Debugging Unrealizable Specifications with Model-Based Diagnosis *

Robert Könighofer, Georg Hofferek, and Roderick Bloem

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

Abstract. Creating a formal specification for a reactive system is difficult and mistakes happen frequently. Yet, aids for specification debugging are rare. In this paper, we show how model-based diagnosis can be applied to localize errors in unrealizable specifications of reactive systems. An implementation of the system is not required. Our approach identifies properties and signals that can be responsible for unrealizability. By reduction to unrealizability, it can also be used to debug specifications which forbid desired behavior. We analyze specifications given as one set of properties, as well as specifications consisting of assumptions and guarantees. For GR(1) specifications we describe how realizability and unrealizable cores can be computed quickly, using approximations. This technique is not specific to GR(1), though. Finally, we present experimental results where the error localization precision is almost doubled when compared to the presentation of just unrealizable cores.

1 Introduction

A formal specification of a reactive system is ideally written before the implementation. This clarifies ambiguities early. Using property synthesis [11, 14, 19, 20, 23, 24], the process of implementing the specification can even be automated. This yields systems which are correct-by-construction. Formal specifications are also used for the precise communication of design intents and interface assumptions. They are even sold as intellectual properties for verification [10]. In all these scenarios, a corresponding implementation is not available when creating the specification.

Just like any other engineering process, writing a formal specification is an error-prone activity [5, 6, 15, 22]. At the same time, debugging an incorrect specification is complicated, especially if no implementation is available. This issue has received little attention in research, and there is almost no tool support available. Ideally, the specification is sound and complete. A specification is *complete* if no incorrect system (with respect to the informal design intent) conforms to it. A specification is *sound*, if all correct systems are valid implementations of the specification. For some applications the specification has to be complete, but

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

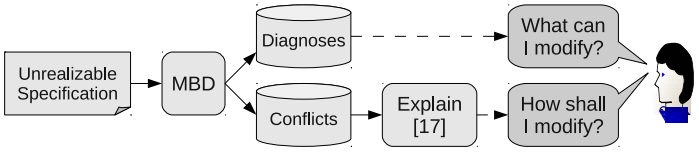

Fig. 1. Debugging an unrealizable specification with model-based diagnosis (MBD).

in any case, it must be sound. This paper addresses the debugging of unsound specifications without a corresponding implementation.

A special case of unsoundness is unrealizability. A specification is unrealizable if no system (no Mealy machine) can implement it. Note that for reactive systems, realizability is not equal to satisfiability [24]. Satisfiability means that there is one input/output trace fulfilling the specification. Realizability requires the existence of a valid output trace for *every* input trace. Additionally, outputs may depend on past and present inputs only. Our experience with the creation of complete specifications [2, 3, 1] shows that mistakes often lead to unrealizable but satisfiable specifications. Manual analysis to find the bug is time consuming or even intractable. Unlike software, there is no way to execute an unrealizable specification in order to locate the error. Debugging specifications that disallow desired behavior (i.e., unsound specifications), as well as specifications which allow undesired behavior (i.e., incomplete specifications) can be reduced to debugging unrealizability of specifications. The reader is referred to [17] for details. Hence, the approach presented in this paper can be used to debug unsoundness as well.

In this paper, we present a technique for error localization in unrealizable specifications of reactive systems, using model-based diagnosis [25] (MBD). In MBD, a system description and some incorrect behavior that contradicts this description are given. The goal is to identify components that may be responsible for the behavior. Our setting is different. We are only given an unrealizable specification. Our first contribution is to define a formalism to make MBD applicable nevertheless: Instead of diagnosing a conflict between the system description and the behavior, we diagnose the unrealizable specification, which is seen as an inherently conflicting system description. We identify properties and signals that can be weakened in order to resolve the unrealizability. Not only single-fault but also multiple-fault diagnoses can be computed. The technique is very generic and applies to most temporal logics, including LTL. It can even be used in other application domains such as the debugging of unsatisfiable formulas in a SAT solver by substituting the concept of realizability with that of satisfiability.

In our solution, diagnoses are computed using minimal unrealizable cores (parts of the specification which are unrealizable on their own) and realizability checks. This can be time consuming. As our second contribution, we therefore show how to boost performance of realizability checking and unrealizable core computation using approximations. We use Generalized Reactivity(1) specifications, but the optimization technique is again rather generic. We finally present experimental results for this class of specifications.

As illustrated in Fig. 1, MBD yields two outcomes: diagnoses and (minimal) conflicts. Diagnoses indicate possible error locations. In order to fix the error in the best way, the user also has to understand the underlying problem. This information is contained in the conflicts, which are unrealizable cores. Conflicts, and thereby the root causes for unrealizability, can be explained with counter-strategies [17]. However, understanding one conflict is not enough since a fix has to resolve all of them. The same problem exists in [5], where one unrealizable core is presented as diagnostic aid. In contrast, our approach identifies exactly those components that can be modified in a way to resolve *all* inconsistencies in the specification simultaneously. This information prevents the user from modifying parts of the specification which contribute to some but not to all conflicts. This work therefore improves and complements [5, 17].

Other related work includes detection of incompleteness in specifications both regarding a given implementation [4, 13, 15] and with the specification analyzed stand-alone [6, 8, 12]. In contrast, our work is focused on specifications that are not sound. MBD has been successfully applied to various software- and hardware debugging problems [7, 21, 27, 26]. The setting considered in this work deviates from the classical setting in that we are not given a system description and a conflicting observation, but only an inherently conflicting system description.

The remainder of this paper is organized as follows. Section 2 explains existing concepts and establishes some notation. Section 3 introduces our debugging method in a generic way. An efficient implementation for Generalized Reactivity(1) specifications is described in Section 4. Experimental results are presented in Section 5. Section 6 concludes the work.

2 Preliminaries

2.1 Model-Based Diagnosis

Model-based diagnosis [9, 25] (MBD) is a technique for error localization in a system. The explanation in this section follows [25]. Let SD be a description of the correct system behavior and let OBS be an observation of an erroneous behavior. Both SD and OBS are sets of logical sentences. The system consists of components COMP. A component $c \in \mathsf{COMP}$ can behave abnormally (denoted $\mathsf{AB}(c)$) or normally (denoted $\neg\,\mathsf{AB}(c)$). SD consists of component descriptions of the form $\neg\,\mathsf{AB}(c) \Rightarrow N_c$, where N_c defines the normal behavior of component c, and a description of their interplay. The observation contradicts the system description, i.e., it would not be possible if all components behaved normally. 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 contradiction. The goal of MBD is to identify sets of components that may have caused the erroneous behavior OBS. Such sets are called diagnoses. Formally, $\Delta \subseteq \mathsf{COMP}$ is a *diagnosis* iff it is a minimal set such that

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

is consistent. Minimal means that no subset $\Delta' \subset \Delta$ is a diagnosis. A diagnosis Δ with $|\Delta| = 1$ is called a *single-fault diagnosis*. Diagnoses can be computed with

conflicts. A *conflict* is a set $C \subseteq \mathsf{COMP}$ so that

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

is inconsistent. That is, a conflict is a set of components that cannot all behave normally. A conflict C is *minimal* if no $C' \subset C$ is a conflict. A diagnosis must explain all conflicts, so it must have at least one element in common with every conflict. This relation can be formalized with hitting sets. A *hitting set* for a collection $\mathcal{K}$ of sets is a set H for which $\forall K \in \mathcal{K} \,.\, H \cap K \neq \emptyset$ holds. A hitting set H is *minimal* if no subset $H' \subset H$ is a hitting set. A set $\Delta \subseteq \mathsf{COMP}$ is a diagnosis iff Δ is a minimal hitting set for the set of all conflicts. Equivalently, $\Delta \subseteq \mathsf{COMP}$ is a diagnosis iff it is a minimal hitting set for the set of all *minimal* conflicts. Hence, computing diagnoses reduces to computing minimal hitting sets for the collection of minimal conflict sets. An algorithm is presented in [25]. It triggers conflict computations on-the-fly. Diagnoses are generated in order of increasing cardinality. That is, simpler explanations are produced first.

2.2 Games and the μ-Calculus

Games and the μ-calculus will be used to define realizability and approximations thereof. A *(finite state, two player) game* is a tuple $\mathcal{G} = (Q, \Sigma, T, q_0, \mathsf{Win})$, where Q is a finite set of states, Σ is some finite alphabet, $T : Q \times \Sigma \to Q$ is a deterministic and complete transition function, $q_0 \in Q$ is the initial state, and $\mathsf{Win} : Q^\omega \to \{\mathsf{false}, \mathsf{true}\}$ is the winning condition. We assume that $\Sigma = \mathcal{X} \times \mathcal{Y}$, where $\mathcal{X} = 2^X$, $\mathcal{Y} = 2^Y$, and X and Y are two disjoint sets of Boolean variables. The game is played by two players, the *environment* and the *system*. A *play* $\overline{\pi}$ of $\mathcal{G}$ is defined as an infinite sequence of states $\overline{\pi} = q_0 q_1 q_2 \ldots \in Q^\omega$ such that $q_{i+1} = T(q_i, \sigma_i)$. In each step, the letters $\sigma_i = (x_i, y_i)$ are 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 $\overline{\pi} = q_0 q_1 q_2 \ldots$ is *won* by the system if $\mathsf{Win}(\overline{\pi})$ holds. A state $q \in Q$ is called *winning*, if starting from this state, the system can, for all possible inputs which it might be presented, choose outputs in each step, such that the play is won. The set of winning states $W \subseteq Q$ is called the *winning region* of the system.

The (propositional) μ-calculus [18] is an extension of propositional logic with a least fixpoint operator μ and a greatest fixpoint operator ν. We use this calculus, extended with two preimage operators MX^{s} and MX^{e}, to describe fixpoint computations over sets $Q' \subseteq Q$ of states in a game $\mathcal{G}$. Let Var be a set of variables ranging over subsets of Q. Every variable $Z \in \mathrm{Var}$ and every set $Q' \subseteq Q$ of states is a valid μ-calculus formula. Also, if 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 $Z \in \mathrm{Var}$, the μ-calculus formulas $\mu Z \,.\, R(Z)$, $\nu Z \,.\, R(Z)$, $\mathsf{MX}^{\mathrm{s}}(R)$, and $\mathsf{MX}^{\mathrm{e}}(R)$

are μ-calculus formulas. The latter are defined as

$$\begin{aligned}
\mu Z.R(Z) &= \bigcup_i Z_i, \quad \text{where } Z_0 = \emptyset \text{ and } Z_{i+1} = R(Z_i), \\
\nu Z.R(Z) &= \bigcap_i Z_i, \quad \text{where } Z_0 = Q \text{ and } Z_{i+1} = R(Z_i), \\
\mathsf{MX}^{\mathrm{s}}(R) &= \{q \in Q \mid \forall x \in \mathcal{X} \,.\, \exists y \in \mathcal{Y} \,.\, T(q,(x,y)) \in R\}\,, \text{ and} \\
\mathsf{MX}^{\mathrm{e}}(R) &= \{q \in Q \mid \exists x \in \mathcal{X} \,.\, \forall y \in \mathcal{Y} \,.\, T(q,(x,y)) \in R\}\,.
\end{aligned}$$

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

2.3 Specifications for Reactive Systems

A reactive system is a Mealy machine that continuously interacts with its environment via a set X of inputs and a set Y of outputs. Without loss of generality we assume that all signals are Boolean. We consider two kinds of specifications. The first one, denoted with φ_P, consists of a set P of properties. Let $\overline{\sigma} \models p$ denote that trace $\overline{\sigma} \in \Sigma^\omega$ fulfills property $p \in P$. Then $\overline{\sigma}$ fulfills φ_P, written $\overline{\sigma} \models \varphi_P$, iff for all $p \in P$, $\overline{\sigma} \models p$ holds. We assume two special properties $\top$ and $\bot$, where $\overline{\sigma} \models \top$ and $\overline{\sigma} \not\models \bot$ for every $\overline{\sigma}$. The second kind of specifications, denoted $\varphi_{A,G}$, is given as a pair (A, G), where A is a set of environment assumptions and G is a set of system guarantees. If all assumptions are met, the specification requires the system to fulfill all guarantees, i.e.,

$$\overline{\sigma} \models \varphi_{A,G} \quad \text{iff} \quad \big(\text{for all } a \in A,\, \overline{\sigma} \models a\big) \text{ implies } \big(\text{for all } g \in G,\, \overline{\sigma} \models g\big). \tag{3}$$

A Generalized Reactivity(1) [23] specification (GR(1), for short) is of the form $\varphi_{A,G}$. It can be transformed [23] into a game $\mathcal{G}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathsf{Win})$ with m special sets $J_i^e \subseteq Q$ and n sets $J_j^s \subseteq Q$ of states. Every J_i^e is a set of accepting states for the environment. They correspond to environment assumptions. Analogously, the sets J_j^s correspond to system guarantees. The winning region $W_{\mathrm{sys}}^{\mathrm{GR1}}$ of the system is characterized [23] by the formula

$$W_{\mathrm{sys}}^{\mathrm{GR1}} = \nu Z \,.\, \bigcap_{j=1}^{n} \mu Y \,.\, \bigcup_{i=1}^{m} \nu X \,.\, J_j^s \cap \mathsf{MX}^{\mathrm{s}}\, Z \cup \mathsf{MX}^{\mathrm{s}}\, Y \cup \neg J_i^e \cap \mathsf{MX}^{\mathrm{s}}\, X. \tag{4}$$

A GR(1) specification is realizable iff $q_0 \in W_{\mathrm{sys}}^{\mathrm{GR1}}$ in the corresponding game.

2.4 Minimization Algorithms

We will use two different minimization algorithms for unrealizable core computation. *Delta Debugging* [28] is an algorithm to isolate the cause of a failure. Given a procedure test and some input C that makes test fail (denoted $\mathsf{test}(C) = \text{✘}$),

it computes a minimal input $\hat{C} = \mathsf{ddmin}^{\mathsf{test}}(C)$, with $\hat{C} \subseteq C$, which still makes test fail. In the best case, ddmin requires only a logarithmic number of checks whether test fails. In the worst case, a quadratic number of checks is needed. The reader is referred to [28] for detailed information on the algorithm.

The second minimization algorithm [5] we use removes one element after the other from the given set C. If test passes without a particular element, this element is part of the unrealizable core and is added again. Otherwise, the algorithm proceeds without this element. Thus, exactly $|C|$ invocations of test are required. This algorithm will be denoted $\mathsf{linMin}^{\mathsf{test}}$.

3 Model-Based Diagnosis for Unrealizability

In this section, we describe how MBD can be used to perform error localization in an unrealizable specification. The difference to standard MBD is that we do not diagnose a conflict between a system description and an observation but an inherently conflicting system description. Our technique can be applied to many specification languages. The prerequisites are that properties can be removed from specifications, output signals can be existentially quantified in properties, and that a decision procedure for the realizability of a specification is available.

3.1 Property Specifications

In this section, we assume that the unrealizable specification is given as a set P of properties for a system with inputs X and outputs Y. Since every property typically represents a relatively self-contained and independent aspect of the system behavior, we define every property to be a component. That is, $\mathsf{COMP}_P = P$. Furthermore, we define the system description SD_P to be the tuple (P, X, Y). There is no observation in our setting. As an abstraction for the notion of logical consistency, we define a function $\mathsf{consistent}_{\mathsf{SD}_P} : 2^{\mathsf{COMP}_P} \rightarrow \{\mathsf{true}, \mathsf{false}\}$. Intuitively, $\mathsf{consistent}_{\mathsf{SD}_P}(P')$ with $P' \subseteq \mathsf{COMP}_P$ gives true iff the system description SD_P is consistent under the assumption that all components $c \in P'$ behave normally. More formally, we transform φ_P to $\tilde{\varphi}_P$ such that for every trace $\overline{\sigma} \in \Sigma^\omega$, $\overline{\sigma} \models \tilde{\varphi}_P$ iff for all $p \in P$, $\neg\,\mathsf{AB}(p)$ implies $\overline{\sigma} \models p$. Moreover, $\mathsf{consistent}_{\mathsf{SD}_P}(P') = \mathsf{true}$ iff $\tilde{\varphi}_P$ is consistent with $\neg\,\mathsf{AB}(p')$ for all $p' \in P'$. That is, abnormal properties do not have to be fulfilled in $\tilde{\varphi}_P$, so we can think of them as removed from the specification. Let realizable be a function which decides the realizability of a specification. Then, we can finally use the notion of realizability to define consistency as $\mathsf{consistent}_{\mathsf{SD}_P}(P') = \mathsf{realizable}(P')$.

Lemma 1. *The function* $\mathsf{consistent}_{\mathsf{SD}_P}$ *is monotonic, i.e.,*
$\forall P'' \subseteq P' \subseteq P \,.\, \neg\,\mathsf{consistent}_{\mathsf{SD}_P}(P'') \Rightarrow \neg\,\mathsf{consistent}_{\mathsf{SD}_P}(P')$.

This follows from the fact that adding properties to an unrealizable specification preserves unrealizability. In analogy to Eq. 2, a conflict is a set $C_P \subseteq COMP_P$

such that $\mathsf{consistent}_{\mathsf{SD}_P}(C_P) = \mathsf{false}$, i.e., $\mathsf{realizable}(C_P) = \mathsf{false}$. A minimal conflict is therefore what is also called a minimal unrealizable core of the specification [5]. Corresponding to Eq. 1, a diagnosis is finally a minimal set $\Delta_P \subseteq \mathsf{COMP}_P$ such that $\mathsf{consistent}_{\mathsf{SD}_P}(\mathsf{COMP}_P \setminus \Delta_P) = \mathsf{true}$.

Observation 1 *Let C_P be a conflict for an unrealizable specification $\varphi_P = P$. In order to make φ_P realizable, at least one property $p \in C_P$ must be modified.*

One cannot obtain a realizable specification by modifying properties $p \in (P \setminus C_P)$ only, because the set C_P of properties forms an unrealizable specification already. Any superset of properties will be unrealizable as well (cf. Lemma 1). Informally speaking, there must be something wrong with at least one conflict element.

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

Clearly, the specification becomes realizable if all $p \in \Delta_P$ are replaced by $\top$. Minimality follows from the definition.

Example 1. The specification $\varphi_P = \{p_1, p_2, p_3\}$ with[1] $p_1 = \mathsf{G}(I_1{=}0 \Rightarrow O_1{=}1)$, $p_2 = \mathsf{G}(I_1{=}1 \Rightarrow O_1{=}0)$, $p_3 = \mathsf{GF}(I_1 \Leftrightarrow O_1 \wedge I_2 \Leftrightarrow O_2)$, $X = \{I_1, I_2\}$, and $Y = \{O_1, O_2\}$ is unrealizable because p_1 and p_2 require that output O_1 is always the negation of input I_1, while p_3 requires these two signals to be equal infinitely often. The specification contains the conflicts $C_1 = \{p_1, p_2, p_3\}$, $C_2 = \{p_1, p_3\}$, and $C_3 = \{p_2, p_3\}$, the latter two being minimal. Conflict C_2 states that no system can implement p_1 and p_3 at the same time, independent of p_2. Analogously for C_3. The sets $\Delta_1 = \{p_3\}$ and $\Delta_2 = \{p_1, p_2\}$ are the minimal hitting sets for the collection of all minimal conflicts, since they share at least one element with both C_2 and C_3. Hence, Δ_1 and Δ_2 are (the only) diagnoses. According to Observation 2, the user can therefore weaken either property p_3 or both properties p_1 and p_2 in order to make the specification realizable. Suppose now that the user is only given the unrealizable core C_2. In this case, she might attempt to modify p_1 alone in order to resolve the unrealizability. However, this is not possible because no matter how p_1 is changed, even if it is $\top$, there is still a conflict between p_2 and p_3. Our debugging approach takes this circumstance into account by combining information about all unrealizable cores. This allows for more precise error localization than the presentation of a single unrealizable core.

3.2 Assumption/Guarantee Specifications

Let $\varphi_{A,G} = (A, G)$ be an unrealizable specification consisting of the environment assumptions A and the system guarantees G.

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.*

[1] We use the common LTL syntax, where G represents the temporal operator "always" and F denotes "eventually".

This follows directly from Eq. 3, since replacing any assumption with $\bot$ would give a realizable specification. In other words, it does not make sense to search for assumptions that can be modified in order to obtain a realizable specification, because every assumption can be modified in such a way. But not every guarantee can be altered to obtain a realizable specification. Hence, we define $\mathsf{COMP}_G = G$, implicitly assuming that all assumptions are correct. Furthermore, we define the system description $\mathsf{SD}_{A,G}$ to be be the tuple (A, G, X, Y), and $\mathsf{consistent}_{(A,G,X,Y)}(G') = \mathsf{realizable}((A, G'))$ for $G' \subseteq G$. It follows that a conflict is a set $C_G \subseteq G$ of guarantees such that $\mathsf{realizable}((A, C_G)) = \mathsf{false}$. Consequently, a diagnosis is a minimal set $\Delta_G \subseteq G$ such that $\mathsf{realizable}((A, G \setminus \Delta_G)) = \mathsf{true}$.

Lemma 1 and the Observations 1 and 2 apply in this case as well, with the obvious adaptations. Since specifications φ_P are special cases of specifications $\varphi_{A,G}$ (with $A = \emptyset$), we restrict our investigations to the latter form in the following.

3.3 Diagnosing Variables

In the previous section we showed how to identify (sets of) guarantees that can be weakened in order to obtain a realizable specification. In this section we define a formalism to identify *signals* that may be over-constrained, i.e., signals that can be less restricted in order to resolve the unrealizability. We also show how the two approaches can be combined.

Let $\varphi_{A,G} = (A, G)$ be an unrealizable specification over the 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. As already introduced rather informally in [17], we can define an existential quantification $(A, \exists Y' \,.\, G)$ of the outputs $Y' \subseteq Y$ in the guarantees of $\varphi_{A,G}$ with semantics

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

The existential quantification $\exists Y' \,.\, g$ in one single guarantee $g \in G$ is defined as

$$\overline{\sigma} \models \exists Y' \,.\, g \quad \text{iff} \quad \exists y'_0 y'_1 y'_2 \ldots \in \left(2^{Y'}\right)^\omega .(x_0, y_0^E)(x_1, y_1^E)(x_2, y_2^E) \ldots \models g,$$

where $y_i^E = (y_i \setminus Y') \cup y'_i$ for all $i \geq 0$. Informally speaking, an existential quantification $\exists Y' \,.\, G$ of the variables $Y' \subseteq Y$ in all guarantees $g \in G$ removes all restrictions on the variables $y \in Y'$. The specification $(A, \exists Y' \,.\, G')$ allows arbitrary values for all outputs $y \in Y'$ in all time steps. Also note that the quantification is performed on every single guarantee, and not on the conjunction of all guarantees. With $\mathsf{COMP}_Y = Y$ and $Y' \subseteq Y$, we define $\mathsf{consistent}_{(A,G,X,Y)}(Y') = \mathsf{realizable}((A, \exists Y \setminus Y' \,.\, G))$. Consequently, a conflict is a set $C_Y \subseteq Y$ of outputs such that $\mathsf{realizable}((A, \exists Y \setminus C_Y \,.\, G)) = \mathsf{false}$. Finally, a diagnosis is a minimal set $\Delta_Y \subseteq Y$ such that $\mathsf{realizable}((A, \exists \Delta_Y \,.\, G)) = \mathsf{true}$. Every Δ_Y contains signals that may be over-constrained, because removing restrictions on these signals resolves the unrealizability.

An alternative definition $\tilde{\exists}$ of the existential quantification operates on the conjunction of all guarantees instead of every guarantee in isolation. As an example where this makes a difference, consider the specification $\varphi_{A,G} = (A, G) =$

$(\{\top\}, \{\mathsf{G}(O), \mathsf{G}(\neg O)\})$, where O is the only output. We have that $(A, \exists\{O\}\,.\,G)$ is realizable (it allows all traces), while $(A, \tilde{\exists}\{O\}\,.\,G)$ 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 quantification in the conjunction of all guarantees means is typically much more difficult, because complex dependencies between the guarantees may exist.

The approaches of diagnosing properties and signals can also be combined by defining $\mathsf{COMP} = G \cup Y$. With $\mathsf{consistent}_{(A,G,X,Y)}(B) = \mathsf{realizable}((A, \exists Y \setminus B\,.(G \cap B)))$ and $B \subseteq \mathsf{COMP}$ we have that $C_{Y,G} \subseteq (Y \cup G)$ is a conflict iff $\mathsf{realizable}((A, \exists Y \setminus C\,.(G \cap C))) = \mathsf{false}$. A diagnosis is now a minimal set $\Delta_{Y,G} \subseteq (Y \cup G)$ such that $\mathsf{realizable}((A, \exists(Y \cap \Delta)\,.(G \setminus \Delta))) = \mathsf{true}$.

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

Proof. $\mathsf{realizable}((A, \exists\Delta_Y\,.\,G)) \Rightarrow \mathsf{realizable}((A, \exists(Y \cap \Delta_Y)\,.(G \setminus \Delta_Y)))$ and
$(\forall\Delta'_Y \subset \Delta_Y\,.\,\neg\,\mathsf{realizable}((A, \exists\Delta'_Y\,.\,G))) \Rightarrow$
$(\forall\Delta'_Y \subset \Delta_Y\,.\,\neg\,\mathsf{realizable}((A, \exists(Y \cap \Delta'_Y)\,.(G \setminus \Delta'_Y))))$ since $Y \cap \Delta''_Y = \Delta''_Y$ and $G \setminus \Delta''_Y = G$ for all $\Delta''_Y \subseteq \Delta_Y$. Analogously for Δ_G.

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

Example 2. Let $\varphi_{A,G} = (\emptyset, \{p_1, p_2, p_3\})$, where p_1, p_2, and p_3 are defined as for Example 1. The minimal conflicts are $C_1 = \{p_1, p_3, O_1\}$ and $C_2 = \{p_2, p_3, O_1\}$. The sets $\Delta_1 = \{p_3\}$, $\Delta_2 = \{p_1, p_2\}$, and $\Delta_3 = \{O_1\}$ form the minimal hitting sets for the collection $\{C_1, C_2\}$ of minimal conflicts, and hence also the diagnoses for the unrealizable specification $\varphi_{A,G}$. Compared to Example 1, we obtain $\Delta_3 = \{O_1\}$ as additional diagnosis. That is, when using not only guarantees but also outputs as components for diagnosis, we also get the explanation that O_1 may be over-constrained. No diagnosis involves O_2, so O_2 does not contribute to the unrealizability. Properties are formulated in terms of signals. Therefore, having both properties and signals as diagnoses allows to track down the error from both these dimensions. For instance, in this example, it is natural to assume that there is something wrong with O_1 in p_3 but not with O_2 in p_3.

3.4 Implementation

An implementation of the diagnosis approach as described in the previous section is straightforward. The only prerequisite is that a decision procedure for the realizability of a specification has to be available. We use the algorithm of [25] to compute diagnoses via a hitting set tree. It requires a procedure to compute conflicts not containing a given list of components, if such a conflict exists. This

can be implemented with a single realizability check. However, the algorithm performs better if the computed conflicts are minimal. Such a procedure can be implemented as

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

where $\mathsf{min}_{\mathsf{SD}}(M)$, with $M \subseteq \mathsf{COMP}$ and $\neg\, \mathsf{consistent}_{\mathsf{SD}}(M)$, returns a set $\hat{M} \subseteq M$ such that $\neg\, \mathsf{consistent}_{\mathsf{SD}}(\hat{M})$ and $\forall M' \subset \hat{M} \,.\, \mathsf{consistent}_{\mathsf{SD}}(M')$. The procedure $\mathsf{min}_{\mathsf{SD}}$ can be implemented, e.g., as $\mathsf{min}_{\mathsf{SD}}(M) = \mathsf{ddmin}^{\mathsf{test}_{\mathsf{SD}}}(M)$ or $\mathsf{min}_{\mathsf{SD}}(M) = \mathsf{linMin}^{\mathsf{test}_{\mathsf{SD}}}(M)$ with $\mathsf{test}_{\mathsf{SD}}(M') = \text{✘} \Leftrightarrow \neg\, \mathsf{consistent}_{\mathsf{SD}}(M')$. Experiments [17] show that ddmin is often much faster than linMin. When using ddmin, the monotonicity of test (cf. Lemma 1) can be exploited to speed up computation [28]: all encountered sets M' for which $\mathsf{test}_{\mathsf{SD}}(M') \neq \text{✘}$ holds are stored. If a subset M'' of a stored set M' is tested, $\mathsf{test}_{\mathsf{SD}}(M'') \neq \text{✘}$ can be returned without actually invoking the realizability check.

4 Efficient Implementation for GR(1) Specifications

In our framework, diagnoses computation requires many unrealizable core computations, which in turn require lots of realizability checks. Thus, it is of utmost importance that these procedures are implemented efficiently. In this section, we show how the performance of these operations can be improved, using GR(1) specifications as an example. For GR(1) specifications, realizability can be decided by constructing a game, computing the winning region for the system, and checking if the initial state is contained in this winning region. This applies to many other specification languages as well. We use approximations of the winning region to define approximations of realizability. The only GR(1)-specific part is how these approximations of the winning region are defined.

A procedure $\mathsf{realizable}_O(\varphi)$ is called an *over-approximation* of $\mathsf{realizable}(\varphi)$ iff $\forall \varphi \,.\, \mathsf{realizable}(\varphi) \Rightarrow \mathsf{realizable}_O(\varphi)$. Such an over-approximation can be used to compute a minimal unrealizable core in two steps. In the first step, an over-approximation C' of the core is computed using a minimization algorithm which repeatedly applies $\mathsf{realizable}_O$. In a second step, C' is further reduced to the exact core C by repeatedly applying the exact realizability check. If a procedure $\mathsf{realizable}_O$ can be found which is both fast and accurate, this two-step approach can increase the performance of the unrealizable core computation significantly. The leverage comes from the fact that the expensive exact checks are performed on relatively small subsets of the specification only.

For GR(1) specifications, we can define $\mathsf{realizable}^{\mathrm{GR1}}(\varphi) \Leftrightarrow q_0 \in W^{\mathrm{GR1}}_{\mathrm{sys}}$. See Section 2.3. In order to obtain approximations to this procedure, we define the

following sets of states in the GR(1) game $\mathcal{G}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathsf{Win})$:

$$A^{\mathrm{GR1}}_{\mathrm{sys}} = \mu X \,.\, A' \cup \mathsf{MX}^{\mathrm{s}}(X) \quad \text{with} \quad A' = \bigcup_{i=1}^{m} \nu Y \,.\, \neg J^e_i \cap \mathsf{MX}^{\mathrm{s}}(Y),$$

$$B^{\mathrm{GR1}}_{\mathrm{sys}} = \mu X \,.\, B' \cup A^{\mathrm{GR1}}_{\mathrm{sys}} \cup \mathsf{MX}^{\mathrm{s}}(X) \quad \text{with} \quad B' = \nu Y \,.\, \mathsf{MX}^{\mathrm{s}}(Y) \cap \bigcap_{j=1}^{n} J^s_j \text{ , and}$$

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

These sets were chosen such that they are close to $W^{\mathrm{GR1}}_{\mathrm{sys}}$ or $Q \setminus W^{\mathrm{GR1}}_{\mathrm{sys}}$, yet fast to compute. Some more approximations were tried, but the listed ones showed the best performance in experiments. $A^{\mathrm{GR1}}_{\mathrm{sys}}$ contains all states from which the system can force the environment to violate one assumption. The set B' consists of all states from which the system can enforce to stay in the intersection of all J^s_j forever, thus fulfilling all system guarantees. $B^{\mathrm{GR1}}_{\mathrm{sys}}$ contains the set of states from which the system can force the play into a state of B' or $A^{\mathrm{GR1}}_{\mathrm{sys}}$. Clearly, $A^{\mathrm{GR1}}_{\mathrm{sys}} \subseteq W^{\mathrm{GR1}}_{\mathrm{sys}}$ and $B^{\mathrm{GR1}}_{\mathrm{sys}} \subseteq W^{\mathrm{GR1}}_{\mathrm{sys}}$. C' is the set of states from which the environment can enforce to stay in the intersection of all sets J^e_i of accepting states of the environment but outside one particular set J^s_j of accepting states of the system. Hence, $C^{\mathrm{GR1}}_{\mathrm{env}} \subseteq (Q \setminus W^{\mathrm{GR1}}_{\mathrm{sys}})$. With these sets, we define $\mathsf{realizable}^{\mathrm{GR1}}_O(\varphi) \Leftrightarrow q_0 \notin C^{\mathrm{GR1}}_{\mathrm{env}}$ as an over-approximation of $\mathsf{realizable}^{\mathrm{GR1}}$, and

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

as a more efficient implementation of $\mathsf{realizable}^{\mathrm{GR1}}$. Finally, we have

$$\mathsf{cNot}^{\mathrm{GR1}}_{\mathsf{SD}}(B) = \begin{cases} \text{None} & \text{if } \mathsf{realizable}^{\mathrm{GR1}}_E((A, \exists(Y \cap B) \,.\, G \setminus B)) \\ \mathsf{min}^{\mathrm{GR1}}_{\mathsf{SD}}((G \cup Y) \setminus B) & \text{otherwise,} \end{cases}$$

with $\mathsf{min}^{\mathrm{GR1}}_{\mathsf{SD}}(M) = \mathsf{linMin}^{\mathsf{test}^2_{\mathsf{SD}}}(\mathsf{ddmin}^{\mathsf{test}^1_{\mathsf{SD}}}(M))$. The algorithm ddmin operates on $\mathsf{test}^1_{\mathsf{SD}}$ defined as $\mathsf{test}^1_{\mathsf{SD}}(M') = \text{✘} \Leftrightarrow \neg\, \mathsf{realizable}^{\mathrm{GR1}}_O((A, \exists(Y \setminus M') \,.\, G \cap M'))$, and thus returns an over-approximation of the minimal unrealizable core. This over-approximation is further reduced by linMin, operating on $\mathsf{test}^2_{\mathsf{SD}}(M') = \text{✘} \Leftrightarrow \mathsf{realizable}^{\mathrm{GR1}}_E((A, \exists(Y \setminus M') \,.\, G \cap M'))$, and yielding the exact core. We use linMin instead of ddmin in the second step, because ddmin does not perform well when given an almost minimal set. It would waste many checks until the granularity is high enough to actually remove elements. As another performance optimization, we use early termination in the fixpoint computations. That is, if realizability or unrealizability is implied by an iterate of a fixpoint already, we abort the computation. Furthermore, for $\mathsf{realizable}^{\mathrm{GR1}}_E$, we use the set $Q \setminus C^{\mathrm{GR1}}_{\mathrm{env}}$ of states as a starting point for the outermost greatest fixpoint computation of $W^{\mathrm{GR1}}_{\mathrm{sys}}$.

5 Experimental Results

We implemented the diagnosis approach for the class of GR(1) specifications inside Marduk, the back-end of RATSY [1]. The implementation[2] as well as the scripts[3] to reproduce our performance results are available for download. In this section we first give an example to illustrate the usefulness of our debugging approach. After that, we provide performance results for some more benchmarks.

5.1 Example

Our example specification (`G5wst2` in Table 1) describes a generalized buffer connecting 5 senders to 2 receivers. It consists of 24 signals and 110 properties. We use lower case letters for inputs and upper case letters for outputs. The inputs `stob_req`i, with $0 \leq i \leq 4$, are used to signal that sender i requests to send data. With the outputs `BTOS_ACK`i the buffer can acknowledge to sender i that sending is allowed. The buffer also communicates with a FIFO storage unit. The output `ENQ` is set if data should be enqueued into the FIFO. All other signals are not relevant for this example. A guarantee $\mathsf{G}(\texttt{ENQ=0})$ has been added artificially to make the specification unrealizable. This guarantee forbids the buffer to enqueue data into the FIFO, which makes it impossible to handle any requests by the senders. An unrealizable core contains the system guarantees

$$\texttt{BTOS\_ACK4=0} \wedge \texttt{ENQ=0} \wedge \texttt{DEQ=0} \tag{5}$$

$$\mathsf{G}((\texttt{BTOS\_ACK4=0} \wedge \mathsf{X}\,\texttt{BTOS\_ACK4=1}) \Rightarrow \mathsf{X}\,\texttt{ENQ=1}) \tag{6}$$

$$\mathsf{G}(\texttt{ENQ=0}) \tag{7}$$

$$\mathsf{G}\,\mathsf{F}(\texttt{stob\_req4=1} \Leftrightarrow \texttt{BTOS\_ACK4=1}) \tag{8}$$

$$\mathsf{G}((\texttt{rtob\_ack0=1} \wedge \mathsf{X}\,\texttt{rtob\_ack0=0}) \Rightarrow \mathsf{X}\,\texttt{DEQ=1}) \tag{9}$$

$$\mathsf{G}((\texttt{rtob\_ack1=1} \wedge \mathsf{X}\,\texttt{rtob\_ack1=0}) \Rightarrow \mathsf{X}\,\texttt{DEQ=1}) \tag{10}$$

$$\mathsf{G}(\texttt{empty=1} \Rightarrow \texttt{DEQ=0}) \tag{11}$$

and the outputs `BTOS_ACK4`, `ENQ`, and `DEQ`. This means that the system could not fulfill the listed guarantees even if it could set all outputs other than `BTOS_ACK4`, `ENQ`, and `DEQ` completely arbitrarily. This is the case because `BTOS_ACK4=0` initially (Eq. 5). It cannot change to 1 due to Eq. 6 and 7. If `stob_req4=1` forever, then the fairness guarantee stated in Eq. 8 cannot be fulfilled. The remaining guarantees (Eq. 9 to 11) are in the core because without them the system could enforce a violation of one environment assumption.

When the user is given this core (as suggested by [5]), or even an explanation [17] thereof, she has to face the following problem: She is presented only one inconsistency in the specification, but many more may exist, and she has to resolve all of them. There are often many ways to fix the presented conflict but only a few ways to fix all conflicts in the specification simultaneously. Our

[2] http://rat.fbk.eu/ratsy/index.php/Main/Download
[3] http://www.iaik.tugraz.at/content/research/design_verification/others/

approach identifies guarantees and outputs which can be modified to resolve *all* conflicts. In our example, the user could learn from the presented core that Eq. 6 should be weakened. However, this does not make the original specification realizable, since other conflicts not containing Eq. 6 exist. The same argument can be made for Eq. 8. In fact, identical unrealizable cores exist with `BTOS_ACK`i instead of `BTOS_ACK4` for all $0 \leq i \leq 3$. What all conflicts have in common is the guarantee in Eq. 7 (the error we introduced). Our algorithm identifies exactly this guarantee and the signal `ENQ` as the only single-fault diagnoses. Hence, it provides much more accuracy in error localization than unrealizable cores or explanations thereof.

5.2 Performance Evaluation

We evaluated our diagnosis approach using the same benchmarks as in [16, 17]. These contain mutants of two parameterized specifications. One is an arbiter for the ARM AHB bus [2], parameterized with the number of masters it can handle. Mutants are denoted `A`nei, where n is the number of masters, e is the error that was introduced artificially, and i is a running index to distinguish different modifications of the same kind. The term `woef` for e means that a fairness assumption was removed, `wsf` means that a fairness guarantee was added, and `wst` indicates that the specification was augmented with a safety guarantee. The second specification defines a generalized buffer [3], connecting n senders to two receivers. Mutants are denoted `G`nei with the same syntax. All specification mutants have between 90 and 6004 properties and 22 to 218 signals. They are all satisfiable but unrealizable. The experiments were performed on an Intel Centrino 2 processor with 2×2.0 GHz, 3 GB RAM, running 32-bit Linux.

Table 1 summarizes performance results. The Columns 1 to 3 give results for one minimal conflict (= unrealizable core) computation, using the algorithm of [17]. Column 1 gives the size of the conflict, Column 2 the number of guarantees in the conflict, and Column 3 the time needed for conflict computation. The Columns 4 to 8 summarize results for single-fault diagnoses computation. The number of single-fault diagnoses is given in Column 4. The next column lists the count of diagnoses that are guarantees. The time for diagnosis computation as described in Section 3.4 with `ddmin` as a minimization algorithm is shown in Columns 6. Column 7 presents the time for exactly the same computation, but when using the performance optimizations with approximations of realizability, as introduced in Section 4. Column 8 presents the according speed-up factor due to these optimizations. The Columns 9 to 12 finally give the number of diagnoses with at most two and three elements and the according computation times, respectively. Entries preceded by '>' indicate time-outs.

Our experimental results underline three statements. First, and most importantly, MBD gives a higher precision for error localization than the computation of a single unrealizable core. Single-fault diagnoses computation produces 40 % less fault candidates than unrealizable core computation, where every unrealizable core element is taken as a fault candidate (Column 1 versus Column 4). That is, 40 % of the unrealizable core elements cannot be modified in such a way

Table 1. Performance Results.

Column	1	2	3	4	5	6	7	8	9	10	11	12
	One Conflict			Single-Fault Diagnoses					Mult. Fault Diagnoses			
	$\lvert C\rvert$	$\lvert C \cap G\rvert$	time (optimized)	$\lvert\{\Delta_1 \mid \lvert\Delta_1\rvert = 1\}\rvert$	$\lvert\{\Delta_1 \mid \Delta_1 \subseteq G\}\rvert$	time	time, optimized	speed-up factor	$\lvert\{\Delta_2 \mid \lvert\Delta_2\rvert \leq 2\}\rvert$	time: $\{\Delta_2\}$	$\lvert\{\Delta_3 \mid \lvert\Delta_3\rvert \leq 3\}\rvert$	time: $\{\Delta_3\}$
	[-]	[-]	[sec]	[-]	[-]	[sec]	[sec]	[-]	[-]	[sec]	[-]	[sec]
`A2woef1`	9	4	0.5	7	2	2.6	1.5	1.8	14	13	47	48
`A4woef1`	11	4	2.9	8	1	311	232	1.3	15	719	20	4.6 k
`A5woef1`	12	4	12	9	1	2.5 k	960	2.6	16	6.1 k	21	>40 k
`A2wsf1`	8	4	0.4	6	2	5.0	1.4	3.5	6	2.3	7	6.7
`A4wsf1`	9	4	2.5	7	2	213	62	3.4	7	70	8	124
`A5wsf1`	9	4	16	7	2	4.9 k	1.1 k	4.7	7	482	8	1.6 k
`A2wsf2`	12	6	0.8	6	2	7.4	1.7	4.3	32	9.9	36	12
`A4wsf2`	20	10	5.6	7	2	444	59	7.6	64	831	72	1.1 k
`A5wsf2`	12	5	46	12	5	8.0 k	1.7 k	4.7	12	1.5 k	12	1.7 k
`A2wst1`	9	4	0.5	8	3	3.6	1.3	2.8	8	1.7	12	3.8
`A4wst1`	11	4	1.5	9	2	93	47	2.0	9	50	10	52
`A5wst1`	12	4	5.0	10	2	1.6 k	336	4.8	10	348	10	365
`A2wst2`	10	5	0.6	8	3	3.4	1.6	2.1	8	2.6	12	5.6
`A4wst2`	12	5	2.0	9	2	115	50	2.3	9	55	10	62
`A5wst2`	13	5	7.0	10	2	449	221	2.0	10	231	10	254
`G5woef1`	15	8	1.4	10	6	3.3	3.0	1.1	14	7.2	27	20
`G20woef1`	15	8	8.2	10	4	24	23	1.0	14	43	42	664
`G100woef1`	15	8	125	10	4	1.2 k	1143	1.1	14	1.9 k	-	>40 k
`G5wsf1`	19	11	3.5	14	7	6.2	6.1	1.0	24	12	24	19
`G20wsf1`	49	26	929	44	22	787	1.0 k	0.8	54	1.2 k	54	1.2 k
`G100wsf1`	-	-	>40 k	-	-	>40 k	>40 k	-	-	>40 k	-	>40 k
`G5wsf2`	7	4	0.5	2	1	1.1	1.1	1.0	27	10	37	64
`G20wsf2`	14	9	2.7	2	1	8.1	9.2	0.9	72	2.9 k	-	>40 k
`G100wsf2`	7	4	116	2	1	404	407	1.0	-	>40 k	-	>40 k
`G5wst1`	7	3	0.4	1	1	1.1	0.7	1.5	10	2.5	19	7.2
`G20wst1`	7	3	2.1	1	1	6.3	5.3	1.2	10	18	10	42
`G100wst1`	7	3	112	1	1	305	302	1.0	10	1.3 k	10	3.0 k
`G5wst2`	9	6	0.8	2	1	1.6	1.3	1.2	8	4.2	8	14
`G20wst2`	9	6	3.1	2	1	8.2	6.9	1.2	8	24	8	108
`G100wst2`	9	6	120	2	1	402	368	1.1	8	1.2 k	8	4.2 k
total	358	177	1.5 k	226	85	22 k	8.0 k	2.7	500	19 k	542	19 k

that the specification becomes realizable. When only guarantees and no signals are considered, even more than half of the core elements can be excluded from being single-fault candidates (Column 2 versus Column 5). For `Gnwsf2`, `Gnwst1`, and `Gnwst2` only the one guarantee (and in two cases one signal of this guarantee) that was injected in order to make the specification unrealizable could be identified by our diagnosis approach. Also for `A4woef1` and `A5woef1`, only one guarantee is reported as modifiable in order to make the specification realizable. Moreover, in many cases, the number of diagnoses does not increase much with increasing bound on the cardinality of the diagnoses (Columns 9 and 11 versus Column 4). Hence, with MBD, the user often obtains very precise information stating where the unrealizability of the specification at hand can be fixed.

Second, MBD is more expensive than the computation of a single unrealizable core, unfortunately. Single-fault diagnosis computation takes about 5 times longer than unrealizable core computation (Column 3 versus Column 7). Without our performance optimizations, the situation is even worse (Column 3 versus Column 6). This is somewhat surprising, since there is only one additional realizability check per unrealizable core element required (for checking whether the core element is a single-fault diagnosis). The explanation for the large difference in the computation time is that most checks during unrealizable core computation are performed on small subsets of the original specification. In contrast, a check whether a given component forms a diagnosis is performed on nearly the entire specification, and is thus also much more expensive. For multiple-fault diagnosis computation, we can observe that the computation time increases noticeably with increasing bound on the cardinality of the diagnoses (Column 7 versus Columns 10 and 12). However, this is not considered as severe problem since the user is typically interested in small diagnoses anyway.

Third, the performance optimizations discussed in Section 4 seem to be effective. Using approximations of realizability, we obtain a speed-up of factor 2.7 when compared to the straightforward implementation described in Section 3.4 (Column 6 versus Column 7, contrasted in Column 8).

6 Conclusion

In this work we showed how model-based diagnosis can be used to locate bugs in unrealizable specifications, although the setting is quite different to the classical model-based diagnosis setting. Our approach computes signals and properties that can be weakened in order to make the specification realizable. It also yields conflicts which can be explained using counterstrategies to gain deeper insight into the problem. Hence, this work complements [17] in a nice way. Although our diagnosis approach is certainly a valueable debugging aid on its own, we believe that the combination with [17], as shown in Fig. 1, provides the user with an even more powerful debugging tool.

Experimental results for GR(1) specifications are promising, especially for single-fault diagnoses. Compared to unrealizable core computation, model-based diagnosis produces 40 % less fault candidates. However, this improvement in the

accuracy comes at the price of a higher computational effort. Hence, model-based diagnosis can be considered as a stronger weapon against unrealizability. It may be an overkill for simpler bugs, where a glimpse on an unrealizable core suffices to fix the problem. However, the higher precision for error localization can be very important for more trickier bugs involving many properties. In order to tackle the performance problem, we showed on the example of GR(1) formulas how realizability and unrealizable cores can be computed faster using approximations. We achieve a speed-up factor of 2.7 for single-fault diagnoses computation. The concept is not specific to GR(1) and can be used for other logics as well.

In the future, we plan to investigate techniques to rule out diagnoses with additional user input, e.g., simulation traces. Furthermore, we plan to experiment with more fine-grained component definitions such as parts of properties.

References

1. R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY - a new requirements analysis tool with synthesis. In *CAV*, pages 425–429. Springer, 2010. LNCS 6174.
2. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *DATE*, pages 1188–1193. ACM, 2007.
3. R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007.
4. H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. A practical approach to coverage in model checking. In *CAV*, pages 66–78. Springer, 2001. LNCS 2102.
5. A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *VMCAI*, pages 52–67. Springer, 2008. LNCS 4905.
6. K. Claessen. A coverage analysis for safety property lists. In *FMCAD*, pages 139–145. IEEE, 2007.
7. L. Console, G. Friedrich, and D. Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *IJCAI*, pages 1494–1499. Morgan-Kaufmann, 1993.
8. 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. IEEE, 2005.
9. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
10. S. Dellacherie. Automatic bus-protocol verification using assertions. In *Global Signal Processing Expo Conference (GSPx)*, 2004.
11. E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *CAV*, pages 263–277. Springer, 2009. LNCS 5643.
12. D. Fisman, O. Kupferman, S. Seinvald, and M. Y. Vardi. A framework for inverent vacuity. In *HVC*. Springer, 2008. LNCS 5394.
13. Y. V. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *DAC*, pages 300–305, 1999.
14. B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, pages 117–124. IEEE, 2006.

15. S. Katz, O. Grumberg, and D. Geist. "Have I written enough properties?" — A method of comparison between specification and implementation. In *CHARME*, pages 280–297. Springer, 1999. LNCS 1703.
16. R. Könighofer. Debugging formal specifications with simplified counterstrategies. Master's thesis, IAIK, Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria, 2009.
17. R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD*, pages 152–159. IEEE, 2009.
18. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
19. O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542. IEEE, 2005.
20. A. Morgenstern and K. Schneider. Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. *CoRR*, abs/1006.1408, 2010.
21. B. Peischl and F. Wotawa. Automated source-level error localization in hardware designs. *IEEE Design and Test of Computers*, 23:8–19, 2006.
22. I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *DAC*, pages 821–826. ACM, 2006.
23. N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380. Springer, 2006. LNCS 3855.
24. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
25. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
26. M. Stumptner and F. Wotawa. Debugging functional programs. In *IJCAI*, pages 1074–1079. Morgan Kaufmann, 1999.
27. F. Wotawa. Debugging VHDL designs using model-based reasoning. *Artificial Intelligence in Engineering*, 14(4):331–351, 2000.
28. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.