

Finding and Fixing Faults

Barbara Jobstmann,

EPFL, Switzerland

Stefan Staber,

OneSpin Solutions, Germany

Andreas Griesmayer,

UNU-IIST, Macao S.A.R. China

Roderick Bloem

Graz University of Technology, Austria

Abstract

Knowing that a program has a bug is good, knowing its location is better, but a fix is best. We present a method to automatically locate and correct faults in a finite state system, either at the gate level or at the source level. We assume that the specification is given in Linear Temporal Logic, and state the correction problem as a game, in which the protagonist selects a faulty component and suggests alternative behavior. The basic approach is complete but as complex as synthesis. It also suffers from problems of readability: the correction may add state and logic to the system. We present two heuristics. The first avoids the doubly exponential blowup associated with synthesis by using nondeterministic automata. The second heuristic finds a memoryless strategy, which we show is an NP-complete problem. A memoryless strategy corresponds to a simple, local correction that does not add any state. The drawback of the two heuristics is that they are not complete unless the specification is an invariant. Our approach is general: the user can define what constitutes a component, and the suggested correction can be an arbitrary combinational function of the current state and the inputs. We show experimental results supporting the applicability of our approach.

Key words: Debugging; Fault localization; Fault correction; Verification; Games; Controller synthesis; Linear temporal logic; Memoryless strategies; Repair

* This work was supported in part by the European Union under contract 507219 (PROSYD) and performed while the authors were at Graz University of Technology.

1 Introduction

1.1 Motivation

Industrial data suggests that up to two thirds of design time is spent in verification-related activities, which includes detecting, localizing, and correcting faults. Of this time, detection of the bugs takes only about one third, the other two thirds, or almost half the total design time, being spent locating and correcting faults.

Nevertheless, there is little tool support for localization and correction. Formal verification tools typically return a counterexample when verification fails, but even if such a trace is available, it may be hard work to find the fault contained in the system. Dynamic verification techniques typically only return the state of the system when a failure occurs, which is of even less use.

The goal of this paper is to present a formal, automatic method to locate and correct faults in finite state systems both on the source-code level and on the gate level.

1.2 Summary of the Approach

In this paper, we take the view that a component may be responsible for a fault if and only if it can be replaced by an alternative that makes the system correct. Thus, fault localization and correction are closely connected, and our approach combines the two. We assume a finite-state sequential system, which can be hardware or finite-state software. We furthermore assume that a (partial) specification is given in Linear Temporal Logic (LTL), and we endeavor to find and fix a fault in such a way that the new system satisfies its specification for all possible inputs. Our fault model is quite general: we assume that any component can be replaced by an arbitrary function in terms of the inputs and the state of the system.

Our approach decides which component is incorrect and how to replace it, independent of the choice what a component is. We consider the independence of the definition of components to be an important strength of the approach as the proper concept of a component depends on the application. For circuits, components will typically be gates or small blocks such as adders. For systems specified on the source code level (in languages such as VHDL and Verilog), it is natural to consider expressions as the unit of granularity. (Although we can also include, for example, the left-hand sides of assignments.) Thus, for finite-state programs both fault localization and correction are performed

at the expression level, even though an expression may correspond to multiple gates on the implementation level.

The correction problem is closely related to the synthesis problem [1–4]. In order to automatically synthesize a system, a complete specification is needed, which is a heavy burden on the user [5,6]. For the correction problem, on the other hand, we only need as much of the specification as is necessary to find a valid correction. This can be likened to model checking, for which we do not need a full specification either. (This has the obvious drawback that an automatic correction may violate an unstated property and needs to be reviewed by a designer.) Furthermore, the modification is limited to a small portion of the program. The structure and logic of the program are left untouched, which makes it amenable to further modification by the user. Automatically synthesized programs may be hard to understand.

We consider the fault localization and correction problem as an infinite game. The game is played between the environment (the antagonist) and the system (the protagonist). The environment provides the inputs and the system makes two decisions:

- (1) Initially, the system decides which component is incorrect, and
- (2) at every clock cycle the system decides the output of the faulty component.

The winning condition for the protagonist is the satisfaction of the specification. The game is won if the system can choose outputs for the component in such a way that the system satisfies the specification regardless of the inputs. After describing related work in Section 2 and introducing the necessary definitions and notation in Section 3, we discuss this construction in detail in Section 4.

One can find the proper choices for the system by computing a strategy on the product of the model and a deterministic automaton for the formula. This yields a sound and complete approach to localization and correction. This approach, however, has its drawbacks: the size of the deterministic automaton is doubly exponential in the size of the formula, computing the strategy is computationally complex, and the construction is very difficult to implement. In order to avoid these drawbacks, we construct a Büchi game that is the product of the program game and the standard nondeterministic automaton for the specification. If the product game is won, so is the program game, but because of the nondeterminism in the automaton, the converse does not hold. In many cases, however, we can find a winning finite state strategy anyway. (See Section 5.1.)

To implement a correction corresponding to a finite state strategy, we may need to add state to the program, mirroring the specification automaton. Such

a correction may significantly alter the program by inserting new variables and new assignments throughout the code, which is unlikely to please the developer. Instead, we look for a memoryless strategy, which corresponds to a correction that changes only the faulty part of the system and does not introduce new state. In Section 5.2 we show that deciding whether such a strategy exists is NP-complete, so in Section 5.3 we develop a heuristic to find one.

We obtain a conservative algorithm that yields valid corrections and is complete for invariants. It may, however, fail to find a memoryless correction for other types of properties, either because of nondeterminism in the automaton or because of the heuristic that constructs a memoryless strategy.

In Section 6.1 we describe a symbolic method to extract a correction from the strategy. We prove the correctness of the approach in Section 6.2 and show that its complexity is comparable to that of model checking in section 6.3.

We have implemented the algorithm in VIS and we present our experience with the algorithm in Section 7.

This article is based on [7], [8], and [9].

2 Related Work

2.1 *Understanding Failure*

Researchers have taken different approaches to make debugging easier. One approach is to make the “failure-inducing input” easier to understand. In the setting of model checking, this has been a concern all along. For instance, [10] and [11] consider generating short counterexamples as these are likely to be easier to read. More recently, [12] proposes a method to remove irrelevant variables from a counterexample derived using bounded model checking. Similarly, in the setting of software testing, Zeller and Hildebrandt [13] consider the problem of simplifying the input that causes the failure.

The authors of [14] show how to help the user understand the counterexample. The user partitions the inputs in to signals controlled by the system and those controlled by the environment. Using a formulation in terms of games, the counterexample is then partitioned into “fated” and “free will” segments. The fault can not be avoided by changing the system variables in the fated segments, but the free-will segments offer a possibility to escape the fault by choosing different values for the system variables. Thus, the segmentation

highlights important events. This paper borrows from [14] both in terms of the conceptual framework and in terms of implementation. In particular, by giving the system control of internal signals instead of inputs, we can derive a repair instead of a partition of the counterexample.

A further approach to help the user understand a failure (which is not necessarily the same as locating the fault) is to consider several similar traces, some of which show failure and some success [15–19]. The similarities between failure traces and their differences with the successful traces give an indication of the parts of the program that are likely to be involved in the failure. The selection of the traces is crucial and the methods to select them range from user provided traces to automated computation of similar runs. In [19], Groce et al. show how to generate a successful trace that is close to a given counterexample with respect to a distance metric. They use a SAT based model checker, which generates a propositional formula whose satisfying assignments represent a counterexample. With that formula and the specification, they build an optimization problem to find the closest successful run. Like other trace-based methods, this method is neither sound nor complete: Since a fault may not be observable for some inputs even if the faulty line is executed, the fault may occur in some successful traces. Vice-versa, there may be correct lines that occur only in unsuccessful traces. Note that in any case, the granularity of these methods consists of basic blocks. On the other hand, experimental results show that trace based methods perform quite well in practice. It would be interesting to see whether these methods could be used as efficient preprocessing steps to narrow down the choice of fault locations.

2.2 Correction

There has been considerable research in correction of combinational circuits. Such approaches are typically also applicable to sequential circuits, as long as the state space is not re-encoded. Usually, a correct version of the circuit is assumed to be available as a specification. For instance, optimization may have introduced a bug, and the unoptimized version can be used as a reference model. The papers [20] and [21] discuss formal methods of fault localization and correction based on Boolean equations. The fault model of [20] is the same one we use for sequential circuits: any gate can be replaced by an arbitrary function. Chung, Wang, and Hajj [22] improve these methods by pruning the set of possible faults. They consider only a set of “simple,” frequently occurring design faults. In [23] an approach is presented that may fix multiple faults of limited type by generating special patterns.

Work on sequential fault localization and correction is more sparse. In the sequential setting, we assume that it is not known whether the state is correct

at every clock tick, either because the reference model has a different encoding of the state space, or because the specification is given in a logic rather than as a circuit. Wahba and Borrione [24] discuss a method of finding single errors of limited type (forgotten or extraneous inverter, and/or gate switched, etc.) in a sequential circuit. The specification is assumed to be another sequential circuit, but their approach would presumably also work if the specification is given in a temporal logic. The algorithm finds the fault using a given set of test patterns. It iterates over the time frames, in each step removing from suspicion those gates that would, if changed, cause an output to become or remain incorrect. Our work improves that of Wahba and Borrione in two respects: we use a more general fault model, and we correct the circuit for any possible input, not just for a given test sequence. Both improvements are important in a setting where a specification is available rather than a reference model. As far as we are aware, there are currently no complete approaches to correct a broken system with a fault model of comparable generality.

Buccafurri et al. [25] consider the correction problem for CTL as an abductive reasoning problem and present an approach that is based on calling the model checker once for every possible correction to see if it is successful. Our approach needs to consider the problem only once, considering all possible corrections at the same time, and is likely to be more efficient.

Ebnanasir et al. [26] research the problem of correcting faulty programs written in the programming language Unity. Typical programs in Unity are highly nondeterministic. Repairs of such programs are constructed by removing edges from the transition structure that corresponds to the program and are manually mapped back to the source code. The aims of this work are very similar to ours. In particular, the authors have independently proven that deciding whether a memoryless strategies exists is NP-complete.

Chang et al. [27] present a method to repair faulty circuits using a set of simulation traces, under the assumption that the correct values of the state bits are known. Their effort focuses on reuse of existing logic to synthesize a repair, something that we do not address. Their approach does not construct the repair from a specification and may thus suggest incorrect fixes, which is counteracted by iterating the repair process.

Janjua and Mycroft [28] describe how to automatically insert synchronization statements in a multithreaded program in order to prevent bugs due to an unfortunate scheduling. In [29], a C-like language is presented in which programs can be *sketched*: unknown constants can be represented by `??`. A synthesizer then completes the sketch to adhere to a specification.

2.3 Model Based Diagnosis

We will spend some more space contrasting our approach with model based diagnosis, as it is one of the few systematic approaches to fault localization.

Model based diagnosis originates with the localization of faults in physical systems. Console et al. [30] show its applicability to fault localization in logic programs. In model based diagnosis, a correct model is not assumed to exist. Rather, an oracle provides an example of correct behavior that is inconsistent with the behavior of the program. The reasoning is performed on the faulty program. This approach has been extended to functional programs [31], hardware description languages [32], and object oriented programs [33].

Model based diagnosis comes in two flavors: abduction-based and consistency-based diagnosis [34]. Abduction-based diagnosis [35] assumes that the set of fault models is enumerated, i.e., it is known in which ways a component can fail. Using these fault models, it tries to find a component of the model and a corresponding fault that explains the observation. The approach of [24] can be seen as an abductive approach: it works with a small set of given fault models and combines localization with correction. The set of fault models that we consider in this work, however, is equal to the number of replacement functions in terms of the inputs and the state variables of the system, which is doubly exponential. Thus, we do not consider it wise to enumerate all possible fault models and our approach should not be considered abductive.

Consistency-based diagnosis [36,37] does not require the possible faults to be known and does not produce a correction. Rather, it considers the faulty behavior as a contradiction between the actual and the expected behavior of the system. It proceeds by dropping the assumptions on the behavior of each component in turn. If this removes the contradiction, the component is considered a candidate for correction. In this setting, each component is described as a set of constraints. For example, an AND gate x with inputs i_1 and i_2 is described as

$$\neg\text{faulty}_x \Rightarrow (\text{out}_x \Leftrightarrow i_1 \wedge i_2),$$

where faulty_x means that x is considered responsible for the failure. Note that nothing is stated about the behavior of the gate when faulty_x is asserted. The task of consistency-based diagnosis is to find a minimal set Δ of components such that the assumption $\{\text{faulty}_c \mid c \in \Delta\} \cup \{\neg\text{faulty}_c \mid c \notin \Delta\}$ is consistent with the expected behavior.

In diagnosis of sequential circuits, an incorrect trace is given and diagnosis is performed using the unrolling of the circuit as the model. A single predicate is used to indicate the malfunctioning of all occurrences of a given component

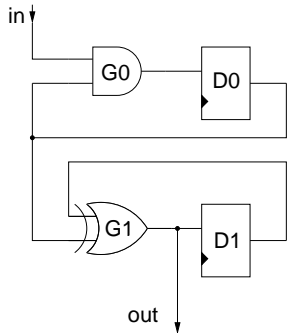


Fig. 1. Simple circuit. The initial state is $D0=0$, $D1=0$.

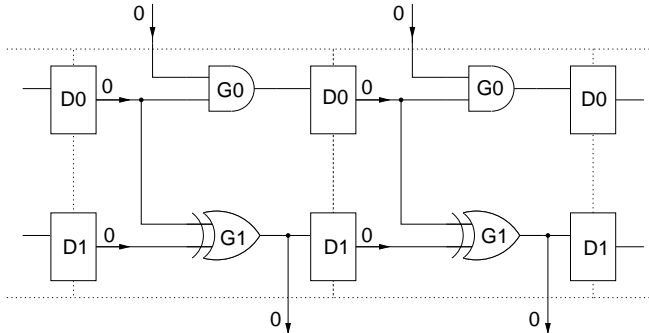


Fig. 2. Unrolling of circuit in Fig. 1

[38]. (Although alternatives are possible [39].) Fahim Ali et al. [40], for example, present a SAT-based method for consistency-based diagnosis that is based on this principle. Instead of using “faulty” predicates, they use multiplexers with one free input.

Consistency-based reasoning has weaknesses when multiple instances of a component appear: components may be reported as diagnoses, although no consistent repair exists. (A similar observation is made in [41] for multiple test cases.) Hamscher and Davis [42] show that consistency-based diagnosis is indiscriminate in the sequential case. For instance, if dropping the constraints of a component removes any dependency between input and output, that component is a diagnosis. In sequential circuits, because of the replication of components, this may hold for many components.

An example of a diagnosis without a consistent repair is given by the sequential circuit shown in Fig. 1. Suppose the initial state of the circuit is $(0, 0)$ and the specification is $(out = 0) \wedge G((out = 0) \leftrightarrow X(out = 1))$. Fig. 2 shows the unrolling of the circuit corresponding to a counterexample of length 2. Consider the XOR gate. Any output is possible if the constraints for this gate are removed, so it is a diagnosis. The AND gate is also a diagnosis. The conclusion that either gate can be the cause of the failure, however, is incorrect. There is no replacement for the XOR gate that corrects the circuit: for the given input, the output of the XOR gate needs to be 0 in the first and 1 in the second time frame. This is impossible because the inputs to the gate are necessarily 0 in both time frames. The circuit can be corrected, but the only way to fix the circuit for the given input sequence is to replace the AND gate by any gate whose output is 1 when both inputs are 0.

In diagnosis of physical systems, faults may be intermittent, and a consistent explanation of the faulty behavior may not be required. In the setting of correction, however, the replacement must be consistent and functional [38]. Thus, repairability is the proper notion for fault localization, and for maximum precision, the combination of fault localization and correction is essential.

Model based diagnosis gives a general, formal methodology of fault localization, but its two flavors each have shortcomings. First, the abduction-based approach can only handle a small set of possible faults. Second, the consistency-based method does not suggest a correction and, moreover, is unable to differentiate between correctable and non-correctable diagnoses. Furthermore, model based diagnosis does not deal with the problem of correcting a system for any possible input, but only finds a correction that is valid for a fixed set of inputs. Our approach is precise and finds corrections that are valid for all inputs.

3 Preliminaries

In this section, we describe the necessary theoretical background for our work.

3.1 LTL

Linear Temporal Logic (LTL) [43] is a temporal logic built from the atomic propositions in AP , their negations, the temporal modalities X (next), U (until), and R (releases), and Boolean conjunction and disjunction. An LTL formula defines a set of words. Intuitively, $X\varphi$ holds for a given word if φ holds in the suffix of the word starting from the second position, $\varphi U\psi$ holds if ψ holds on some position and φ holds up until that point, and $\varphi R\psi$ holds if either ψ holds on all positions, or $\varphi \wedge \psi$ holds on some position and ψ hold in all positions up to that one. As usual, we define $F\varphi = \text{true} U \varphi$ and $G\psi = \text{false} R \psi$. Note that our LTL formulas are defined in negation normal form. We define $L(\varphi) = \{w \in (2^{AP})^\omega \mid w \models \varphi\}$. For an intuitive explanation and a formal definition of LTL, see [44].

3.2 Finite State Machines and Circuits

A *finite state machine (FSM)* over a finite alphabet AP is a tuple $M = (S, s_0, I, \delta, \lambda)$, where S is a finite set of states, $s_0 \in S$ is the initial state, I is a finite set of inputs, $\delta : S \times I \rightarrow S$ is the transition function, and $\lambda : S \rightarrow 2^{AP}$ is the labeling function.

A *circuit* consists of inputs, outputs, flip-flops, gates, and wires that connect them. The formalization of circuits is straightforward, as is their connection to FSMs. A circuit is *combinational* if it does not contain any latches.

Suppose we are given a circuit A and a gate d of A that we suspect to be incorrect. We can remove d (and any logic that drives only d) to obtain a new circuit in which the output of d has become a new input. Repairing A then reduces to finding a controller for the new input. Let B be a circuit with as inputs the inputs of A plus the flip-flops of A and a single Boolean output. The *substitution of B for d* is the circuit obtained by merging A and B and replacing d with the output of B . The substitution is *combinational* if B is combinational. The *(combinational) LTL repair problem* is that of finding, given A and a specification φ (that A does not necessarily fulfill), a gate d of A and a (combinational) circuit B with one output such that the substitution of B for d fulfills φ . The *(combinational) LTL repairability problem* is the corresponding decision problem.

3.3 Games

A *game G* over AP is a tuple $(S, s_0, I, C, \delta, \lambda, F)$, where S , s_0 , and λ are the same as for FSMs, I and C are finite sets of environment inputs and system choices, $\delta : S \times I \times C \rightarrow S$ is the partial transition function, and $F \subseteq S^\omega$ is the winning condition, a set of infinite sequences of states. With the exception of this section and Section 5.3, we will assume that δ is a complete function.

In the terminology of control theory of discrete event systems [45], our game is a plant with a specification. The environment actions are given by the set I and the plant can be controlled by a set of actions C . The challenge is to find a controller that delivers the proper system choices, such that the combination of controller and plant satisfies the specification F . The game-theoretic equivalent of a controller is a strategy.

Given a game $G = (S, s_0, I, C, \delta, \lambda, F)$, a *(finite state) strategy* is a tuple $\sigma = (V, v_0, \mu)$, where V is a finite set of states, $v_0 \in V$ is the initial state, and $\mu : S \times V \times I \rightarrow 2^{C \times V}$ is the *move function*. Intuitively, a strategy is an automaton that determines a set of possible responses to an environment input, and its response may depend on the current state of the game and a finite memory of the past. Note that strategies are nondeterministic. We need nondeterminism in the following in order to have maximal freedom when we attempt to convert a finite state strategy to a memoryless strategy. (See below.)

A *play* on G according to σ is a finite or infinite sequence $\pi = s_0 v_0 \xrightarrow{i_0 c_0} s_1 v_1 \xrightarrow{i_1 c_1} \dots$ such that $(c_j, v_{j+1}) \in \mu(s_j, v_j, i_j)$, $s_{j+1} = \delta(s_j, i_j, c_j)$, and either

- (1) the play is infinite, or
- (2) there is an n such that $\mu(s_n, v_n, i_n) = \emptyset$ or $\delta(s_n, i_n, c_n)$ is not defined,

which means that the play is finite.

A play is *winning* if it is infinite and $s_0s_1\cdots \in F$. (If $\mu(s_n, v_n, i_n) = \emptyset$, the strategy does not suggest a proper system choice and the game is lost.) A strategy σ is *winning* on G if all plays according to σ on G are winning (for all possible nondeterministic choices that a strategy can make). Returning to the control theory terminology, the strategy describes a (nondeterministic) controller, which should be connected to the plant (our game). If the strategy is winning, the combination of plant and controller is guaranteed to satisfy the specification, regardless of the nondeterministic choices of the controller.

A *memoryless strategy* is a finite state strategy (V, v_0, μ) with $V = \{v_0\}$. Since in this case the state set and the initial state are uninteresting, we can identify the strategy with its move function μ , and further simplify it by leaving out the single state. Thus, we will write a memoryless strategy as a function $\sigma : S \times I \rightarrow 2^C$. Likewise, we write a play according to a memoryless strategy as a sequence $s_0 \xrightarrow{i_0c_0} s_1 \xrightarrow{i_1c_1} \dots$, leaving out the state of the strategy automaton.

We extend the labeling function λ to plays. Given a play $\pi = s_0v_0 \xrightarrow{i_0c_0} s_1v_1 \xrightarrow{i_1c_1} \dots$, the *output word* is $\lambda(\pi) = \lambda(s_0)\lambda(s_1)\dots$. Likewise, the *input word* is $i(\pi) = i_0i_1\dots$, the sequence of system inputs. The *output language* (*input language*) $L(G)$ ($I(G)$) of a game is the set of all $\lambda(\pi)$ ($i(\pi)$) with π winning.

Games can be classified according to their winning condition. A *safety game* has the condition $F = \{s_0s_1\cdots \mid \forall i : s_i \in A\}$ for some $A \subseteq S$. The winning condition of an *LTL game* is the set of sequences satisfying an LTL formula φ . In this case, we will write φ for F . *Büchi games* are defined by a set $B \subseteq Q$, and require that a play visits the Büchi constraint B infinitely often. For such games, we will write B for F .

We can convert an LTL formula φ over the set of atomic propositions AP to a Büchi game $A = (Q, q_0, 2^{AP}, C, \delta, \lambda, B)$ such that $I(A)$ is the set of words satisfying φ [46,47]. The system choice C models the nondeterminism of the automaton. The size of the resulting automaton is exponential in the length of the formula in the worst case. (See [44] for an introduction.)

3.4 Solving Games

We will now introduce some notation for solving games. We assume a basic understanding of the μ -calculus. (See [44].) For a set $A \subseteq S$, the set

$$\text{MX } A = \{s \mid \forall i \in I \exists c \in C \exists s' \in A : (s, i, c, s') \in \delta\}$$

is the set of states from which the system can force a visit to a state in A in one step. The set $MA \cup B$ is defined by the μ -calculus formula $\mu Y. B \cup A \cap MX(Y)$. It defines the set of states from which the system can force a visit to B without leaving A . The *iterations* of this computation are

$$Y_0 = B \text{ and} \\ Y_{j+1} = Y_j \cup (A \cap MX Y_j) \text{ for } j \geq 0.$$

From Y_j the system can force a visit to B in at most j steps. Note that there are only finitely many distinct iterations because the set of states is finite.

We define $MG A = \nu Z. A \cap MX Z$, the set of states from which the system can avoid leaving A . Note that these fixpoints are similar to the ones used in model checking of fair CTL and are easily implemented symbolically. The difference is the use of MX instead of EX .

Using these fixpoint formulas, we can compute memoryless strategies for safety and Büchi games [48]. For a safety game with condition A , the strategy

$$\sigma(s, i) = \{c \in C \mid \exists s' \in MG A : (s, i, c, s') \in \delta\}$$

is winning if and only if $s_0 \in MG A$. For a Büchi game, we define $W = \nu Z. MX MZ \cup (Z \cap B)$. The set W is the set of states from which the system can win the Büchi game. Let Y_0 through Y_n be the set of distinct iterations of the fixpoint $MW \cup (W \cap B) = W$. We define the *attractor strategy* for B to be

$$\sigma(s, i) = \{c \in C \mid \exists j, k < j, s' \in Y_k : s \in Y_j \setminus Y_{j-1}, (s, i, c, s') \in \delta\} \cup \\ \{c \in C \mid s \in Y_0, \exists s' \in W : (s, i, c, s') \in \delta\}.$$

The attractor strategy brings the system ever closer to B , until it visits B , and then brings it back to a state from which it can force another visit to B .

4 Localization and Correction as a Game

We show how to search for a faulty component and a correct replacement by means of sequential circuits, where the specification is an LTL formula φ . A correction is a replacement of one gate by an arbitrary Boolean function in terms of the primary inputs and the current state. Focusing on sequential Boolean circuits allows us to give a simpler exposition, while retaining full generality. We will return to source-level debugging in Section 7.

Suppose we are given a circuit, a corresponding FSM $M = (S, s_0, I, \delta, \lambda)$, and an LTL specification φ . We extend M to a game between the system and the

environment. Suppose the gates in the circuit are numbered by $0 \dots n$. We extend the FSM to a game by the following two steps

- (1) We extend the state space to $(S \times \{0 \dots n\}) \cup s'_0$. Intuitively, if the system is in state (s, d) , we suspect gate d to be incorrect; s'_0 is a new initial state. From this state, the system can choose which gate is considered faulty.
- (2) We extend the transition function to reflect that the system can choose the output of the suspect gate.

In Step 2, supposing that gate d is suspect, we remove it from the combinational logic and obtain a new combinational logic with one more input. Let the function computed by this new circuit be given by $\delta_d : S \times I \times \{0, 1\} \rightarrow S$, where the third argument represents the new input.

We construct the LTL game $G = (S', s'_0, I, C', \delta', \lambda', \varphi)$, where

$$\begin{aligned}
S' &= (S \times \{0, \dots, n\}) \cup s'_0, \\
C' &= \{0, \dots, n\}, \\
\delta'(s'_0, i, c) &= (s_0, c), \\
\delta'((s, d), i, c) &= (\delta_d(s, i, c \bmod 2), d), \\
\lambda(s'_0) &= \text{arbitrary}, \\
\lambda'((s, d)) &= \lambda(s), \text{ and} \\
\varphi' &= \mathbf{X} \varphi.
\end{aligned}$$

We will call this game the *program game*, reflecting that in general, the system to be corrected is a finite-state program. Note that the full range $\{0, \dots, n\}$ of the system choice is only used in the new initial state s'_0 to choose the suspect gate. Afterwards (for the Boolean case) we only need two values to decide the correct output of the gate, so we use the modulo operator. Also note that the decision which gate is suspect does not depend on the inputs: $\delta'(s'_0, i, c)$ does not depend on i . Finally, note that the specification does not consider the first state and that hence, the label of s'_0 is immaterial.

We will now give an example. Consider the circuit in Fig. 1. The corresponding FSM is shown in Fig. 3. Suppose we want the output to alternate between 0 and 1. Stating this specification in terms of state variables (noting that D1 contains the value of the output in the previous cycle), we obtain

$$\mathbf{X}(D1 = 0 \wedge \mathbf{G}((D1 = 0) \leftrightarrow \mathbf{X}(D1 = 1))).$$

From the finite state machine, we obtain the game shown in Fig. 5. In the initial state the system chooses which gate (G0 or G1) is faulty. The upper part of the game in Fig. 5 corresponds to an arbitrary function for gate G0, the lower one represents a replacement of gate G1.

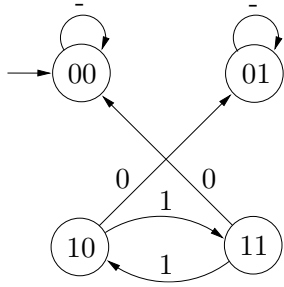


Fig. 3. Faulty system corresponding to Fig. 1. The labels on the states give the value of D0 and D1, the edges are labeled with the input values.

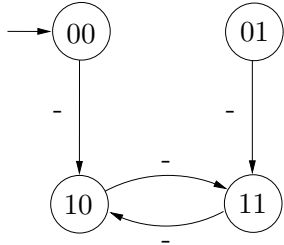


Fig. 4. Corrected system.

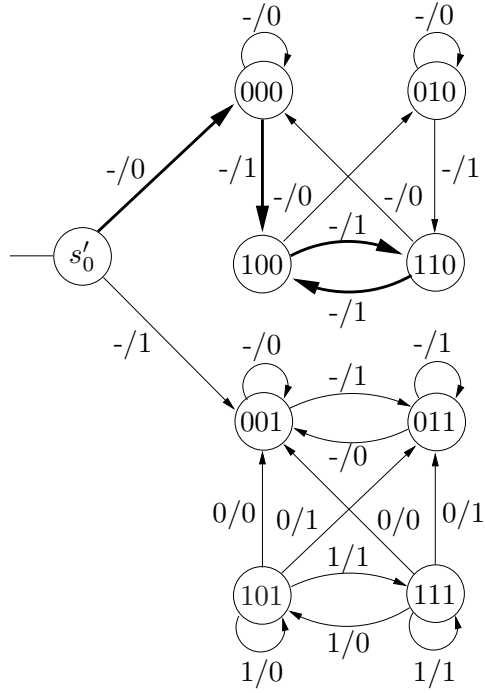


Fig. 5. Game to localize and correct the fault. States are labeled with D0, D1, and the suspect component. Edges are labeled with the input, a slash, and the system choice.

Once we have constructed the game, we select system choices that restrict the game to those paths that fulfill the specification. In our example, first we choose a transition from s'_0 to either the upper or the lower part of the game. Suppose we choose the transition from s'_0 to 000: we try to fix the fault by replacing gate G0. In state 000 we select transitions that lead to a path adhering to the given specification. In Fig. 5 the bold arrows only allow paths with the sequence 001010... for D1 as required by the specification. Taking only these transitions into account we get the function shown in Table 1 for the system choice c . For the 3rd and 4th Line in Table 1 we can choose arbitrary values for the system choice, which gives us freedom in picking the desired correction. Since we aim for corrections that yield simple modified systems, we choose the simplest implementation, which sets $c = 1$ all the time. This corresponds to a circuit in which G0 is the constant 1. The corresponding finite state machine is shown in Fig. 4.

The extensions necessary to handle multiple faults are quite simple. First, instead of picking one faulty component, we pick n . Then we need to select a proper value for each of the faulty components for each combination of state and input.

Table 1
Function for the system choice

State			Input	Choice
$S \times \{0, 1\}$			I	C'
D_0	D_1	d	i	c
0	0	0	0	1
0	0	0	1	1
0	1	0	0	–
0	1	0	1	–
1	0	0	0	1
1	0	0	1	1
1	1	0	0	1
1	1	0	1	1

5 Solving the Game

In this section, we show how to compute a winning strategy and how to extract a correction from it. In Section 5.1 we show how to construct a finite state winning strategy for the program game by computing a strategy on the product of the program game and an automaton representing the specification. Because we use a nondeterministic automaton, this is a heuristic. We also discuss under which conditions we can guarantee that we can find such a strategy.

A finite state strategy for the program game corresponds to a correction that adds states to the program. Since we want a correction that is as close as possible to the original program, we search for a memoryless strategy. In Section 5.2, we show that it is NP-complete to decide whether a memoryless strategy exists, and in Section 5.3, we present a heuristic to construct a memoryless strategy. This heuristic may fail to find a valid memoryless strategy even if one exists.

5.1 Finite State Strategies

Given two games $G = (S, s_0, I_G, C_G, \delta_G, \lambda_G, F_G)$ and $A = (Q, q_0, 2^{AP}, C_A, \delta_A, \lambda_A, F_A)$, let the *product game* be

$$G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, F), \text{ where}$$

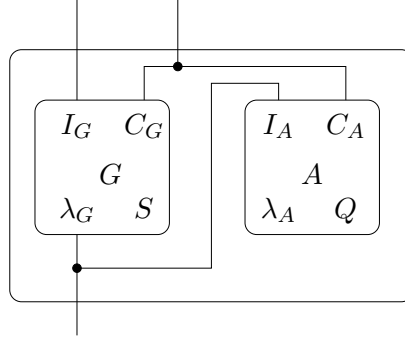


Fig. 6. Graphical representation of the product game $G \triangleright A$.

$$\begin{aligned} \delta((s, q), i_G, (c_G, c_A)) &= (\delta_G(s, i_G, c_G), \delta_A(q, \lambda_G(s), c_A)), \\ \lambda(s, q) &= \lambda_G(s), \text{ and} \\ F &= \{(s_0, q_0), (s_1, q_1), \dots \mid s_0, s_1, \dots \in F_G \wedge q_0, q_1, \dots \in F_A\}. \end{aligned}$$

Game A is the *specification automaton*. Intuitively, the output of G is fed to the input of A , and the winning conditions are conjoined. Therefore, the output language of the product is the intersection of the output language of the first game and the input language of the second. (See Fig. 6.)

Lemma 1 For games G, A , $L(G \triangleright A) = L(G) \cap I(A)$.

Lemma 2 Let G and A be games. If a finite state winning strategy σ for $G \triangleright A$ exists, then there is a finite state winning strategy σ' for G such that for all plays π of G according to σ' , $\lambda(\pi) \in L(G)$ and $\lambda(\pi) \in I(A)$.

PROOF. Let $\sigma = (V, v_0, \mu)$ be a winning finite state strategy for $G \triangleright A$. Note that $\mu : (S \times Q) \times V \times I_G \rightarrow 2^{(C_G \times C_A) \times V}$. Let $\sigma' = (Q \times V, (q_0, v_0), \mu')$ with $\mu' : S \times (Q \times V) \times I_G \rightarrow 2^{C_G \times (Q \times V)}$ such that

$$\begin{aligned} \mu'(s, (q, v), i_G) &= \{(c_G, (q', v')) \mid \\ &\exists c_A : ((c_G, c_A), v') \in \mu((s, q), v, i_G) \wedge q' = \delta_A(q, \lambda_G(s), c_A)\}. \end{aligned}$$

Let $\pi' = s_0(q_0, v_0) \xrightarrow{i_{G_0} c_{G_0}} s_1(q_1, v_1) \dots$ be a play of G according to σ' . Then there are c_{A_i} such that $\pi = (s_0, q_0)v_0 \xrightarrow{i_{G_0}(c_{G_0}, c_{A_0})} (s_1, q_1)v_1 \dots$ is a play of $G \triangleright A$ according to σ . Because σ is winning, π is winning and $\lambda(\pi) \in L(G) \cap I(A)$. Since $\lambda(\pi') = \lambda(\pi)$ it follows that $\lambda(\pi') \in L(G) \cap I(A)$, and because π' is arbitrary, this implies that σ' is winning. \square

The finite state strategy σ' is the product of the specification automaton A and the finite state strategy σ for $G \triangleright A$. If $F_G = S^\omega$, then σ' is the winning strategy

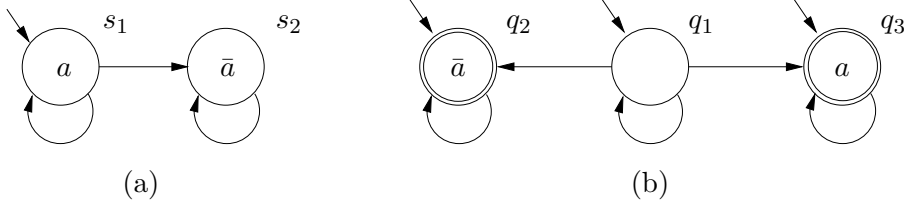


Fig. 7. (a) Game G in which the environment can choose a value for variable a . (b) automaton A for $\text{FG}(a) \vee \text{FG}(\bar{a})$.

for the game G with the winning condition defined by A . The following result (an example of *game simulation*, cf. [48]) follows from Lemma 2.

Theorem 3 *Let $G = (S, s_0, I, C, \delta, \lambda, \varphi)$ be an LTL game, let G' be as G but with the winning condition S^ω , and let A be a Büchi game with $I(A) = L(\varphi)$. If there is a winning strategy for the Büchi game $G' \triangleright A$ then there is a finite state winning strategy for G .*

Note that the converse of the theorem does not hold because A may be non-deterministic.

For example, there is no winning strategy for the game $G \triangleright A$ shown in Fig. 7. If the system decides to move to the state q_3 in the automaton, the environment can decide to move to s_2 (set $a = 0$), a move that the system cannot match. If, on the other hand, the automaton waits for the environment to move to s_2 , the environment can stay in s_1 forever and thus force a non-accepting run. Hence, although the game fulfills the specification expressed by the automaton, there is no winning strategy. Note that the inability to provide a winning strategy depends not only on the structure of the specification automaton, but also on the structure of the game. For instance, if we remove the edge from s_1 to s_2 , limiting the choices of the environment, a strategy for the product exists.

We can always find a strategy if the specification automaton is deterministic. However, the translation of an LTL formula to a deterministic Büchi automaton, if possible, requires a doubly exponential blowup and the best known upper bound for deciding whether a translation is possible is EXPSpace [49]. To prevent this blowup, we can either use heuristics to reduce the number of nondeterministic states in the automaton [50], or we can use a restricted subset of LTL. Maidl [51] shows that translations in the style of [52] (of which we use a variant, [53]) yield deterministic automata for the formulas in the set LTL^{det} , which is defined as follows: If φ_1 and φ_2 are LTL^{det} formulas, and p is a predicate, then p , $\varphi_1 \wedge \varphi_2$, $\text{X} \varphi_1$, $(p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2)$, $(p \wedge \varphi_1) \text{U} (\neg p \wedge \varphi_2)$ and $(p \wedge \varphi_1) \text{W} (\neg p \wedge \varphi_2)$ are LTL^{det} formulas. Note that this set includes invariants ($\text{G} p$) and the formula $\neg p \text{U} p$, which is equivalent to $\text{F} p$. LTL^{det} describes the intersection of LTL and CTL. In fact, deterministic Büchi automata describe exactly the LTL properties expressible in the alternation-free μ -calculus, a superset of CTL [49].

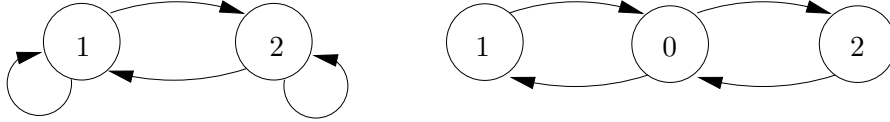


Fig. 8. Games with and without memoryless strategy. The system can choose between multiple outgoing edges and aims to visit state 1 and 2 infinitely often.

Alur and La Torre [54] define a set of LTL fragments for which we can compute deterministic automata using a different tableau construction. They are classified by means of the operators used in their subformulas. (On the top level, negation and other Boolean connectives are always allowed.) Alur and La Torre give appropriate constructions for the classes $LTL(F, \wedge)$ and $LTL(F, X, \wedge)$. In contrast, for $LTL(F, \vee, \wedge)$ and $LTL(G, F)$ they show that the size of a corresponding deterministic automaton is necessarily doubly exponential in the size of the formula.

5.2 Memoryless Strategies are NP-Complete

The finite state strategy corresponding to the product game defined in the last section may be quite awkward to implement as it requires the program to keep track of the state of the specification automaton. This means that we have to add extra state, which (in the case of programs) we have to update whenever a variable changes that is referenced in the specification. Instead, we wish to construct a memoryless strategy. Such a strategy corresponds to a correction that does not require additional state.

Note that not every winning game has a memoryless strategy. Fig. 8 shows two games with the winning condition that state 1 and 2 are both visited infinitely often. From each state, the system can choose an outgoing edge. Both games are winning for the system, but only the left game has a memoryless winning strategy. In the game on the right, a winning strategy has to alternate between the edge from state 0 to state 1 and the edge from state 0 to state 2.

In this section we prove that deciding whether a game with a generalized Büchi acceptance condition has a memoryless strategy is NP-complete. Thus, there is no algorithm to decide whether an LTL game has a memoryless strategy that is polynomial in the size of the game graph, unless $P = NP$.

A directed graph G is *homeomorphic* to a directed *pattern graph* P under a mapping f from the nodes of P to the nodes of G , if there is a mapping from the edges of P to node-disjoint paths in G such that an edge in P from v to w is mapped to a simple path in G from $f(v)$ to $f(w)$.

The *subgraph homeomorphism problem* is to decide, given G , P , and f , whether

G is homeomorphic to P . Fortune, Hopcroft, and Wyllie [55] study the *fixed pattern graph homeomorphism problem*, in which P is fixed. They partition the set of pattern graphs in two subsets. First, if all edges in the pattern graph are outgoing from or ingoing to a designated node, the fixed pattern graph homeomorphism problem is solvable in polynomial time. Second, for all other pattern graphs, the fixed pattern graph homeomorphism problem is NP-complete. It follows that deciding whether there is a simple cycle between two nodes in a given graph is NP-complete. Consider a game with the winning condition that nodes u and v are visited infinitely often. Clearly, there is a memoryless winning strategy for the game if and only if there is a simple cycle between u and v .

Theorem 4 *It is NP-complete to decide whether a game with a winning condition defined by the LTL formula $\mathbf{GF} p \wedge \mathbf{GF} q$ has a memoryless strategy.*

PROOF. Given a memoryless strategy we can compute whether it satisfies $\mathbf{GF} p \wedge \mathbf{GF} q$ by

- (1) removing from the graph all edges that do not satisfy the strategy and
- (2) checking whether all reachable states satisfy the CTL formula $\mathbf{AXAF} p \wedge \mathbf{AXAF} q$.

This can clearly be done in polynomial time. Thus, our problem is in NP.

To show that it is NP-hard, let P be the pattern graph that consists of 2 nodes, u and v , connected in a cycle. Suppose we are given a graph G and a mapping f , we construct a game G' from G with initial state $f(u)$, leaving all nondeterministic choices to the system, labeling $f(u)$ and $f(v)$ by p and q , respectively, and taking $\mathbf{GF} p \wedge \mathbf{GF} q$ as the winning condition. The size of G' is polynomial in the size of G , and G' has a memoryless winning strategy if and only if there are disjoint paths from $f(u)$ to $f(v)$ and from $f(v)$ to $f(u)$ in G , which is true iff G is homeomorphic to P . Thus, the problem of deciding whether a game has a memoryless winning strategy is NP-hard for the winning condition $\mathbf{GF} p \wedge \mathbf{GF} q$. \square

It follows that for LTL games in general there is no algorithm to decide whether there is a memoryless winning strategy that runs in time polynomial in the size of the underlying graph, unless $\mathbf{P} = \mathbf{NP}$. This is independent of the fact that finding a winning strategy has a high complexity in terms of the size of the LTL formula.

The same result follows if the winning condition is given in terms of a Büchi automaton since $\mathbf{GF} p \wedge \mathbf{GF} q$ is easily expressed as a Büchi automaton.

5.3 Heuristics for Memoryless Strategies

Since we cannot compute a memoryless strategy in polynomial time, we use a heuristic.

Recall that we attempt to solve the game $G = (S, s_0, I_G, C_G, \delta_G, \lambda_G, \varphi)$ by constructing a specification automaton $A = (Q, q_0, 2^{AP}, C_A, \delta_A, \lambda_A, B)$ with $I(A) = L(\varphi)$ and subsequently solving the game $G' \triangleright A$, where G' is derived from G by making all plays accepting.

Suppose that $\sigma : (S \times Q) \times I_G \rightarrow 2^{C_G \times C_A}$ is a winning strategy for $G' \triangleright A$. Let R be the set of reachable states of the product game, and let W be the set of winning states. As a first step, we construct a candidate memoryless strategy for the game. We define $\tau' : S \times I_G \rightarrow C_G$ to be

$$\tau'(s, i_G) = \left\{ c_G \mid \forall q \in Q \text{ for which } (s, q) \in R \cap W, \text{ we have} \right. \\ \left. \exists c_A \in C_A : (c_G, c_A) \in \sigma((s, q), i_G) \right\}.$$

Intuitively, we obtain τ' by taking the moves common to all reachable, winning states of the strategy automaton¹. The restriction to reachable states is important as some unreachable states may not allow any repair.

If τ' is winning, then so is σ , but the converse does not hold. We may, however, be able to make τ' winning by restricting its nondeterminism. To that end, we construct a game G' from G by restricting the transition function to adhere to τ' :

$$\delta_{G'} = \{(s, i, c, s') \in \delta \mid c \in \tau'(s, i)\}.$$

This may introduce states without a successor. We see whether we can avoid such states by computing $W' = \text{MG } S$. If we find that $s_0 \notin W'$, we cannot avoid visiting a dead-end state, and we give up trying to find a correction. If, on the other hand, $s_0 \in W'$, we get our final memoryless strategy by restricting τ' to W' :

$$\tau(s, i_G) = \tau'(s, i_G) \cap (W' \times I_G).$$

This ensures that a play that starts in W' remains there and never visits a dead-end. We thus obtain the following theorem.

Theorem 5 *If $s_0 \in W'$ then τ is a memoryless winning strategy of G .*

PROOF. Take a play $\pi = s_0 \xrightarrow{i_{G_0} c_{G_0}} s_1 \xrightarrow{i_{G_1} c_{G_1}} \dots$ of G according to τ and note

¹ We may treat multiple Büchi constraints, if present, in the same manner. This is equivalent to using the counting construction to reduce the number of constraints to one.

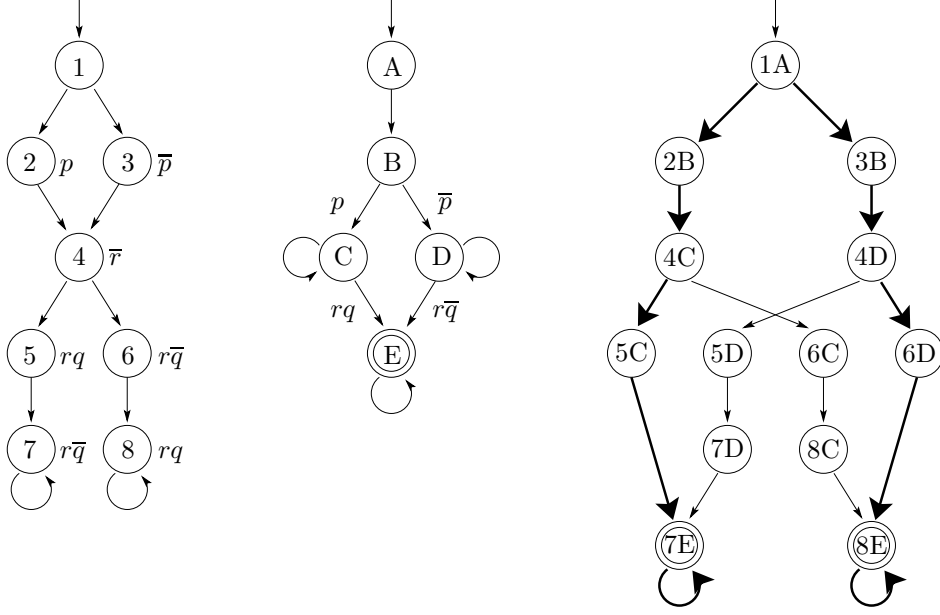


Fig. 9. The heuristic is incomplete. From left to right: an FSM, a specification automaton, and their product. Juxtaposition denotes conjunction and a bar denotes negation.

that it is also a play according to τ' . We will build a play $\pi' = (s_0, q_0) \xrightarrow{i_{G_0}(c_{G_0}c_{A_0})} (s_1, q_1) \dots$ according to σ . Since σ is winning, π' is a winning play of $G \triangleright A$ and thus $\lambda(s_0 s_1 \dots) \in I(A) = L(\varphi)$, which means that π is winning according to the condition φ of G .

We build π' inductively by choosing q_j and c_{A_j} such that π' is a play according to σ . (Note that q_0 is the initial state of A , and s_j , i_{G_j} , and c_{G_j} are as in π .) Suppose that π' is built according to σ up to (s_j, q_j) . Thus, (s_j, q_j) is reachable and since σ is winning, $(s_j, q_j) \in W$. Then, since π is a play according to τ' , $\exists c_A \in C_A : (c_{G_j}, c_A) \in \sigma((s_j, q_j), i_{G_j})$ so we can choose a c_{A_j} according to σ and find the corresponding q_{j+1} using $\delta_A(q_j, \lambda_G(s_j), c_{A_j})$. \square

Figure 9 shows that the heuristic is not complete. The game is shown in the left part of the figure. Assume that the environment can choose where to go from State 1 and the system can choose where to go from State 4. The specification automaton is shown in the middle. It assures that a p in the second cycle is eventually followed by $r \wedge q$, and similarly for $\neg p$ and $r \wedge \neg q$. Clearly, the game fulfills the specification: Each of the possible system choices in State 4 corresponds to a memoryless strategy. The right side of the figure shows the product automaton and, in bold, the attractor strategy. Because the attractor strategy always chooses the shortest path, the choices for 4C and 4D do not agree on whether to go to State 5 or 6. Thus, our heuristic does not find a winning strategy in this example.

6 Correcting the System

In this section, we wrap up the theory by showing how to extract a correction from a strategy. We show that the approach is correct and discuss its complexity.

6.1 Extracting a Correction

This section shows a symbolic method to extract a correction statement from a memoryless strategy. We determinize the strategy by finding proper assignments to the system choices that can be used for the faulty component. For any given state of the system, the given strategy may allow for multiple assignments. This gives us two possibilities. First, we can present the user with a range of possible corrections, and let her choose an appropriate one, taking into account any unstated assumptions and the user’s sense of aesthetics. Second, we can use the freedom to find a simple function in the range of allowed corrections. The user can exclude a correction and ask for an alternative if it violates an unstated assumption. In this section, we will take the second approach, and describe how to find a simple function in the allowed range.

Note that we may not want the correction to depend on certain variables of the system, for example, because they are out of the scope of the component that is being corrected. In that case, we can treat them the same way we treat the states of the automaton: we can universally quantify these variables from the strategy and its winning region and check that the strategy still supplies a valid response for all combinations of state and input. This represents an ad-hoc solution to what is effectively a problem of controller synthesis with partial observability [56–58].

For each value v_j of the system choice, we calculate a set $P_j \subseteq S \times I$ for which the value is allowed by the winning strategy. We can use these sets P_j to suggest the correction “if P_0 then v_0 else if P_1 then ...”, in which P_j is an expression that represents the set P_j . The expression P_j , however, can be quite complex: even for small examples it can take over a hundred lines, which would make the suggested correction inscrutable.

Therefore, we exploit the fact that the sets P_j can overlap to construct new sets A_j that are easier to express. We have to ensure that we still cover all winning and reachable states using the sets A_j . We assign each A_j , in order, to a small expression (in the form of a BDD) such that

$$P_j \setminus \left(\bigcup_{i < j} A_i \cup \bigcup_{i > j} P_i \right) \subseteq A_j \subseteq P_j.$$

Intuitively, A_j contains all states in P_j that are neither covered by an A_k with $k < j$, nor by a P_k with $k > j$. We then replace P_j with an expression for A_j to get our correction suggestion.

For simultaneous assignment to many variables, we may consider generating corrections for each variable separately, in order to avoid enumerating the domain. For example, we could assign the variables one by one instead of simultaneously.

Extracting a simple correction is similar to multi-level logic synthesis in the presence of satisfiability don't cares and may be amenable to multi-level minimization techniques [59]; the problem of finding the smallest expression for a given relation is NP-hard by reduction from 3SAT. One optimization we may attempt is to vary the order of the A_j s, but in our experience, the suggested corrections are typically quite readable, as long as they pertain to control logic.

6.2 Correctness

If a winning memoryless strategy for the system exists, it determines (at least) one incorrect component plus a replacement function, and, for invariants, an existing correction will always be found.

Suppose that we are given an FSM $M = (S, s_0, I, \delta, \lambda)$ and a specification φ . Recall the definitions of δ_d and δ' from Section 4. For a function $f : S \times I \rightarrow \{0, 1\}$, let $\delta[d/f]$ be the transition function obtained from δ by replacing gate d by combinational logic specified by f : $\delta[d/f](s, i) = \delta_d(s, i, f(s, i))$. Let $M[d/f]$ be the corresponding FSM.

Let $G = (S', s'_0, I, C', \delta', \lambda', \varphi)$ be the LTL game defined in Section 4 and let $\sigma : ((S \times \{0 \dots n\}) \cup s'_0) \times I \rightarrow 2^{\{0 \dots n\}}$ be a memoryless winning strategy for this game. Since the transition from the initial state s'_0 is independent of the input i , so is the strategy for this state. Let $D = \sigma(s'_0, i)$ for some i be the set of proposed faulty components.

The strategy is a relation and a correction is a function compatible with it. Formally, let \mathcal{F}_d be the set of all functions $f : S \times I \rightarrow \{0, 1\}$ such that $f(s, i) \in \{c \bmod 2 \mid c \in \sigma((s, d), i)\}$. We claim that D contains only correctable single-fault diagnoses, that $\{\mathcal{F}_d\}_{d \in D}$ contains only valid corrections, and that for invariants there are no other single correctable diagnoses and corrections.

Theorem 6 *Let $d \in \{0 \dots n\}$ and let $f : S \times I \rightarrow \{0, 1\}$. We have that $d \in D$ and $f \in \mathcal{F}_d$ implies that $M[d/f]$ satisfies φ . If φ is an invariant, then $M[d/f]$ satisfies φ implies $d \in D$ and $f \in \mathcal{F}_d$.*

PROOF. Suppose $d \in D$ and $f \in \mathcal{F}_d$. Let

$$\pi = s'_0 \xrightarrow{i'_0 d} (s_0, d) \xrightarrow{i_0 f(s_0, i_0)} (s_1, d), \dots$$

Since $f(s_j, i_j) \in \sigma((s_j, d), i_j) \pmod{2}$, π is a winning play according to σ , and $s_0, s_1, \dots \models \varphi$. Now note that

$$\begin{aligned} (s_{j+1}, d) &= \delta'((s_j, d), i_j, f(s_j, i_j)) \\ &= (\delta_d(s_j, i_j, f(s_j, i_j)), d) \\ &= (\delta[d/f](s_j, i_j), d). \end{aligned}$$

Thus, s_0, s_1, \dots is the run of $M[d/f]$ for input sequence i_0, i_1, \dots , and this run satisfies φ .

For the second part, suppose φ is an invariant, and say $M[d/f]$ satisfies φ . Then for any input sequence, the run of $M[d/f]$ satisfies φ , and from this run we can construct a winning play as above. The play stays within the winning region, and by construction of the strategy for a safety game, all system choices that do not cause the play to leave the winning region are allowed by the strategy. Thus, the play is according to the winning strategy, so $d \in D$ and $f \in \mathcal{F}_d$. \square

Note that for LTL properties in general, the theorem holds in only one direction. A correction can always be found, but its size is doubly exponential and it adds extra state to the system. If we want to avoid these drawbacks, we can not guarantee that we find a correction.

6.3 Complexity

We will first show that the (sequential) LTL repairability problem is 2-EXP complete and then we will discuss the complexity of our heuristic.

LTL-realizability is the problem of deciding, given an LTL formula φ and a partition of the atomic propositions in φ into inputs I and outputs O , whether there exists a circuit that satisfies φ . We define 1-output-LTL-realizability to be the LTL-realizability problem with the restriction that $|O| = 1$. We need the following straightforward lemma

Lemma 7 *The 1-output-LTL-realizability problem is 2EXP-complete*

PROOF. Inclusion in 2EXP follows trivially from the inclusion of LTL-realizability problem in 2EXP. To prove hardness, let φ be an LTL formula

with inputs I and outputs $O = \{y_0, \dots, y_{n-1}\}$. We will build an equirealizable formula φ' with inputs $\{c_1, \dots, c_{\lceil \lg n \rceil}\} \cup I$ and only one output, y , incurring a polynomial blowup. This proves our lemma.

The trick is to sequentialize the atomic propositions. To this end, we use the inputs c_j to encode a counter c . We define $\psi = (c = 0) \wedge \bigwedge_{i=0}^{n-1} \mathbf{G}((c = i) \rightarrow \mathbf{X}(c = i + 1 \bmod n))$.

Let \mathbf{X}^i denote a sequence of i \mathbf{X} s. Let $t(x) = x$, for $x \in I$, let $t(y_i) = \mathbf{X}^i y$, for $y_i \in O$, let $t(\mathbf{X} \chi) = \mathbf{X}^n t(\chi)$, and let $t(\chi \mathbf{U} \xi) = ((c = 0) \rightarrow t(\chi)) \mathbf{U}((c = 0) \wedge t(\xi))$, and similar for release. Then $\varphi' = \psi \rightarrow t(\varphi)$ is equirealizable to φ . \square

Theorem 8 *The LTL repairability problem is 2EXP-complete.*

PROOF. The lower bound follows from the reducibility of 1-output-LTL-realizability to LTL-repairability. A formula with one output is realizable iff the circuit consisting of a single output and a single arbitrary gate is repairable.

A 2EXP algorithm proceeds by converting the specification φ to a deterministic parity tree automata with a doubly exponential number of states and exponential index. An FSM is constructed for the circuit, incurring an exponential blowup. Then, the product of the automaton and the FSM corresponding to the circuit is taken as described in Section 4 and the winning strategy of the corresponding parity game is computed in time exponential in the index of the automaton and polynomial in the number of states of the automaton and in the size of the FSM. The resulting algorithm is doubly exponential in the size of the specification and singly exponential in the size of the circuit. \square

We do not know of a tight bound for the problem of combinational LTL repairability.

The complexity of the heuristic presented in this paper is singly exponential in the length of the formula and quadratic in the number of states of the game. Since the number of states of the game is $|\text{COMP}|$ times as large as the size of the FSM, the algorithm is polynomial in the size of the FSM and exponential in the size of the original circuit. (The analysis remains valid for any fixed number of simultaneous faults, it is exponential in the number of simultaneous faults.)

Like the Emerson-Lei algorithm, which is typically used for model checking [60,61], a symbolic implementation needs a quadratic number of preimage computations to compute the winning region of a Büchi game. (The number of preimages is an important measure of complexity [62].) For invariants,

model checking and correction both need a linear number of preimage computations. Thus, the time complexity of our algorithm matches that of LTL model checking, both in terms of the size of the specification and in terms of the size of the FSM.

The structure of the game makes it particularly amenable to symbolic computation. The game consists of multiple subgames, each one corresponding to a particular fault, linked by a common initial state. (See Fig. 5 for a pictorial representation of a game.) Thus, although the size of the state space increases linearly in the number of possible faults, the number of preimage computations remains quadratic in the number of states in the original system.

Although the combination of universal and existential quantification makes preimage computations more expensive and we have to do additional work to extract the correction, we expect that correction is feasible for a large class of designs for which BDD-based model checking is possible.

7 Examples

In this section we present experiments that demonstrate the applicability of our approach using examples on the source level. We have implemented our algorithm on top of VIS-2.1 [63]. We turn our examples into games by instrumenting them using a simple Perl script.

The game constructed from a program proceeds as follows. First, the system decides which component is faulty. Next, we “execute” the extended version of the program, where the environment decides any inputs and the system decides the behavior of the faulty component. Because the selection of the faulty component is performed before any inputs are passed to the program, the diagnosis does not depend on the inputs, and is valid regardless of their values.

The examples, the Perl script, and the instrumented versions of the examples are available at: <http://www.ist.tugraz.at/verify/view/Projects/FindAndFix>.

7.1 Locking Example

Fig. 10 shows an abstract program that realizes simple lock operations [64,16]. Nondeterministic choices in the program are represented by *. The specification should hold regardless of the nondeterministic choices taken, and thus the program abstracts a set of concrete programs with different if and while conditions. The method `lock()` acquires the lock, represented by the variable

<pre> int got_lock = 0; do{ 1 if (*) { 2 lock(); 3 got_lock = got_lock + 1;} 4 if (got_lock != 0) { 5 unlock();} 6 got_lock = got_lock - 1; 7 } while(*) void lock() { 11 assert(L = 0); 12 L = 1; } void unlock(){ 21 assert(L = 1); 22 L = 0; } </pre>	<pre> 0 diagnose = schoice{11,13,14,16,17} int got_lock = 0; do{ 1.0 if (diagnose = 11) 1.1 tmp = schoice(true, false); 1.2 else 1.3 tmp = *; 1.4 if (tmp) { 2 lock(); 3.0 if (diagnose = 13) 3.1 tmp = schoice(0,..,n-1); 3.2 else 3.3 tmp = got_lock + 1; 3.4 got_lock = tmp;} 4.0 if (diagnose = 14) 4.1 tmp = schoice(true, false); 4.2 else 4.3 tmp = (got_lock != 0) 4.4 if (tmp) { 5 unlock();} 6.0 if (diagnose = 16) 6.1 tmp = schoice(0,..,n-1); 6.2 else 6.3 tmp = got_lock - 1; 6.4 got_lock = tmp; 7.0 if (diagnose = 17) 7.1 tmp = schoice(true, false); 7.2 else 7.3 tmp = *; 7.4 } while(tmp) </pre>
--	---

Fig. 10. Locking Example

Fig. 11. Instrumented Locking Example

L, if it is available. If the lock is already held, the assertion in Line 11 is violated. In the same way, `unlock()` releases the lock, if it is held. The fault is located in Line 6, which should be within the scope of the `if` command. This example is particularly interesting because the error is caused by switching lines, which does not fit our fault model.

In this example, we choose expressions to be the components. The example has five expressions: the conditions of the `if` statements in Line 1 and 4, the `while` statement in Line 7, and the right-hand side (RHS) of the assignments to `got_lock` in Line 3 and 6. In order to illustrate the instrumentation of the source code, Fig. 11 shows an instrumented version of the program. In Line 0 we have introduced a variable `diagnose`. The game chooses one of the five candidates for `diagnose`. (Function `schoice` represents a system choice.)

If a line is selected by `diagnose`, the game determines a new value for the

```

1  int least = input1;
2  int most = input1;

3  if(most < input2)
4    most = input2;
5  if(most < input3)
6    most = input3;
7  if(least > input2)
8    most = input2;
9  if(least > input3)
10   least = input3;

11 assert (least <= most);

```

Fig. 12. MinMax Example

expression in that line (again represented by the function `schoice`). Note that the original values are kept in the other lines.

The algorithm finds three possible repairs.

- (1) Set the if-condition in Line 1 to 1. Both `lock()` and `unlock()` are then called in every loop iteration.
- (2) Set the loop condition to false in Line 7. Clearly that works, because the loop is now executed only once and the wrong value of `got_lock` does not matter.
- (3) Set `got_lock` to 0 in Line 6. This is a valid correction, because now `unlock()` is only called if `got_lock` has been incremented before in Line 3.

The last suggestion is satisfactory: it is a correction for the program no matter which concrete conditions are used for the if and while conditions.

Note that our method does not recognize the intent of the designer to place the assignment to `got_lock` within the scope of the `if`, but it finds a correction (arguably a simpler one) regardless.

7.2 *Minmax Example*

Minmax is a simple program to compute the maximum and the minimum of three input values [19]. The minimum is stored in `least`, the maximum is stored in `most`. The fault is located in Line 8 in Fig. 12. Instead of assigning `input2` to `least` the value is assigned to `most`.

As components, we choose the expressions plus the left-hand sides of the expressions. Thus, we have a larger class of possible faults than in the last

example. Note that a correction for a left-hand side should be independent of the state of the program. Therefore, the corrections for the left-hand side are decided together with the faulty components before the inputs are read.

The algorithm provides two diagnoses and the corresponding corrections. The algorithm suggests setting the if-condition in Line 7 to false. In Line 8 more than one correction is possible. The algorithm suggests changing the LHS of the assignment to `least`, or changing the RHS either to `input1` or to `input3`. It is obvious that all of the suggested corrections are valid for the assertion `(least <= most)`, and that the assertion is too weak: it does not guarantee that the smallest value is assigned to `least` and the largest value to `most`. We make the assertion more precise:

```
(least <= input1) && (least <= input2) && (least <= input3) &&
(most >= input1) && (most >= input2) && (most >= input3)
```

With this specification we find one diagnosis and correction: Change the LHS from `most` to `least` in Line 8.

As stated before, our approach is not restricted to invariants. In order to show the applicability of our approach, we change the specification. We modify the program to initialize `error` to 1 and to set it to 0 if the assignment holds. We also change the property to the LTL formula $F(\text{error} = 0)$, meaning: “`error` must eventually be equal to 0”. Although this is not an invariant, our algorithm is again able to find the correction.

7.3 *Sequential Multiplier*

The four-bit sequential multiplier shown in Fig. 13 is introduced in [42] to show the limits of model-based diagnosis for sequential circuits. The multiplier has two input shift-registers `A` and `B`, and a register `Q` which stores intermediate data. If input `INIT` is high, shift registers `A` and `B` are loaded with the inputs and `Q` is reset to zero. In every clock cycle register `A` is shifted right and register `B` is shifted left. The least significant bit (LSB) of `A` is the control input for the multiplexer. If it is high, the multiplexer forwards the value of `B` to the adder, which adds it to the intermediate result stored in register `Q`. After four clock cycles `Q` holds the product $A \cdot B$.

The multiplier has a fault in the adder: The output of the single-bit full adder responsible for bit 0 always adds 1 to the correct output. The components we use for fault localization are the eight full adders, the eight AND gates in the multiplexer, and the registers `A`, `B`, and `Q`.

Our approach is able to find the faulty part in the adder and provides a correction for all possible inputs. It suggests using an OR gate for bit 0. This

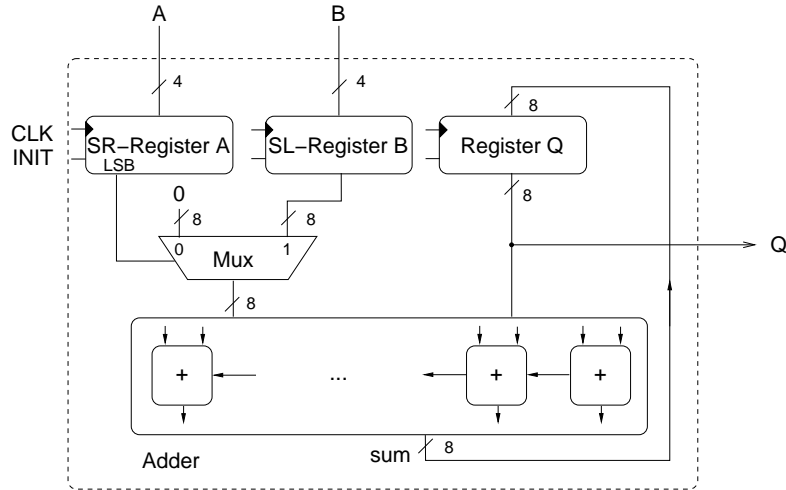


Fig. 13. Sequential Multiplier

is simpler than the correction we expected (a half adder) and still correct: In the first time step, Q is 0 and in all subsequent steps, the LSB of B is 0 because B is shifted left. Thus, the result of adding the least significant bits of B and Q is always 0 or 1. No other corrections are suggested.

Let us contrast our approach to model based diagnosis. If we load A and B with 6 and 9, respectively, the output is 58 instead of 54. Consistency-based diagnosis finds the registers B and Q , the AND gate for bit two in the multiplexer and the full adders for the three least significant bits as candidates. We can reduce the number of diagnoses by using multiple test cases and computing the intersection of the reported diagnoses. However, the full adder for bit one is a candidate in every test case. To see this, note that after four time slices the computed result is the correct value plus four. Regardless of the inputs, the carry bit of the full adder for bit 1 will have value 1 in at least one time step. If we change this value to 0, the calculated result of the multiplication is reduced by four and we obtain the correct result. Likewise, Q is a diagnosis for every test case. This example shows once more that consistency-based diagnosis finds candidates that cannot have caused the fault.

The example can also be used to show that it is not possible to correct a fault using a single test case: for any single test case there is a valid correction for the full adder for bit one. There is not, however, one correction that is valid for all test cases. (Cf. [41].) This conclusion can only be reached by considering multiple inputs, which is what our approach does.

7.4 Critical Sections

Fig. 14 demonstrates how to cope with problems when checking properties that have no deterministic automaton (see Section 5.1). The example from [25] depicts two processes that share `flag` and `turn` variables, which are used to avoid concurrent access to the variables `x` and `y` [65]. The example contains an arbiter (not shown) that nondeterministically yields control to either Process A or B, and records its choice in the variable `arbiter`. The fault is that `turn1B` is set to `false` in Line 2 of Process A. The correct value is `true`. This can cause both a deadlock and a violation of the critical region of `x`.

To check if Process B is eventually allowed to access `x` when it is waiting for it, we check the property $FairArbiter \rightarrow G(Bwaiting \rightarrow F \neg Bwaiting)$ where $FairArbiter = GF(arbiter = A) \wedge GF(arbiter = B)$ and $Bwaiting$ is true whenever Process B is in Lines 3 or 7. As the implication leads to a negation of $FairArbiter$, the automaton for the formula is nondeterministic. Our algorithm cannot find a strategy for the product game of the program and this automaton (See Fig. 7). We bypass this problem by manually changing the arbiter to switch processes infinitely often. The repair found in that setting (described below) is correct also in the more general setting, as is easily verified by model checking.

Since the fault causes the violation of two independent properties in our specification, we use the approach on the two properties separately and on a conjunction of the two properties in order to compare the results.

We start with the property that only checks for violation of the critical regions of `x` and `y`. Our approach reports three possible error locations and corrections.

- (1) Change the assignment value in Line 2 in Process A to `true`. This is the actual location of the fault.
- (2) Set the while-condition in Line 3 in Process A to `true`. This is a valid correction because we only ask for mutual exclusion and we do not violate the critical regions if we prevent progress in Process A.
- (3) Likewise, the algorithm suggests preventing progress in Process B by setting the while-condition in Line 3 in Process B to `true`.

In the second experiment we use the property that checks starvation of Process A and B respectively. Again, our approach reports three possible error locations.

- (1) The actual fault location is reported: change the assignment value in Line 2 in Process A to `true`.
- (2) Change the assignment value in Line 1 in Process A to `false`. This ensures that Process A and B make progress, but does not guarantee mutual

<pre> Process A 1 flag1A = true; 2 turn1B = false; 3 while(flag1B && turn1B); 4 x = x && y; 5 flag1A = false; 6 if(turn1B){ 7 flag2A = true; 8 turn2B = true; 9 while(flag2B && turn2B); 10 y = false; 11 flag2A = false;} 12 goto 1; </pre>	<pre> Process B 1 flag1B = true; 2 turn1B = false; 3 while(flag1A && !turn1B); 4 x = x && y; 5 flag2B = true; 6 turn2B = false; 7 while(flag2A && !turn2B); 8 y = !y; 9 x = x y; 10 flag2B = false; 11 flag1B = false; 12 goto 1; </pre>
--	--

Fig. 14. Critical Section Example

exclusion.

- (3) The algorithm suggests ensuring progress in process B by setting the while-condition in Line 3 in Process B to `false`.

The intersection of the results provides only one location and correction that is valid for both experiments: changing the assignment in Line 2 in Process A. We confirm this by running a third experiment where we use the conjunction of the properties for mutual exclusion and starvation. As expected, the algorithm yields only one suggestion corresponding to the change in Line 2.

7.5 TCAS

To compare the quality of our results with related work, we use the TCAS example used in a number of papers on error localization [18,19,66]. TCAS (Traffic Collision Avoidance System) is a task of the Siemens test suite [67] and consists of about 150 lines of C-code in 41 different versions with known errors. The suite also contains a set of test cases and their results for the different TCAS-versions. We consider a program line to be a possible fault location, if the approach results in a repair at that line.

The translation from C-code to Verilog is done with a Perl script. To avoid the big state space induced by the 32 bit integer variables in C, all constants from the original example are divided by 100 and the bit-size for the Verilog example is reduced to 8 Bit. Verification that these changes do not change the relevant behavior of the correct and faulty versions of TCAS is performed using the CBMC model checker [68] on appropriately modified versions of the C code. As our tool reports repairs as Boolean expressions, arbitrary repairs on the data path are hard to recognize by the human user. (See also future

work in Section 8.) Therefore, we limit ourselves to faults in the control flow, and variations of the original structure including switched binary operators ($<$, \leq , $>$, \geq), use of the wrong constant within an expression and `true` and `false` for assignments to Boolean variables. Overall, 25 positions were instrumented.

To compare our results with previous work, we use the specification defined in [19] for 5 of the faulty versions of TCAS and the scoring function proposed by Renieris and Reiss [18] to evaluate our results. The scoring function is a measure on the program dependency graph (PDG) for the distance between the fault candidates reported in the tool and the lines where the faults were introduced. Higher numbers are better and reflect a small amount of lines that have to be searched before reaching the fault. We use CodeSurfer [69] to generate the PDG, the score is computed using code provided by Manos Renieris.

The approach performs very well for all of the examples used in [19]. The score for all five examples is higher than 0.97, which exceeds all results from localization approaches we are aware of.

7.6 Processor

In order to compare the efficiency of the correction algorithm to that of model checking, we have introduced a fault in the control logic of VSA-R. VSA-R is an 8-bit version of a simple DLX-style processor by Fabio Somenzi [70]. The Verilog description of VSA-R has 149 lines of code (excluding comments), and all statements involving control variables were annotated (in total 25 locations). The algorithm finds one reparable location and the correct corresponding repair.

On a 2.8GHz Linux machine with 2GB of RAM, model checking completes in about 2.5 seconds. Computing the correction takes about 2.8s.

8 Conclusions and Future Work

We have presented an integrated approach to localizing and correcting faults in finite-state systems for which a specification is given in LTL. Our approach uses a very general fault model in which a component is replaced by an arbitrary new (combinational) function. The choice of component is free in our approach and can be adapted to the application at hand. Although we have formulated the approach for single faults, it is applicable to localization and correction of multiple faults as well.

Our approach proceeds by building the product of a game corresponding to the faulty program and the automaton reflecting the specification. If the product game has a winning strategy, we can correct the program. However, because of nondeterminism, a strategy may not exist for the product even if a correction exists. We could circumvent this problem by determinizing the automaton, but the cost is exponential and for many combinations of program and specification, nondeterminism is not a problem.

A winning finite state strategy corresponds to a correction that introduces new state. In order to find a local, combinational correction, we turned to the problem of finding a memoryless strategy. We have shown that deciding whether a memoryless strategy exists is NP-complete, and we have presented a conservative heuristic that conjoins the strategies for the different states of the automaton. We have also described a heuristic that finds an efficient correction for a given memoryless strategy. The algorithm is complete for invariants as they have deterministic automata consisting of one state.

The complexity of the algorithm is comparable to the complexity of model checking, which makes us optimistic as to the practical applicability of the approach. We have implemented a symbolic version of the algorithm and the initial experimental results show that the algorithm finds readable corrections in acceptable time.

A natural extension of this work would be to evaluate the effect of determinizing the automaton before computing a strategy. It would also be interesting to examine to what extent we can minimize the negative effects of using a finite state strategy, e.g., by using a dependent variable analysis [71] to minimize the amount of added state.

The corrections produced by our approach are Boolean expressions. Although such corrections are very readable when it comes to control logic, they are not well suited for the data path. A Boolean expression for an arithmetical expression is likely to be complicated. One may try to use techniques like the ones proposed in [72] to find such true arithmetical expressions: One can evaluate the Boolean functions on random instances, ask Daikon, the tool described in [72], for a corresponding arithmetical expression, and then check that for correspondence to the Boolean functions.

Finally, we have implemented a repair algorithm using a solver for Quantified Boolean Formulas [73]. We have also extended the repair approach to push-down games, so that we can correct Boolean programs that appear in a SLAM-style abstraction/refinement approach [74]. We have used this idea to suggest repairs for faulty device drivers written in C [75].

References

- [1] A. Church, Logic, arithmetic and automata, in: Proceedings International Mathematical Congress, 1962.
- [2] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proc. Symposium on Principles of Programming Languages (POPL '89), 1989, pp. 179–190.
- [3] B. Jobstmann, R. Bloem, Optimizations for LTL synthesis, in: 6th Conference on Formal Methods in Computer Aided Design (FMCAD'06), 2006, pp. 117–124.
- [4] B. Jobstmann, S. Galler, M. Weiglhofer, R. Bloem, Anzu: A tool for property synthesis, in: Computer Aided Verification, 2007, pp. 258–262.
- [5] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, M. Weiglhofer, Automatic hardware synthesis from specifications: A case study, in: Proceedings of the Conference on Design, Automation and Test in Europe, 2007, pp. 1188–1193.
- [6] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, M. Weiglhofer, Specify, compile, run: Hardware form PSL, in: 6th International Workshop on Compiler Optimization Meets Compiler Verification, 2007.
- [7] B. Jobstmann, A. Griesmayer, R. Bloem, Program repair as a game, in: K. Etessami, S. K. Rajamani (Eds.), 17th Conference on Computer Aided Verification (CAV'05), Springer-Verlag, 2005, pp. 226–238, LNCS 3576.
- [8] S. Staber, B. Jobstmann, R. Bloem, Finding and fixing faults, in: D. Borrione, W. Paul (Eds.), 13th Conference on Correct Hardware Design and Verification Methods (CHARME '05), Springer-Verlag, 2005, pp. 35–49, LNCS 3725.
- [9] S. Staber, B. Jobstmann, R. Bloem, Diagnosis is repair, in: 16th International Workshop on Principles of Diagnosis, 2005, pp. 169–174.
- [10] E. Clarke, O. Grumberg, K. McMillan, X. Zhao, Efficient generation of counterexamples and witnesses in symbolic model checking, in: Proceedings of the Design Automation Conference, San Francisco, CA, 1995, pp. 427–432.
- [11] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Trail-directed model checking, *Electronic Notes in Theoretical Computer Science* 5 (3), Software Model Checking Workshop 2001.
- [12] K. Ravi, F. Somenzi, Minimal assignments for bounded model checking, in: International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04), Barcelona, Spain, 2004, pp. 31–45, LNCS 2988.
- [13] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering* 28 (2) (2002) 183–200.

- [14] H. Jin, K. Ravi, F. Somenzi, Fate and free will in error traces, *Software Tools for Technology Transfer* 6 (2) (2004) 102–116.
- [15] A. Zeller, Isolating cause-effect chains from computer programs, in: 10th International Symposium on the Foundations of Software Engineering (FSE-10), 2002, pp. 1–10.
- [16] A. Groce, W. Visser, What went wrong: Explaining counterexamples, in: *Model Checking of Software: 10th International SPIN Workshop*, Springer-Verlag, 2003, pp. 121–135, LNCS 2648.
- [17] T. Ball, M. Naik, S. K. Rajamani, From symptom to cause: Localizing errors in counterexample traces, in: 30th Symposium on Principles of Programming Languages (POPL 2003), 2003, pp. 97–105.
- [18] M. Renieris, S. P. Reiss, Fault localization with nearest neighbor queries, in: *International Conference on Automated Software Engineering*, Montreal, Canada, 2003, pp. 30–39.
- [19] A. Groce, S. Chaki, D. Kroening, O. Strichman, Error explanation with distance metrics, *International Journal on Software Tools Technology Transfer* 8 (3) (2006) 229–247.
- [20] J. C. Madre, O. Coudert, J. P. Billon, Automating the diagnosis and the rectification of design error with PRIAM, in: *Proceedings of the International Conference on Computer-Aided Design*, 1989, pp. 30–33.
- [21] H.-T. Liaw, J.-H. Tsiah, C.-S. Lin, Efficient automatic diagnosis of digital circuits, in: *Proceedings of the International Conference on Computer-Aided Design*, 1990, pp. 464–467.
- [22] P.-Y. Chung, Y.-M. Wang, I. N. Hajj, Logic design error diagnosis and correction, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2 (1994) 320–332.
- [23] M. Tomita, T. Yamamoto, F. Sumikawa, K. Hirano, Rectification of multiple logic design errors in multiple output circuits, in: *Proceedings of the Design Automation Conference*, 1994, pp. 212–217.
- [24] A. Wahba, D. Borrione, Design error diagnosis in sequential circuits, in: *Correct Hardware Design and Verification Methods (CHARME'95)*, 1995, pp. 171–188, LNCS 987.
- [25] F. Buccafurri, T. Eiter, G. Gottlob, N. Leone, Enhancing model checking in verification by AI techniques, *Artificial Intelligence* 112 (1999) 57–104.
- [26] A. Ebnenasir, S. Kulkarni, B. Bonakdarpour, Revising UNITY programs: Possibilities and limitations, in: *Proc. Principles of Distributed Systems (OPODIS'05)*, 2005, pp. 275–290.
- [27] K.-H. Chang, I. L. Markov, V. Bertacco, Fixing design error with counterexamples and resynthesis, in: *Asia and South Pacific Design Automation Conference (ASP-DAC'07)*, 2007, pp. 944–949.

- [28] M. U. Janjua, A. Mycroft, Automatic correction to safety violations in programs, in: Thread Verification (TV'06), 2006, Unpublished.
- [29] L. T. Solar-Lezama, A., R. Bodik, V. Saraswat, D. Seshia, Combinatorial sketching for finite programs, in: Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), 2006, pp. 404–415.
- [30] L. Console, G. Friedrich, D. Theseider Dupré, Model-based diagnosis meets error diagnosis in logic programs, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93), Morgan-Kaufmann, 1993, pp. 1494–1499.
- [31] M. Stumptner, F. Wotawa, A model-based approach to software debugging, in: Proceedings on the Seventh International Workshop on Principles of Diagnosis, 1996.
- [32] G. Friedrich, M. Stumptner, F. Wotawa, Model-based diagnosis of hardware designs, in: European Conference on Artificial Intelligence, 1996, pp. 491–495.
- [33] C. Mateis, M. Stumptner, F. Wotawa, A value-based diagnosis model for Java programs, in: Proceedings of the Eleventh International Workshop on Principles of Diagnosis, 2000.
- [34] L. Console, P. Torasso, A spectrum of logical definitions of model-based diagnosis, *Computational Intelligence* 7 (3) (1991) 133–141.
- [35] D. L. Poole, R. Goebel, R. Aleliunas, Theorist: a logical reasoning system for defaults and diagnosis, in: N. Cercone, G. McCalla (Eds.), *The Knowledge Frontier: Essays in the Representation of Knowledge*, Springer Verlag, 1987, pp. 331–352.
- [36] J. d. Kleer, B. C. Williams, Diagnosing multiple faults, *Artificial Intelligence* 32 (1987) 97–130.
- [37] R. Reiter, A theory of diagnosis from first principles, *Artificial Intelligence* 32 (1987) 57–95.
- [38] S. Staber, G. Fey, R. Bloem, R. Drechsler, Automatic fault localization for property checking, in: Second International Haifa Verification Conference (HVC 2006), 2006, pp. 50–64.
- [39] B. Peischl, F. Wotawa, Automated source-level error localization in hardware designs, *IEEE Design and Test of Computers* 23 (2006) 8–19.
- [40] M. Fahim Ali, A. Veneris, S. Safarpur, R. Drechsler, A. Smith, M. Abadir, Debugging sequential circuits using Boolean satisfiability, in: International Conference on Computer Aided Design, 2004, pp. 204–209.
- [41] M. Stumptner, F. Wotawa, Debugging functional programs, in: Proceedings on the 16th International Joint Conference on Artificial Intelligence, 1999.
- [42] W. Hamscher, R. Davis, Diagnosing circuits with state: An inherently underconstrained problem, in: Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI'84), 1984, pp. 142–147.

- [43] A. Pnueli, The temporal logic of programs, in: IEEE Symposium on Foundations of Computer Science, Providence, RI, 1977, pp. 46–57.
- [44] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, Cambridge, MA, 1999.
- [45] P. J. G. Ramadge, W. M. Wonham, The control of discrete event systems, Proceedings of the IEEE 77 (1989) 81–98.
- [46] P. Wolper, M. Y. Vardi, A. P. Sistla, Reasoning about infinite computation paths, in: Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, 1983, pp. 185–194.
- [47] O. Lichtenstein, A. Pnueli, Checking that finite state concurrent programs satisfy their linear specification, in: Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, 1985, pp. 97–107.
- [48] W. Thomas, On the synthesis of strategies in infinite games, in: Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, 1995, pp. 1–13, LNCS 900.
- [49] O. Kupferman, M. Y. Vardi, Freedom, weakness, and determinism: From linear-time to branching-time, in: Proc. 13th IEEE Symposium on Logic in Computer Science, 1998.
- [50] R. Sebastiani, S. Tonetta, “More deterministic” vs. “smaller” Büchi automata for efficient LTL model checking, in: Correct Hardware Design and Verification Methods (CHARME’03), Springer-Verlag, Berlin, 2003, pp. 126–140, LNCS 2860.
- [51] M. Maidl, The common fragment of CTL and LTL, in: Proc. 41th Annual Symposium on Foundations of Computer Science, 2000, pp. 643–652.
- [52] R. Gerth, D. Peled, M. Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, in: Protocol Specification, Testing, and Verification, Chapman & Hall, 1995, pp. 3–18.
- [53] F. Somenzi, R. Bloem, Efficient Büchi automata from LTL formulae, in: E. A. Emerson, A. P. Sistla (Eds.), Twelfth Conference on Computer Aided Verification (CAV’00), Springer-Verlag, Berlin, 2000, pp. 248–263, LNCS 1855.
- [54] R. Alur, S. La Torre, Deterministic generators and games for LTL fragments, in: Symposium on Logic in Computer Science (LICS’01), 2001, pp. 291–302.
- [55] S. Fortune, J. Hopcroft, J. Wyllie, The directed subgraph homeomorphism problem, Theoretical Computer Science 10 (1980) 111–121.
- [56] F. Lin, W. M. Wonham, On observability of discrete-event systems, Information Sciences 44 (188) 173–198.
- [57] M. Y. Vardi, An automata-theoretic approach to fair realizability and synthesis, in: Seventh Conference on Computer Aided Verification (CAV’95), Springer-Verlag, 1995, pp. 267–278.

- [58] O. Kupferman, M. Y. Vardi, Church's problem revisited, *The Bulletin of Symbolic Logic* 5 (1999) 245–263.
- [59] G. D. Hachtel, F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, Boston, MA, 1996.
- [60] E. A. Emerson, C.-L. Lei, Efficient model checking in fragments of the propositional mu-calculus, in: *Proceedings of the First Annual Symposium of Logic in Computer Science*, 1986, pp. 267–278.
- [61] K. Ravi, R. Bloem, F. Somenzi, A comparative study of symbolic algorithms for the computation of fair cycles, in: W. A. Hunt, Jr., S. D. Johnson (Eds.), *Formal Methods in Computer Aided Design*, Springer-Verlag, 2000, pp. 143–160, LNCS 1954.
- [62] R. Bloem, H. N. Gabow, F. Somenzi, An algorithm for strongly connected component analysis in $n \log n$ symbolic steps, *Formal Methods in System Design* 28 (1) (2006) 37–56.
- [63] R. K. Brayton, et al., VIS: A system for verification and synthesis, in: T. Henzinger, R. Alur (Eds.), *Eighth Conference on Computer Aided Verification (CAV'96)*, Springer-Verlag, Rutgers University, 1996, pp. 428–432, LNCS 1102.
- [64] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: *Proceedings of the 29th symposium on Principles of programming languages (POPL'02)*, ACM Press, 2002, pp. 58–70.
- [65] G. L. Peterson, Myths about the mutual exclusion problem, *Inf. Process. Lett.* 12 (3) (1981) 115–116.
- [66] A. Griesmayer, S. Staber, R. Bloem, Automated fault localization for C programs, *Electronic Notes in Theoretical Computer Science* 174, *Workshop on Verification and Debugging (V&D'06)*.
- [67] H. Do, S. G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact., *Empirical Software Engineering: An International Journal* 10 (2005) 405–435.
- [68] E. M. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, 2004, pp. 168–176.
- [69] P. Anderson, T. Teitelbaum, Software inspection using codesurfer, *Workshop on Inspection in Software Engineering*.
- [70] Vis verification benchmarks. <http://vlsi.colorado.edu/~vis>.
- [71] A. J. Hu, D. Dill, Reducing BDD size by exploiting functional dependencies, in: *Proceedings of the Design Automation Conference*, Dallas, TX, 1993, pp. 266–271.

- [72] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, *IEEE Transactions on Software Engineering* 27 (2) (2001) 1–25.
- [73] S. Staber, R. Bloem, Fault localization and correction with QBF, in: Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT 2007), 2007, pp. 355–368.
- [74] T. Ball, S. K. Rajamani, Automatically validating temporal safety properties of interfaces, in: M. Dwyer (Ed.), 8th International SPIN Workshop, Springer-Verlag, Toronto, 2001, pp. 103–122, LNCS 2057.
- [75] A. Griesmayer, R. Bloem, B. Cook, Repair of Boolean programs with an application to C, in: 18th Conference on Computer Aided Verification (CAV’06), 2006, pp. 358–371, LNCS 4144.