Finding and Fixing Faults[⋆]

Stefan Staber, Barbara Jobstmann, and Roderick Bloem

Graz University of Technology

Abstract. We present a method for combined fault localization and correction for sequential systems. We assume that the specification is given in linear-time temporal logic and state the localization and correction problem as a game that is won if there is a correction that is valid for all possible inputs. For invariants, our method guarantees that a correction is found if one exists. The set of fault models we consider is very general: components can be replaced by arbitrary new functions. We compare our approach to model based diagnosis and show that it is more precise. We present experimental data that supports the applicability of our approach, obtained from a symbolic implementation of the algorithm in the Vis model checker.

1 Introduction

Knowing that a program has a bug is good. Knowing its location is even better, but only a fix is truly satisfactory.

Even if a failure trace is available, it may be hard work to find the fault contained in the system. Researchers have taken different approaches to alleviate this problem. One approach is to make the traces themselves easier to understand. In the setting of model checking, [JRS02] introduces an approach that identifies points of choice in the failure trace that cause the error and [RS04] 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 [ZH02] consider the problem of simplifying the input that causes failure.

A second approach to help the user understand a failure (which is not necessarily the same as locating the fault) is to consider several similar program traces, some of which show failure and some success [Zel02, GV03, BNR03, RR03, Gro04]. The similarities between failure traces and their differences with the successful traces give an indication of the parts of the program that are likely involved in the failure.

A third approach, which aims to locate the fault, is based on a theory of diagnosis, originally developed for physical systems. We discuss this approach in Section 2 as it warrants a more detailed description.

In this paper, we take the view that a component may be responsible for a fault if it can be replaced by an alternative that makes the system correct. Thus fault localization and correction are closely connected, and we present an approach that 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-time

[⋆] This work was supported in part by the European Union under contract 507219 (PROSYD).

temporal logic (LTL), and we endeavor to find and fix a fault in such a way that the new system satisfies its specifications 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.

Jobstmann et al. [JGB05] present a method for the repair of a set of suspect components. The most important weakness in that work is that a suspicion of the location of the fault has to be given by the user. We solve that weakness by integrating fault localization and correction.

We consider the fault localization and correction problem as an infinite game in which the system is the protagonist and the environment the antagonist. The winning condition for the protagonist is the satisfaction of the specification. The system first chooses which component is incorrect and then, at every clock cycle, which value to use as the output of the component. If for any input sequence, the system can choose outputs of the component such that the system satisfies the specification, the game is won. If the corresponding strategy is memoryless (the output of the component depends only on the state of the system and its inputs), it prescribes a replacement behavior for the component that makes the system correct. The method is complete for invariants, and in practice works well for general LTL properties, even though it is not complete.

Much work has been done in correcting combinational circuits. Typically, a correct version of the circuit is assumed to be available. (For instance, because optimization has introduced a bug.) These approaches are also applicable to sequential circuits, as long as the state space is not re-encoded. The work of [MCB89] and [LTH90] discusses formal methods of fault localization and correction based on Boolean equations. The fault model of [MCB89] is the same one we use for sequential circuits: any gate can be replaced by an arbitrary function. Chung, Wang, and Hajj [CWH94] improve these methods by pruning the set of possible faults. They consider only a set of *simple*, frequently occurring design faults. In [TYSH94] an approach is presented that may fix multiple faults of limited type by generating special patterns.

Work on sequential diagnosis 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 [WB95] 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 with a specification given in a temporal logic. Their 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, make a correct output incorrect or leave an incorrect output 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.

The paper is structured as follows. In Section 2, we discuss the relation of our approach to model based diagnosis and argue that the consistency-based approach is insufficiently precise. Section 3 gives the necessary definitions together with a motivating example. In Section 4, we show how the game can be solved and we prove the correctness and completeness of our approach. In Section 5, we show experimental evidence of the usability of our approach. We assume a basic understanding of LTL, see [CGP99] for an introduction.

2 Model Based Diagnosis for Fault Localization

Model based diagnosis provides a general, logic-based approach to fault localization. In this section, we describe the approach and discuss its shortcomings, which are addressed by our approach.

Model based diagnosis originates with the localization of faults in physical systems. Console et al. [CFTD93] show its applicability to fault localization in logic programs. In model based diagnosis, a model is derived from the source code of the program. It describes the actual, faulty behavior of the system. An oracle provides an example of correct behavior that is inconsistent with the actual behavior of the program. Using the model and the desired behavior, model based diagnosis yields a set of components that may have caused the fault.

Model based diagnosis comes in two flavors: abduction-based and consistency-based diagnosis [CT91]. Abduction-based diagnosis [PGA87] 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 set of fault models that we consider in this work is quite large (doubly exponential in the number of inputs and state variables to the system), and we do not consider it wise to enumerate all possible fault models. Thus, our approach should not be considered abductive.

Consistency-based diagnosis [KW87, Rei87] does not require the possible faults to be known, but rather tries to make the model consistent with the correct behavior by finding a component such that dropping any assumption on the behavior of the component causes the contradiction between the model and the correct behavior to disappear. In this setting, components are described as 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 is asserted. The task of consistency-based diagnosis is to find a minimal set Δ of components such that the assumption $\{\text{faulty}_c \mid c \in \Delta\} \cup \{\neg\text{faulty}_c \mid c \in \text{COMP} \setminus \Delta\}$ is consistent with the oracle (where COMP is the set of components).

Fahim Ali et al. [FAVS$^+$04], for example, present a SAT-based method for consistency-based diagnosis of sequential circuits in which they unroll the circuits and use multiplexers with one free input instead of the faulty predicate.

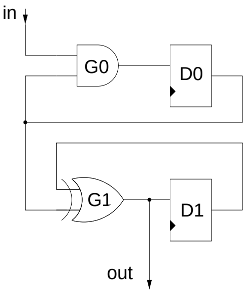

Fig. 1. Simple circuit

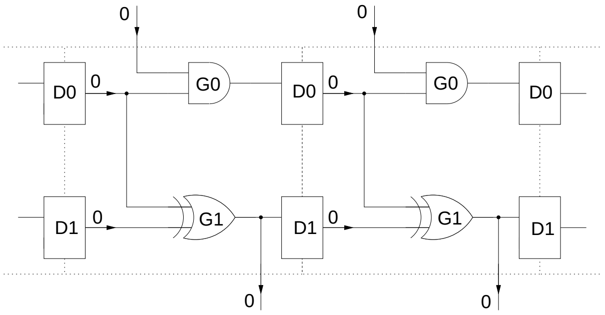

Fig. 2. Unrolling of circuit in Figure 1

Consistency-based reasoning has weaknesses when multiple instances of a component appear, for instance in the unrolling of a sequential circuit. (A similar observation is made in [SW99] for multiple test cases.) In diagnosis of sequential circuits, as in its combinational counterpart, the aim is to find a small set of components that explains the observations. A single incorrect trace is given and diagnosis is performed using the unrolling of the circuit as the model. A single faulty predicate is used for all occurrences of a given component. Hamscher and Davis [HD84] show that consistency-based diagnosis is indiscriminate in this setting: 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 is likely to hold for many components.

For instance, consider the sequential circuit shown in Figure 1. Suppose the initial state of the circuit is $(0, 0)$ and the specification is $(\text{out} = 0) \land \mathsf{G}((\text{out} = 0) \leftrightarrow \mathsf{X}(\text{out} = 1))$. Figure 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 on the outputs of 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 output of the circuit to be correct for the given inputs, 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 single consistent replacement to fix the circuit for the given input sequence is to replace the AND gate by a 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. Thus, correctability 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 significant shortcomings. The abduction-based approach can only handle a small set of possible faults, and the consistency-based method 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 single input. Our approach is precise and finds corrections that are valid for all inputs.

3 Games for Localization and Correction

Using the simple example introduced in the previous section we explain the basic ideas of our approach. Additionally, we introduce some formalisms necessary for the proof of correctness in Section 4.2.

In order to identify faulty components, we need to decide what the components of the system are. In this paper, the components that we use for circuits are gates or sets of closely related gates such as full adders. For finite-state programs, our set of components consists of all expressions and the left-hand side of each assignment. Thus, for finite-state programs both diagnosis and correction are performed at the expression level, even though an expression may correspond to multiple gates on the implementation level.

Given a set of components our approach searches for faulty components and corresponding replacement functions. The range of the replacement function depends on the component model, the domain is determined by the states and inputs. Note that the formulation of our approach is independent of the chosen set of components.

We show how to search for faulty components and correct replacements by means of sequential circuits, where the specification F is the set of runs that satisfies some LTL formula φ. Our approach can handle multiple faults, but for simplicity we use a single fault to explain it. Thus, a correction is a replacement of one gate by an arbitrary Boolean function in terms of the primary inputs and the current state.

A circuit corresponds to a *finite state machine (FSM)* $M = (S, s_0, I, \delta)$, where S is a finite set of states, $s_0 \in S$ is the initial state, I is a finite set of inputs, and $\delta : S \times I \to S$ is the transition function. For example, if we are given the circuit in Figure 1 and we want it to fulfill the specification $(\text{out} = 0) \wedge \mathsf{G}((\text{out} = 0) \leftrightarrow \mathsf{X}(\text{out} = 1))$, we obtain the FSM shown in Figure 3.

We extend the FSM to a game between the system and the environment. A *game* G is a tuple $(S, s_0, I, C, \delta, F)$, where S is a finite set of states, $s_0 \in S$ is the initial state, I and C are finite sets of environment inputs and system choices, $\delta : S \times I \times C \to S$ is the complete transition function, and $F \subseteq S^\omega$ is the winning condition, a set of infinite sequences of states. To simplify matters, we translate the given specification in a corresponding set of sequences. In our example $(\text{out} = 0) \wedge \mathsf{G}((\text{out} = 0) \leftrightarrow \mathsf{X}(\text{out} = 1))$ corresponds to all sequences in which D_1 is 0 in the first two time frames and alternates between 1 and 0 afterwards.

Suppose we are given a circuit and the gates in the circuit are numbered by $0 \ldots n$. We extend the corresponding FSM $M = (S, s_0, I, \delta)$ to a game by the following two steps

1. We extend the state space to $(S \times \{0 \ldots n\}) \cup s'_0$. 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 suspect.

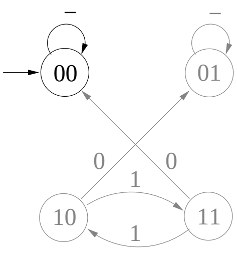

Fig. 3. Faulty system (grey parts are unreachable)

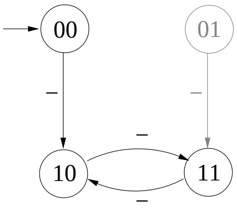

Fig. 4. Corrected system

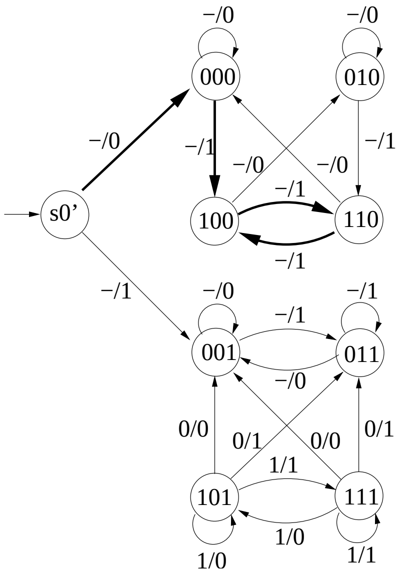

Fig. 5. Game to localize and correct the fault

2. We extend the transition relation to reflect that the system can choose the output of the suspect gate.

If gate d is suspect, it is removed from the combinational logic of our circuit, and we obtain new combinational logic with one more input (and some new outputs, which we ignore). Let the function computed by this new circuit be given by $\delta_d : S \times I \times \{0,1\} \to S$, where the third argument represents the new input.

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

$$\begin{aligned} S' &= (S \times \{0, \ldots, n\}) \cup s'_0, \\ C' &= \{0 \ldots n\}, \\ \delta'(s'_0, i, c) &= (s_0, c), \\ \delta'((s,d), i, c) &= (\delta_d(s, i, c \bmod 2), d), \\ F' &= \{s'_0, (s_0, d_0), (s_1, d_1), \cdots \mid s_0, s_1, \cdots \in F\}. \end{aligned}$$

Note that the full range of the system choice ($\{0 \ldots n\}$) is only used in the new initial state s'_0 to choose the suspect gate. Afterwards, we only need two values to decide the correct output of the gate (0 and 1), 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.

For our simple example, we obtain the game shown in Figure 5. In the initial state the system chooses which of the gates (G_0 or G_1) is faulty. The upper part of the game in Figure 5 corresponds to an arbitrary function for gate G_0, the lower one represents a

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

Table 1. Function for the system choice

replacement of gate G$_1$. The edges are labeled with the values of environment input i and the system choice c separated by a slash, e.g., the transition from state 100 to 010 labeled with $-/0$ means that starting at D$_0$ $= 1$ and D$_1$ $= 0$ and assuming G$_0$ to be faulty, the system choice $C = 0$ forces the latches to be D$_0$ $= 0$ and D$_1$ $= 1$ in the next state regardless of the input.

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. Choosing the transition from s'_0 to 000 means we try to fix the fault by replacing gate G$_0$. In state 000 we select transitions that lead to paths that adhere to the given specification. In Figure 5 the bold arrows only allow paths with the sequence 001010. . . for D$_1$ 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 3$^{\text{rd}}$ and 4$^{\text{th}}$ Line in Table 1 we can choose arbitrary values for the system choice. This choice 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. Using the corresponding transitions in the original model (Figure 3) yields the correct model shown in Figure 4.

Choosing the right transitions of the game corresponds to searching a memoryless winning strategy for the system that fulfills the winning condition F'. Formally, given a game $G = (S, s_0, I, C, \delta, F)$, a *memoryless strategy* is a function $\sigma : S \times I \rightarrow 2^C$, which fixes a set of possible responses to an environment input. A *play* on G according to σ is a finite or infinite sequence $\pi = s_0 \xrightarrow{i_0 c_0} s_1 \xrightarrow{i_1 c_1} \ldots$, such that $c_i \in \sigma(s_i, i_i)$, $s_{i+1} = \delta(s_i, i_i, c_i)$, and either the play is infinite, or $\exists n : \sigma(s_n, i_n) = \emptyset$, which means that the play is finite. A play is *winning* (for the system) if it is infinite and $s_0 s_1 \cdots \in F$. A strategy σ is *winning* on G if all plays according to σ on G are winning. Depending on the winning condition we distinguish different types of games. The winning condition of an *LTL game* is the set of sequences satisfying an LTL formula φ. A *safety game* has the condition $F = \{q_0 q_1 \cdots \mid \forall i : q_i \in A\}$ for some A. The type of the game for localizing and correction depends on the specification. In Section 4, we explain how to obtain a winning strategy and we prove the correctness of our approach.

In order to handle multiple faults we extend the game to select a set of suspect components in the initial state. In every following state the system chooses an output for the suspect component. Thus, the range of the replacement function consists of tuples of outputs, one output for each suspect component.

4 Solving Games

In Section 4.1, we summarize the approach of [JGB05] to find a winning strategy for a game, which we adopt. We describe the approach for safety games in some detail, and briefly recapitulate how to find a strategy when the specification is given in LTL. In Section 4.2, we prove that a winning strategy corresponds to a valid correction and that for invariants a winning strategy exists iff a correction exists.

4.1 Strategies

For a set $A \subseteq S$ let

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

be the set of states from which, for any input, the system can force a visit to a state in A in one step. We define $\mathsf{MG}\, A = \nu Z.A \cap \mathsf{MX}\, Z$ to be the set of states from which the system can avoid leaving A. (The symbol ν denotes a greatest fixpoint, see [CGP99].) Note that the MX operation is similar to the preimage computation in symbolic model checking, apart from the quantification of the input variables. The MG operation mirrors EG.

If the specification is an invariant A, the set $\mathsf{MG}\, A$ is exactly the set of states from which the system can guarantee that A is always satisfied. If the initial state is in $\mathsf{MG}\, A$, the game is won. The strategy for a safety game is then easily found. From any state, and for any input, select any system choice such that the next state is in $\mathsf{MG}\, A$:

$$\sigma(q, i) = \{c \in C \mid \delta(q, i, c) \in A\}.$$

Note that the strategy is immaterial for nodes that are unreachable. The same holds for states that are not winning: they will never be visited.

For LTL specifications, the situation is more intricate.

A *finite-state strategy* determines the set of allowed system choices using a finite-state machine that has a memory of the past input and system choices. A finite-state strategy may, for example, alternately pick two different choices for one and the same system state and input.

We can compute a finite-state strategy for a game with winning condition φ by finding a strategy on the product of the game and a deterministic automaton for φ. A finite-state strategy corresponds to a correction in which the new FSM is the product automaton. Thus, it would add state that corresponds to the automaton for φ.

Finding a deterministic automaton for φ is hard in terms of implementation and needs doubly exponential space. Furthermore, it is probably a bad idea to fix a simple fault by the addition of a large amount of state. Therefore, [JGB05] proposes a heuristic approach. The approach constructs a nondeterministic Büchi automaton from φ in the standard way [VW94], which causes only a singly exponential blowup. It then constructs the product of the Büchi automaton and the game. The result is a Büchi game, which in general has a finite-state strategy. To avoid adding state to the circuit, [JGB05] presents a heuristic to turn a finite-state strategy into a memoryless strategy. The heuristic works by finding choices that are common to all states of the finite-state strategy. These two heuristics imply that the method is not complete: if the property is not an invariant, a correction may not be found even if it exists. We take the view that this tradeoff is necessary for efficiency and simplicity of the correction.

Jobstmann et al. [JGB05] show how a simple correction statement is extracted from a memoryless strategy.

The complexity of the approach is comparable to that of symbolic model checking of a property on the game that has $O(k \cdot \lg |\text{COMP}|)$ more Boolean state variables than the original system, where k is the number of faults assumed.

4.2 Localization and Correction

If a winning positional strategy for the system exists, it determines (at least) one incorrect gate plus a replacement function. To see this, we need some definitions. 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 $\sigma : ((S \times \{0 \dots n\}) \cup s_0') \times I \rightarrow 2^{\{0 \dots n\}}$ be a winning finite-state strategy. Since the transition from the initial state s_0' does not depend on the input i, neither does the strategy for this state. Let $D = \sigma(s_0', i)$ for some i.

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 and $\{\mathcal{F}_d\}_{d \in D}$ contains only valid corrections, and that for invariants there are no other single correctable diagnoses or corrections.

Theorem 1. *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 F. If F is an invariant, then $M[d/f]$ satisfies F implies $d \in D$ and $f \in \mathcal{F}_d$.*

Proof. Suppose $d \in D$ and $f \in \mathcal{F}_d$. Let $\pi = (s_0', (s_0, d), (s_1, d), \dots)$ be the play of G for input sequence $i_0', i_0, i_1, \dots$ so that $(s_{j+1}, d) = \delta'((s_j, d), i_j, f(s_j, i_j))$. Since $f(s_j, i_j) \in \sigma((s_j, d), i_j)$ (mod 2), π is a winning run and $s_0, s_1, \dots \in F$. Now note that $(s_{j+1}, d) = \delta'((s_j, d), i_j, f(s_j, i_j)) = (\delta_d(s, i_j, f(s_j, i_j)), d) = (\delta[d/f](s_j, i_j), d)$. Thus, $s_0, s_1, \dots$ is the run of $M[d/f]$ for input sequence $i_0, i_1, \dots$, and this run is in F.

For the second part, suppose F is an invariant, and say $M[d/f]$ satisfies F. Then for any input sequence, the run of $M[d/f]$ is in F, 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 $f \in \mathcal{F}_d$. □

Note that for LTL properties, the theorem holds in only one direction. The primary reason for this is that a memoryless strategy may not exist for an LTL formula. Furthermore, even if a repair exists, our heuristics may fail to find it [JGB05].

5 Experiments

In this section we present initial experiments that demonstrate the applicability of our approach. We have implemented our algorithm in VIS-2.1 [B$^+$96]. In the current version of the algorithm, the examples are manually instrumented in order to obtain and solve the corresponding games. The instrumentation can easily be automated.

The game constructed from a program proceeds in three steps:

1. decide which component is faulty,
2. read the inputs to the program, and
3. execute the extended version of the program, in which one component is left unimplemented.

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 the inputs.

Our implementation is still incomplete: it builds a monolithic transition relation for the corrected system, which is intractable for large designs. We are investigating the use of partitioned relations.

5.1 Locking Example

Figure 6 shows an abstract program which realizes simple lock operations [GV03]. Nondeterministic choices in the program are represented by `*`. The specification must 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 `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 interesting because the error is caused by switching lines, which does not fit our fault model.

The components of the program that are considered for correction are the expressions in the `if` statement in Line 1, 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, Figure 8 shows an instrumented version of the program. In Line 0 we have introduced a variable `diagnose`. The game chooses one of four lines for `diagnose`. Function `choose` represents a system choice. The result of the function is one of its parameters: `l1`, `l3`, `l6` or `l7`.

If a line is selected by `diagnose`, the game determines a new value for the right-hand side in that line (again represented by the function `choose`. Note that in the other suspect lines the original values are kept.

```
  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; }
```

Fig. 6. Locking Example

```
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. 7. MinMax Example

The algorithm finds three possible error locations: Line 1, 6, or 7. The correction for Line 1 suggests to set the if-condition to `!L`. Both `lock()` and `unlock()` are then called in every loop iteration. Note that the condition could also be set to true, but the algorithm cannot exclude the possibility of reaching Line 1 with `L=1` before it fixes the strategy. The algorithm also suggests to 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. Finally, the algorithm suggests to 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 regardless.

5.2 Minmax Example

Minmax is a simple program to evaluate the maximum and the minimum of three input values [Gro04]. The minimum is stored in `least`, the maximum is stored in `most`. The fault is located of Line 8 in Figure 7. Instead of assigning `input2` to `least` the value is assigned to `most`.

We consider as possible faults the left-hand sides and right-hand sides of the assignments in Lines 4, 6, 8, and 10, and the expressions in Line 3, 5, 7, and 9. 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 assertion in Line 11 is replaced by `if !(least <= most) error=1` and we check the property $\mathsf{G}(\mathrm{error} = 0)$.

The algorithm provides two diagnoses and the corresponding corrections. The algorithm suggests to set the if-condition in Line 7 to false. In Line 8 more than one correction is possible. The algorithm suggests to change the LHS of the assignment to `least`, or to change the RHS either to `input1` or to `input3`. It is obvious that all of the suggested corrections are valid for the assertion `(least <= most)`, but that assertion does not guarantee the intended behavior of the program, namely that the minimum value is assigned to `least` and the maximum value to `most`. We make the specification 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 program and the specification. In a modified program version we initialize `error` with `1` and set it to `0` if the assignment holds. We change the specification to the LTL formula $\mathsf{X}\,\mathsf{X}\,\mathsf{F}(\text{error} = 0)$, meaning: "After two steps `error` must eventually be equal to 0". This is clearly not an invariant. Our algorithm is again able to find the correction.

5.3 Sequential Multiplier

The four-bit sequential multiplier shown in Figure 9 is introduced in [HD84]. 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 $*$ 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 in the adder, 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 to use a half adder for bit 0. This is simpler than the correction we expected 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, a carry never occurs.

Let us consider the candidates for correction that model-based diagnosis finds. 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

```
0    diagnose = choose{l1, l3, l6, l7}
     int got_lock = 0;
     do{
1.0    if (diagnose = l1)
1.1      tmp = choose(true, false);
1.2    else
1.3      tmp = *;
1.4    if (tmp) {
2        lock();
3.0    if (diagnose = l3)
3.1      tmp = choose(0,..,n-1);
3.2    else
3.3      tmp = got_lock + 1;
3.4    got_lock = tmp;}
4      if (got_lock != 0) {
5        unlock();}
6.0    if (diagnose = l6)
6.1      tmp = choose(0,..,n-1);
6.2    else
6.3      tmp = got_lock - 1;
6.4    got_lock = tmp;
7.0    if (diagnose = l7)
7.1      tmp = choose(true, false);
7.2    else
7.3      tmp = *;
7.4  } while(tmp)
```

Fig. 8. Instrumented Lock Example

the multiplication is reduced by four and we obtain the correct result. Similarly 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. This conclusion can only be reached by considering multiple inputs, which is what our approach does.

6 Conclusions

We have presented an integrated approach to localizing and correcting faults in finite-state systems with a specification given in LTL. Our approach uses a very general fault model in which a component is replaced by an arbitrary new function. Though it has been formulated for single faults, it is applicable to localization and correction of multiple faults as well.

The approach, which is based on infinite games, is sound in the sense that a suggested correction is valid for all possible input sequences. If the specification is an

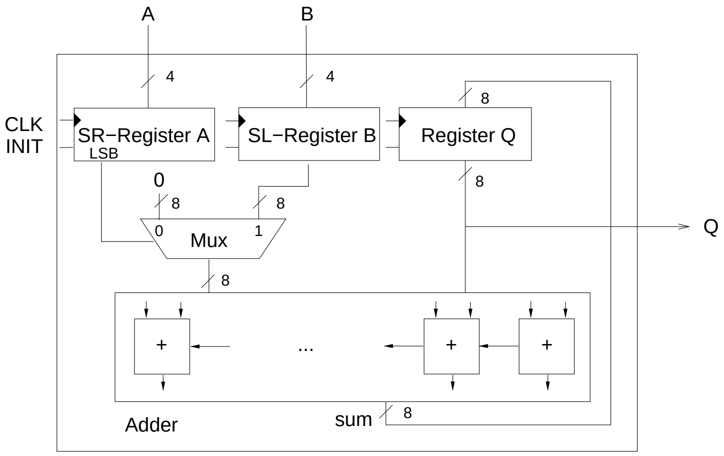

Fig. 9. Sequential Multiplier

invariant, our approach is complete: if a single point of failure exists, the fault is always found and corrected. For general LTL properties, the approach is sound and it performs well in practice, though it is not complete.

We have also shown that the most important competing localization method, model based diagnosis using consistency, does not provide the same precision in locating errors. Other known methods work with very restricted fault models, which are very useful when the fault is incurred during an incorrect optimization or re-encoding step, but does not appear to be applicable for systems for which no reference model is available.

References

[B$^+$96] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[BNR03] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *30th Symposium on Principles of Programming Languages (POPL 2003)*, pages 97–105, January 2003.

[CFTD93] L. Console, G. Friedrich, and D. Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 1494–1499. Morgan-Kaufmann, 1993.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[CT91] L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[CWH94] P.-Y. Chung, Y.-M. Wang, and I. N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:320–332, 1994.

[FAVS$^+$04] M. Fahim Ali, A. Veneris, S. Safarpur, R. Drechsler, A. Smith, and M. Abadir. Debugging sequential circuits using boolean satisfiability. In *International Conference on Computer Aided Design*, pages 204–209, 2004.

[Gro04] A. Groce. Error explanation with distance metrics. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 108–122, Barcelona, Spain, March-April 2004. LNCS 2988.

[GV03] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking of Software: 10th International SPIN Workshop*, pages 121–135. Springer-Verlag, May 2003. LNCS 2648.

[HD84] W. Hamscher and R. Davis. Diagnosing circuits with state: An inherently underconstrained problem. In *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI'84)*, pages 142–147, 1984.

[JGB05] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S. K. Rajamani, editors, *17th Conference on Computer Aided Verification (CAV '05)*, pages 226–238. Springer-Verlag, 2005. LNCS 3576.

[JRS02] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 445–459, Grenoble, France, April 2002. LNCS 2280.

[KW87] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[LTH90] H.-T. Liaw, J.-H. Tsiah, and I. N. Hajj. Efficient automatic diagnosis of digital circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 464–467, 1990.

[MCB89] J. C. Madre, O. Coudert, and J. P. Billon. Automating the diagmosis and the rectification of design error with PRIAM. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–33, 1989.

[PGA87] D. L. Poole, R. Goebel, and R. Aleliunas. Theorist: a logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer Verlag, 1987.

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[RR03] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering*, pages 30–39, Montreal, Canada, October 2003.

[RS04] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, pages 31–45, Barcelona, Spain, March-April 2004. LNCS 2988.

[SW99] M. Stumptner and F. Wotawa. Debugging functional programs. In *Proceedings on the 16th International Joint Conference on Artificial Intelligence*, 1999.

[TYSH94] M. Tomita, T. Yamamoto, F. Sumikawa, and K. Hirano. Rectification of multiple logic design errors in multiple output circuits. In *Proceedings of the Design Automation Conference*, pages 212–217, 1994.

[VW94] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[WB95] A. Wahba and D. Borrione. Design error diagnosis in sequential circuits. In *Correct Hardware Design and Verification Methods (CHARME'95)*, pages 171–188, 1995. LNCS 987.

[Zel02] A. Zeller. Isolating cause-effect chains from computer programs. In *10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 1–10, November 2002.

[ZH02] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.