, \min(|c|, 2n)) & \text{else if } n < |c| \\ c & \text{otherwise,} \end{cases}$$

where $(c_1, \ldots, c_n)$ is a partition of c into n approximately equally-sized parts. The set $\overline{c_i}$ is defined as $c \setminus c_i$.

The intuition behind this algorithm is a *divide and conquer* approach. At first, the set c is split in two parts and both are subjected to the test. If one of them fails the test, i.e., still contains a problem, the algorithm proceeds recursively on this part only. If both parts pass the test, elements from both parts contribute to the problem. Thus, the granularity of the partition is doubled. Since not only the subsets themselves but also their complements are tested, larger sets are tried as well. Eventually the algorithm returns a minimal subset that fails the test.

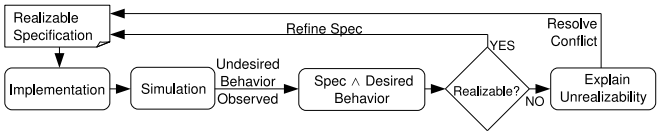


Fig. 2. The flow of our method to handle mismatches with the design intent.

III. Debugging Approach

This section introduces our generic debugging approach. We assume that we are given a temporal specification $\varphi = A \rightarrow G$ over X and Y, where A is a (possibly empty) set of environment assumptions and G is a set of system guarantees. We assume that we are given functions *realizable* and *sat* that decide realizability and satisfiability. We also assume that the specification can be turned into a two-player finite game and that we have a procedure that returns a finite state counterstrategy for an unrealizable specification. We further require that guarantees $g \in G$ can be removed from φ, and that output signals $y \in Y$ can be quantified existentially from the guarantees. These assumptions are relatively weak and hold for such logics as LTL. After presenting the general approach, we will instantiate it for GR(1).

A. Debugging Undesired Behavior

Inconsistencies between the formal specification and the designer's informal design intent may surface when a system has been built that satisfies the specification but not the design intent. The designer then has to change the specification to adhere to the design intent.

Fig. 2 illustrates our approach to handle mismatches with the design intent. We distinguish two cases:

1) The specification is incomplete, i.e., there is an implementation of the specification meeting the design intent.
2) Any system exhibiting the desired behavior violates the specification. This means that the specification is so

restrictive that there exists no implementation of the specification that shows the desired behavior.

The two cases can be distinguished by augmenting the specification with a guarantee that enforces the required behavior. If the modified specification is realizable, the original specification was incomplete and needs to be refined. Otherwise, we need to explain to the user why enforcing the desired behavior makes the specification unrealizable.

First, however, the desired behavior must be specified. We would like to provide the user with a method to obtain a relatively general requirement from the incorrect trace. We propose to allow the user to change any number of signal values in the incorrect trace to the desired value 0, 1, or $-$, where $-$ stands for "don't care". The tool then checks whether the input part of the trace conforms to the environment assumptions. If not, a corresponding warning is given to the user as the system is not required to fulfill its guarantees for such inputs. Next, the tool converts the given trace into a guarantee g_{new} in the following way. Each time step i of the desired trace $\bar{t}$ represents a function $t_i \colon (X \cup Y) \to \{0, 1, -\}$. An input trace $\overline{x} = x_0 x_1 \ldots \in \mathcal{X}^\omega$ conforms to the desired trace $\bar{t}$ (written $\overline{x} \overset{\mathcal{X}}{\models} \bar{t}$) iff

$$\forall i : \forall s \in X : (t_i(s) = 0 \rightarrow s \notin x_i) \wedge (t_i(s) = 1 \rightarrow s \in x_i).$$

Conformance with an output trace (written $\overline{y} \overset{\mathcal{Y}}{\models} \bar{t}$) is defined analogously. The guarantee g_{new} must accept exactly the words $(\overline{x}||\overline{y}) \in \Sigma^\omega$ such that $\overline{x} \overset{\mathcal{X}}{\models} \bar{t} \rightarrow \overline{y} \overset{\mathcal{Y}}{\models} \bar{t}$. This enforces the desired outputs whenever the given input scenario applies. The construction of g_{new} from the desired trace $\bar{t}$ depends on the actual specification language.

The guarantee g_{new} is then added to the original specification $\varphi = A \rightarrow G$ to obtain $\varphi' = A \rightarrow G \cup \{g_{\text{new}}\}$. Next, we check φ' for realizability. If φ' is realizable, it is a valid refinement of φ and the undesired behavior has been successfully eliminated. Otherwise we proceed by explaining the reasons for unrealizability as outlined in the next section.

B. Debugging Unrealizability

The flow of our method to explain unrealizability is depicted in Fig. 3. First, we check for satisfiability. For unsatisfiable specifications, trace-based debugging methods can be used [18]. If the specification is satisfiable, but not realizable, we apply a minimization step to find an unrealizable core. Only the part that is inconsistent or in conflict with the design intent (cf. Section III-A) remains. Based on this simplified specification, we then compute a counterstrategy. The counterstrategy is a finite state strategy such that the specification cannot be fulfilled if the environment adheres to it. We then attempt to obtain a countertrace from the counterstrategy. Countertraces and counterstrategies are finally presented to the user as a summarizing graph, and in form of an interactive game. The next sections explain the steps of this procedure in more detail.

C. Minimizing the Specification

Debugging an unrealizable specification $\varphi = A \rightarrow G$ is especially hard if it is large. However, the cause of unrealizability often involves only small parts of φ. Removing the rest

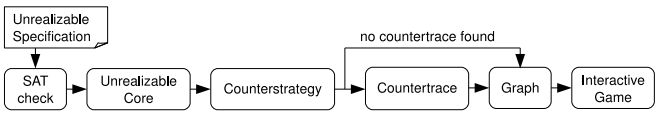


Fig. 3. The flow of our method to explain unrealizability.

leads to a simpler specification $\hat{\varphi}$ that is still unrealizable and easier to debug. Our experiments will show that minimization speeds up the computation of counterstrategies and leads to easier-to-understand games.

Removing environment assumptions would confuse the user during the interactive game as it adds (environment) behavior that the user originally excluded with these assumptions. Thus, we only remove system guarantees, looking for a minimal unrealizable core. Removing guarantees leads to shorter specifications. Surprisingly, the corresponding game is often more difficult to understand than the original. The reason is that removing guarantees adds (system) behavior. This results in more possible plays and a larger game graph. We counteract this effect by removing unnecessary outputs as well. We can do this by existentially quantifying them from all guarantees. This operation will be denoted $\exists Y' : G$ for some set $Y' \subseteq Y$ of outputs to remove.

Lemma 1: If $\varphi = A \rightarrow G$ is a realizable specification, so is $\varphi' = A \rightarrow (\exists Y' : G')$, for all $G' \subseteq G$, $Y' \subseteq Y$.

Lemma 2: Let (Γ, γ_0, ρ) be a counterstrategy for specification $\varphi' = A \rightarrow (\exists Y' : G')$, with $G' \subseteq G$, $Y' \subseteq Y$. Then (Γ, γ_0, ρ) is also a counterstrategy for $\varphi = A \rightarrow G$.

Lemma 1 states that removing guarantees or output signals preserves realizability. Furthermore, Lemma 2 states that minimizing the number of guarantees and output signals is helpful for finding a simple explanation for unrealizability, because the counterstrategy for the minimized system also applies to the original specification.

Cimatti et al. [5] propose to find an unrealizable core by removing one guarantee after the other. Thus, they require exactly $|G|$ checks for realizability to find the core. We attempt to reduce the average number of checks by using delta debugging [26]. This algorithm expects a set to be minimized as argument and uses a function $test$. We define $test(G' \cup Y') = realizable(A \rightarrow (\exists Y \setminus Y' : G'))$ and compute a minimized specification $\hat{\varphi} = A \rightarrow (\exists Y \setminus \hat{Y} : \hat{G})$, where $\hat{Y} = D \cap Y$, $\hat{G} = D \cap G$, and $D = ddmin(G \cup Y)$. We apply Lemma 1 to further reduce the number of realizability checks (cf. [26]): We store all sets $R = G' \cup Y'$ such that $A \rightarrow (\exists Y \setminus Y' : G')$ is realizable. We do not have to recompute realizability if we encounter a subset R' of a stored set R.

Theorem 1: $\forall G' \subseteq \hat{G}, Y' \subseteq \hat{Y} : ((G', Y') \neq (\hat{G}, \hat{Y})) \rightarrow realizable(A \rightarrow (\exists Y \setminus Y' : G'))$.

Proof: See the proof of Proposition 11 in [26]. ■

D. Countertraces

In general, the inputs given by the counterstrategy depend on the previous outputs of the system. Thus, a counterstrategy can be viewed as a graph or an interactive game. To make

things easier for the user we would prefer to construct a single *countertrace*, i.e., an infinite trace $\overline{x} \in \mathcal{X}^\omega$ for which there is no $\overline{y} \in \mathcal{Y}^\omega$ such that $\overline{x}||\overline{y}$ fulfills the specification. If the inputs that the user faces are always the same, regardless of her choice of outputs, the time and effort for understanding why she always loses the game is much lower. Unfortunately, such a countertrace does not always exist. A typical example is the LTL specification $\mathsf{G}(y \leftrightarrow \mathsf{X}\, x)$. This specification is not realizable, but for any given input trace $\overline{x}$ there is an output trace $\overline{y}$ such that $\overline{x}||\overline{y}$ fulfills the specification [21], [17].

We can compute whether a countertrace exists by existentially quantifying the output variables from the game automaton, complementing the automaton, and checking for emptiness. However, the game automaton may be non-deterministic after quantification and complementing it would cause an exponential blow-up. We consider that intractable and therefore present a heuristic. It does not always find a countertrace, even if one exists. However, our experiments show that it suffices for many cases of interest.

Our heuristic starts with a counterstrategy (Γ, γ_0, ρ) for a game $\mathcal{G} = (Q, \Sigma, T, q_0, \mathrm{Win})$, where $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$. We simultaneously compute a countertrace $\overline{x} = x_0 x_1 \ldots \in \mathcal{X}^\omega$ and a sequence of sets $S_i \subseteq (Q \times \Gamma)$, where S_i contains all pairs of state and memory content possible after $x_0 \ldots x_{i-1}$ has been used as input. We start with $S_0 = \{(q_0, \gamma_0)\}$ and define

$$S_{i+1} = \big\{(q', \gamma') \;\big|\; \exists (q, \gamma) \in S_i, y \in \mathcal{Y} : \\ q' = T(q, (x_i, y)) \wedge (q, \gamma, x_i, \gamma') \in \rho\big\},$$

where x_i is chosen arbitrarily from the set $T_i = \{x \in \mathcal{X} \mid \forall (q, \gamma) \in S_i \, \exists \gamma' : (q, \gamma, x, \gamma') \in \rho\}$. Intuitively, the set T_i contains all inputs that conform to the counterstrategy, no matter in which precise state $(q_i, \gamma_i) \in S_i$ the play is. Thus, x_i is independent of the previous moves of the system. If $T_i = \emptyset$ for any i, our heuristic fails.

Let k be the smallest number such that $S_k \subseteq S_j$ for some $j < k$. We can easily show by induction that if we pick $x_{k+i} = x_{j+i}$, then $T_{k+i} \supseteq T_{j+i}$ and $\emptyset \subset S_{k+i} \subseteq S_{j+i}$ for all $i \geq 0$ (the former inclusion by completeness of T, the latter by monotonicity of the definition of S_{i+1}).

Thus, we can stop the computation when we find a set inclusion, obtaining a lasso-shaped countertrace $\overline{x}$ composed of the finite stem $x_0 \ldots x_{j-1}$ and infinite many repetitions of $x_j \ldots x_{k-1}$. Although the upper bound for k is exponential in the number of states, our experiments show that it is typically rather small (< 10 in most cases).

A countertrace represents a strategy $(\Gamma_{\overline{x}}, 0, \rho_{\overline{x}})$ with memory $\Gamma_{\overline{x}} = \{0, \ldots, k-1\}$. We define

$$\rho_{\overline{x}} = \big\{(q, i, x_a, i \oplus 1) \;\big|\; q \in Q \wedge i \in \Gamma_{\overline{x}}\big\}, \text{ where}$$
$$i \oplus 1 = \begin{cases} i+1 & \text{if } a < k-1, \\ j & \text{if } a = k-1. \end{cases}$$

Theorem 2: Every play $\overline{\pi}$ that conforms to the strategy $(\Gamma_{\overline{x}}, 0, \rho_{\overline{x}})$ also conforms to the counterstrategy (Γ, γ_0, ρ) and is thus won by the environment.

Proof: The inputs x_i dictated by $\rho_{\overline{x}}$ are (singleton) subsets of the inputs that are allowed by ρ. This follows trivially from the construction of $\overline{x}$. ∎

E. Graphs and Interactive Games

We suggest to use the counterstrategy (or the countertrace, if found) in two ways in order to illustrate the reason for unrealizability. First, the user can explore the counterstrategy in an interactive game. In every step, the strategy suggests an input and the user selects an output of the system. Playing and losing the game (repeatedly) allows the user to discover the reason for which the specification is unrealizable. Second, we display a graph $\mathbb{G}$ that summarizes all possible plays in the game. Vertices in this graph correspond to state-memory pairs $(q, \gamma) \in (Q \times \Gamma)$, edges represent transitions that are allowed. The graph can be seen as a "cheat sheet" for the interactive game: it allows the user to see how the environment will react to her outputs. Thus, she may discard some choices a priori and thereby reduce the number of plays necessary to understand the cause of unrealizability.

IV. Debugging GR(1) Specifications

In this section we concretize our generic debugging method for GR(1) specifications. The class of GR(1) fulfills all the necessary premises stated in III. The environment assumption as well as the system guarantees are each represented by a set of DBWs. Our approach for debugging undesired behavior requires that the desired behavior can be transformed into a guarantee. Constructing a DBW that accepts the desired behavior is trivial when it is given as a trace. Removing guarantees reduces to removing DBWs from the according set. Removing output signals is done by existentially quantifying them in the symbolic representation of the DBWs. Realizability can be decided as shown in [19]. Synthesis of a counterstrategy for an unrealizable GR(1) specification has not been addressed before in the literature. We will show how this can be achieved in the next section. Furthermore, we address some GR(1) specific aspects of its illustration.

A. Counterstrategies for GR(1) Specifications

We derive a counterstrategy for the co-GR(1) game $\mathcal{G}_{\mathrm{env}}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathrm{Win})$ from some intermediate results in the calculation of the winning region for the environment (Player 1). The winning region for the system (Player 2) is defined in [19]. The winning region for the environment is its complement. Hence, we obtain

$$W_{\mathrm{env}}^{\mathrm{GR1}} = \mu Z \,.\, \bigcup_{j=1}^{n} \nu Y \,.\, \bigcap_{i=1}^{m} \mu X \,. \\ \big(\neg J_j^s \cup \mathsf{MX}\, Z\big) \cap \mathsf{MX}\, Y \cap (J_i^e \cup \mathsf{MX}\, X)\,. \quad (3)$$

Theorem 3: The set $W_{\mathrm{env}}^{\mathrm{GR1}}$ is the set of winning states for the environment in the co-GR(1) game $\mathcal{G}_{\mathrm{env}}^{\mathrm{GR1}}$.

In order to formulate a counterstrategy, we define Z_a to be the a-th iteration (according to Equation 2) of the fixpoint

computation of Z in Equation 3. We also define $Y_{a,j}$ as

$$\nu Y \,.\, \bigcap_{i=1}^{m} \mu X \,.\, \left(\neg J_j^s \cup \mathsf{MX}\, Z_{a-1}\right) \cap \mathsf{MX}\, Y \cap \left(J_i^e \cup \mathsf{MX}\, X\right).$$

Finally, $X_{a,j,i,c}$ is the c-th iteration of the fixpoint computation

$$\mu X \,.\, \left(\neg J_j^s \cup \mathsf{MX}\, Z_{a-1}\right) \cap \mathsf{MX}\, Y_{a-1,j} \cap \left(J_i^e \cup \mathsf{MX}\, X\right).$$

To ease notation, we also define $Z_a^{\text{new}} = Z_a \setminus Z_{a-1}$ and $X_{a,j,i,c}^{\text{new}} = X_{a,j,i,c} \setminus X_{a,j,i,c-1}$. We will further write $i \oplus 1$ for $(i \mod m) + 1$.

Let $\hat{Q} = Q \cap W_{\text{env}}^{\text{GR1}}$ be the set of states from which a counterstrategy for the co-GR(1) game exists. To obtain such a counterstrategy, we define four sub-strategies $\rho_1, \rho_2, \rho_3, \rho_4 \subseteq (\hat{Q} \times \Gamma \times \mathcal{X} \times \Gamma)$, where $\Gamma = \mathcal{I} \times \mathcal{J}$. The set $\mathcal{I} = \{1, \ldots, m\}$ stores the index of the next set J_i^e of accepting states of the environment that the play will reach. The set $\mathcal{J} = \{0, 1, \ldots, n\}$ stores the index of the set J_j^s of accepting states of the system that the environment tries to evade. The value 0 is added to store the fact that the environment has not (yet) committed to any such set. The initial memory content is $\gamma_0 = (1, 0)$.

We will build a strategy that makes sure that the play never moves from an iterate Z_i^{new} to an iterate $Z_{i'}^{\text{new}}$ with $i' > i$. Furthermore, for each i, there is a j such that if the play remains in Z_i^{new} then J_j is never visited. This j is stored in the second element of the memory. It is then easy to see that the environment always wins: If the play reaches Z_1, it is trapped, and the environment wins. In a higher iterate, either a system fairness condition is never fulfilled, or the game moves to a lower iterate. It is important that the environment takes advantage of each opportunity to take the play into a lower iterate, for otherwise the system could be able to visit J_j^s states infinitely often for all j.

Sub-strategy ρ_1 is used to take the game into a smaller iterate of Z whenever possible. This step changes the value of j. However, the system's choice of y can influence to which $Y_{a-1,j}$ the play proceeds. Thus, the environment can only choose the new value for j after the system's move. In order to remember to do so, j is set to 0:

$$\rho_1 = \left\{(q, (i, j), x, (i, 0)) \;\middle|\; \exists\, a \geq 2 : q \in Z_a^{\text{new}} \cap \mathsf{MX}_x(Z_{a-1})\right\}$$

Sub-strategy ρ_2 is applied whenever $j = 0$. It sets j to a suitable value depending on the state the play is in:

$$\rho_2 = \left\{(q, (i, 0), x, (i, j')) \;\middle|\; \exists\, a \geq 1 : q \in (Z_a^{\text{new}} \cap \mathsf{MX}_x(Y_{a,j'})) \setminus \mathsf{MX}(Z_{a-1})\right\}$$

Sub-strategy ρ_3 is applied if the play is in a state of J_i^e. A state in $J_{i\oplus 1}^e$ should be reached next (possibly in several steps), thus ρ_3 updates the content of i:

$$\rho_3 = \left\{(q, (i, j), x, (i \oplus 1, j)) \;\middle|\; j \neq 0 \;\wedge\; q \in J_i^e \;\wedge\; \exists\, a \geq 1 : q \in (Z_a^{\text{new}} \cap \mathsf{MX}_x(Y_{a,j})) \setminus \mathsf{MX}(Z_{a-1})\right\}$$

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

$$\rho_4 = \left\{(q, (i, j), x, (i, j)) \;\middle|\; j \neq 0 \;\wedge \exists\, a \geq 1, c \geq 2 : (q \in Z_a^{\text{new}} \cap X_{a,j,i,c}^{\text{new}} \cap \mathsf{MX}_x(X_{a,j,i,c-1})) \setminus \mathsf{MX}(Z_{a-1})\right\}$$

Theorem 4: In the co-GR(1) game $\mathcal{G}_{\text{env}}^{\text{GR1}}$, the strategy $\left(\mathcal{I} \times \mathcal{J}, (1, 0), \rho_{\text{env}}^{\text{GR1}}\right)$ with $\rho_{\text{env}}^{\text{GR1}} = \rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4$ is a counterstrategy.

B. Graphs and Interactive Games in case of GR(1)

The current memory content of the counterstrategy is presented to the user during the interactive game as well as in the graph $\mathbb{G}$. Knowing the index j of the set J_j^s which the environment tries to evade, the user can focus on reaching this set only. The user might indeed be able to reach a J_j^s state. However, by doing so she allows the tool to force the play into a smaller iterate of Z (using ρ_1).

V. Experimental Results

Our implementation, the specifications, and the scripts needed to reproduce the evaluations are available for download on the Anzu website[1]. Our results are summarized in Table I. All experiments were performed on an Intel Centrino 2 processor with 2×2.0 GHz and 3 GB RAM. We used two different specifications, both parametrized. The first one defines a bus arbiter [2], parameterized with the number of masters. We denote its variants by `Axy`, where `x` is the number of masters and `y` is the kind of error we introduced in order to make the specification unrealizable. With `woef` we indicate that a fairness constraint of the environment was removed, `wsf` means that a fairness constraint of the system was added, and `wst` means that we impose additional restrictions on state transitions of the system. The second specification defines a generalized buffer [3] used by n senders and two receivers. We denote its variants by `Gxy`, where `x` defines the number of senders and `y` is as before. All specification variants are satisfiable but not realizable. Their size ranges from 90 properties over 22 signals (`A2woef`) to 6004 properties over 218 signals (`G100wst`).

Columns 1 to 4 present results without minimization. Columns 1 and 2 list the times for computation of the winning region $W_{\text{env}}^{\text{GR1}}$ and the counterstrategy's relation $\rho_{\text{env}}^{\text{GR1}}$. Column 3 shows the number of vertices in the graph $\mathbb{G}$. Column 4 indicates if a countertrace was found. Columns 5 to 10 summarize results when delta debugging is applied. Column 5 gives the number of realizability checks and Column 6 relates this to the number of checks necessary with the algorithm of [5], which we reimplemented for comparison. Column 7 gives the time for minimization with the algorithm of [5]. The time needed for delta debugging is shown in Column 8, and the time savings compared to [5] are shown as speed-up factor in Column 9. Column 10 finally contains the number of vertices in $\mathbb{G}$ for the minimized specification. This number is a good

[1] http://www.iaik.tugraz.at/content/research/design_verification/anzu/

TABLE I
PERFORMANCE RESULTS. DD = "DELTA DEBUGGING".

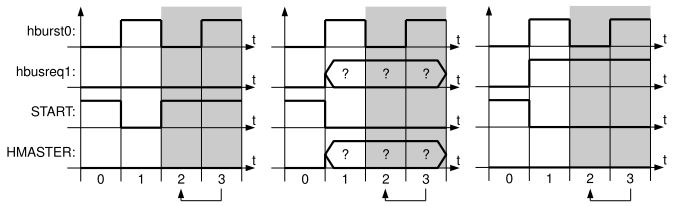
column	1	2	3	4	5	6	7	8	9	10
	without DD				with DD					
	Time: $W_{\text{env}}^{\text{GR1}}$	Time: $\rho_{\text{env}}^{\text{GR1}}$	# Vertices in $\mathbb{G}$	$\not\vdash$ found	# Checks during DD	Reduction of Checks	Time: $\hat{\varphi}$ in [5]	Time: $\hat{\varphi}$ with DD	Speed-Up Factor	# Vertices in $\mathbb{G}$
	[sec]	[sec]	[-]	[-]	[-]	[%]	[sec]	[sec]	[-]	[-]
A2woef	0.3	0.6	27	yes	47	41	5.3	0.7	7.6	5
A4woef	30	23	75	yes	56	59	284	5.1	56	13
A6woef	463	325	267	yes	58	70	7431	33	225	29
A2wsf	0.6	0.4	59	yes	37	54	8.7	0.7	12	5
A4wsf	38	22	171	yes	46	66	754	9.7	78	5
A6wsf	683	248	715	yes	47	76	6958	18	387	5
A2wst	0.4	0.3	43	yes	41	49	4.4	0.8	5.5	7
A4wst	10	18	139	yes	52	62	260	6.6	39	19
A6wst	644	410	683	yes	55	71	11170	72	155	43
G10wsf	2.1	1.8	50	no	124	19	65	53	1.2	9
G20wsf	0.8	2.4	92	no	215	42	29	118	0.3	9
G40wsf	2.0	12	176	no	473	57	165	1317	0.1	9
G100wsf	10	225	204	no			4205	>40k		
G10wst	0.2	0.2	10	yes	42	73	3.4	1.1	3.1	7
G20wst	0.5	0.6	22	yes	40	89	13	2.0	6.5	7
G40wst	2.7	4.3	40	yes	61	94	105	4.0	26	7
G100wsf	19	64	46	yes	64	99	4214	18	234	7
total	1897	1133					31470	1660	19	

indicator how simple it is to gain meaningful insights. As it can be seen, it is reduced dramatically by minimization.

In our experiments, a countertrace was found in about 80% of the cases without minimization, and in all cases after minimization. Our subjective impression is that countertraces are far easier to understand than strategies. The time for the computation of the countertrace from a given counterstrategy is negligible. Hence, with very little additional computational costs, our heuristic provides us with much simpler explanations for unrealizability in many cases.

Minimization reduces the number of formulas in the specification by 85% on average, making them much easier to analyze. Simultaneously, it raises the chance that a countertrace is found and significantly reduces the size of the graph that the user must explore, if no trace is found. Minimization does not take much time in most cases. Additionally, the times for the computation of counterstrategies, countertraces and graphs for the simplified specification are then negligible. Thus, using minimization provides results that are more helpful for the user, without additional costs in terms of CPU time in many cases.

Compared to [5], delta debugging needs about 50% fewer realizability checks in average. However, the reduction in the number of checks does not directly correlate with the time savings, as the time per check heavily depends on the complexity of the specification. Following [5], this complexity decreases quite steadily. With delta debugging there is a chance of removing many guarantees in one step. If this happens early, the performance of delta debugging is overwhelmingly superior. If significant reductions happen late, the performance may be worse, which explains the results for `G20wsf`, `G40wsf`, and `G100wsf`. However, in most cases of our experiments delta debugging is much faster. Both minimization methods lead to specifications of about the same size.



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

Fig. 4. Example: Illustrating a conflict with the design intent.

Example

In this section we present a practical example that illustrates our concepts. It specifies a bus arbiter with 2 masters. We will use lower case letters for inputs and upper case letters for outputs. The output signal `HMASTER` is set to 0 whenever the bus is currently owned by master 0, and set to 1 whenever the bus is occupied by master 1. The output `START` must be raised when the bus ownership changes. With the inputs `hbusreq0` and `hbusreq1`, the bus can be requested by master 0 and master 1, respectively. There are further signals, among them `hburst0`, but their meaning is irrelevant for this example.

Fig. 4(a) depicts a possible simulation run, simplified to signals of interest. The infinite loop is marked with gray background. Suppose the design intent was that `START=0` as long as the input `hburst0` keeps on changing. This behavior is not observed. Following our debugging approach (see Section III-A), the user specifies the design intent as shown in Fig. 4(b). This design intent is added as an additional guarantee to the specification. The new specification (A2wst in Table I) is unrealizable, which means that the design intent that was just added is in conflict with the rest of the specification.

To explain this conflict, our tool first minimizes the specification. The minimization algorithm removes 70% of the 15 output signals and 90% of the 66 formulas specifying the system. Besides the trace with which the user specified the design intent, the following guarantees remain:

$$\texttt{HMASTER=0} \wedge \texttt{START=1}, \tag{4}$$

$$\mathsf{G}((\mathsf{X}\,\texttt{START=0}) \rightarrow (\texttt{HMASTER=1} \leftrightarrow \mathsf{X}\,\texttt{HMASTER=1})) \tag{5}$$

$$\mathsf{G}\,\mathsf{F}(\texttt{HMASTER=1} \vee \texttt{hbusreq1=0}). \tag{6}$$

As long as `hburst0` keeps changing, `START` cannot be raised, since this is forbidden by the trace in Fig. 4(b). Without `START` being raised, the bus ownership cannot change (Equation 5). The bus is initially granted to master 0 (Equation 4). When the bus remains granted to master 0 (`HMASTER=0`) forever and is requested by master 1 (`hbusreq1=1`), the fairness constraint stated in Equation 6 cannot be fulfilled.

Our heuristic is able to find a countertrace that illustrates this problem. It is presented in Fig. 4(c) together with the only output trace that conforms to the design intent and fulfills Equation 4 and 5. Equation 6 is not fulfilled.

There are multiple ways to solve this conflict. Deciding which way is best depends on the design intent and can thus not be answered automatically.

VI. Conclusions and Future Work

In this work we proposed the use of counterstrategies to debug specifications that are unrealizable or in conflict with the design intent. We also discussed techniques to keep the explanations for conflicts as simple as possible.

Countertraces give much simpler explanations than normal counterstrategies, because they are independent of the opponent's moves. We presented a heuristic algorithm without exponential blow-up of the state space that is able to compute such a countertrace for some of the cases in which one exists. Our experience and experimental results show that they are a powerful tool. To the best of our knowledge, countertraces have not been mentioned in literature before.

Minimization by means of delta debugging can greatly reduce the complexity of an unrealizable specification, while still preserving the conflict. Compared to the minimization method of Cimatti et al. [5], our technique is much faster on average. Minimized specifications allow faster computations of counterstrategies, lead to simpler games and summarizing graphs, and increase the chance of finding a countertrace. Thus, performing minimization before applying other debugging techniques is crucial for larger specifications.

In the future, we plan to evaluate our techniques on bugs that were not artificially constructed, but occurred in real specification-development processes. We already started to integrate our debugging approach into RAT [18], utilizing its graphical user interface. Our goal here is to provide high usability and interoperability with other features of RAT.

Acknowledgments: We would like to thank Amir Pnueli for suggesting the use of counterstrategies. We also thank Viktor Schuppan and anonymous reviewers for valuable comments on earlier versions.

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, 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.

[3] 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 http://www.entcs.org/.

[4] Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamamenta Informaticae*, 62(2):139–169, 2004.

[5] 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.

[6] K. Claessen. A coverage analysis for safety property lists. In *Proc. Formal Methods in Computer Aided Design*, pages 139–145, 2007.

[7] S. Dellacherie. Automatic bus-protocol verification using assertions. In *GSPx*, 2004.

[8] D. Fisman, O. Kupferman, S. Seinvald, and M. Y. Vardi. A framework for inverent vacuity. In *Proc. Haifa Verification Conference (HVC)*, 2008.

[9] D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *DATE*, pages 1176–1181, 2007.

[10] 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.

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

[12] 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, Berlin, September 1999. Springer-Verlag. LNCS 1703.

[13] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[14] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science*, pages 531–542, Pittsburgh, PA, October 2005.

[15] 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.

[16] M. Leucker and T. Noll. Truth/SLC - a parallel verification platform for concurrent systems. In *Computer Aided Verification*, pages 255–259. Springer, 2001.

[17] R. Mori and N. Yonezaki. Several Realizability Concepts in Reactive Objects. *Information Modeling and Knowledge Bases*, 1993.

[18] 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.

[19] 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.

[20] 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.

[21] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[22] 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.

[23] C. Stirling. Local model checking games. In *Proc. Concurrency Theory*, pages 1–11. Springer-Verlag, 1995.

[24] L. Tan. PlayGame: A platform for diagnostic games. In *Computer Aided Verification*, pages 492–495. Springer, 2004. LNCS 3114.

[25] 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.

[26] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.