Program Repair as a Game*

Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem

Graz University of Technology

Abstract. We present a conservative method to automatically fix faults in a finite state program by considering the repair problem as a game. The game consists of the product of a modified version of the program and an automaton representing the LTL specification. Every winning finite state strategy for the game corresponds to a repair. The opposite does not hold, but we show conditions under which the existence of a winning strategy is guaranteed. A finite state strategy corresponds to a repair that adds variables to the program, which we argue is undesirable. To avoid extra state, we need a memoryless strategy. We show that the problem of finding a memoryless strategy is NP-complete and present a heuristic. We have implemented the approach symbolically and present initial evidence of its usefulness.

1 Introduction

Model checking formally proves whether a program adheres to its specifications. If not, the user is typically presented with a counterexample showing an execution of the program that violates the specification. The user needs to find and correct the fault in the program, which is a nontrivial task.

The problem of locating a fault in a misbehaving program has been the attention of recent research [SW96, JRS02, BNR03, GV03, Gro04]. Given a suspicion of the fault location, it may still not be easy to repair the program. There may be multiple suggestions, only one of which is the actual fault and knowing the fault is not the same as knowing a fix.

The work presented here goes one step beyond fault localization. Given a set of *suspect statements*, it looks for a modification of the program that satisfies its specifications. It can be used to find the actual fault among the suggestions of a fault localization tool and a correction, while avoiding the tedious debugging that would normally ensue.

The repair problem is closely related to the synthesis problem [PR89]. In order to automatically synthesize a program, a complete specification is needed, which is a heavy burden on the user. For the repair problem, on the other hand, we only need as much of the specification as is necessary to decide the correct repair, just as for model checking we do not need a full specification to detect a fault. (This has the obvious drawback that an automatic repair may violate an unstated property and needs to be reviewed by a designer.) A further benefit is that 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.

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

K. Etessami and S.K. Rajamani (Eds.): CAV 2005, LNCS 3576, pp. 226–238, 2005.

We give the necessary definitions in Section 2. We assume that the specification is given in linear time logic (LTL). The program game is an LTL game that captures the possible repairs of the program, by making some value "unknown" (Section 3.1). We focus on finite-state programs and our fault model assumes that either an expression or the left-hand side of an assignment is incorrect. We can thus make an expression or a left-hand side variable "unknown". We have chosen this fault model for the purpose of illustration, and our method applies equally well to other fault models or even to circuits instead of programs.

The game is played between the environment, which provides the inputs, and the system, which provides the correct value for the unknown expression. The game is won if for any input sequence the system can provide a sequence of values for the unknown expression such that the specification is satisfied. A winning strategy fixes the proper values for the unknown expression and thus corresponds to a repair.

In order to find a strategy, 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, and the nondeterministic automaton may be exponentially smaller than a deterministic equivalent (Section 3.2).

To implement the repair corresponding to a finite state strategy, we may need to add state to the program, mirroring the specification automaton. Such a repair is unlikely to please the developer as it may significantly alter the program, inserting new variables and new assignments throughout the code. Instead, we look for a memoryless strategy, which corresponds to a repair that changes only the suspected lines and does not introduce new variables. In Section 3.3 we show that deciding whether such a strategy exists is NP-complete, so in Section 3.4 we develop a heuristic to find one.

We obtain a conservative algorithm that yields valid repairs and is complete for invariants. It may, however, fail to find a memoryless repair for other types of properties, either because of nondeterminism in the automaton or because of the heuristic that constructs a memoryless strategy. In Section 3.5 we describe a symbolic method to extract a repair from the strategy. We have implemented the algorithm in VIS and we present initial experiences with the algorithm in Section 4.

Our work is related to controller synthesis [RW89], which studies the problem of synthesizing a "controller" for a "plant". The controller synthesis problem, however, does not assume that the plant is malfunctioning, and our repair application is novel. Also, we study the problem finding a memoryless repair, which corresponds to a controller that is "integrated" in the plant. Buccafurri et al. [BEGL99] consider the repair problem for CTL as an abductive reasoning problem and present an approach that is based on calling the model checker once for every possible repair to see if it is successful. Our approach needs to consider the problem only once, considering all possible repairs at the same time, and is likely to be more efficient. Model-based diagnosis can also be used to suggest repairs for broken programs by incorporating proper fault models into the diagnosis problem. Stumptner and Wotawa [SW96] discuss this approach for functional programs. The approach appears to be able to handle only a small amount

of possible repairs, and bases its conclusions on a few failed test cases (typically one) instead of the specification.

2 Preliminaries

In this section, we describe the necessary theoretical background for our work. We assume basic knowledge of the μ-calculus, LTL, and the translation of LTL to Büchi automata. We refer to [CGP99] for an introduction.

A *game* G over AP is a tuple $(S, s_0, I, C, \delta, \lambda, F)$, 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, $\delta : S \times I \times C \rightharpoonup S$ is the partial transition function, $\lambda : S \to 2^{AP}$ is the labeling 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 3.4, we will assume that δ is a complete function. Intuitively, a game is an incompletely specified finite state machine together with a specification. The environment inputs are as usual, and the system choices C represent the freedom of implementation. The challenge is to find proper values for C such that F is satisfied.

Given a game $G = (S, s_0, I, C, \delta, \lambda, F)$, a *(finite state) strategy* is a tuple $\sigma = (Q, q_0, \mu)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\mu : Q \times S \times I \to 2^{C \times Q}$ is the *move function*. Intuitively, a strategy automaton fixes a set of possible responses to an environment input, and its response may depend on 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. For the strategy to be winning, a winning play has to ensue for any nondeterministic choices of the strategy.

A *play* on G according to σ is a finite or infinite sequence $\pi = q_0 s_0 \xrightarrow{i_0 c_0} q_1 s_1 \xrightarrow{i_1 c_1} \dots$ such that $(c_i, q_{i+1}) \in \mu(q_i, s_i, i_i)$, $s_{i+1} = \delta(s_i, i_i, c_i)$, and either the play is infinite, or there is an n such that $\mu(q_n, s_n, i_n) = \emptyset$ 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 $s_0 s_1 \cdots \in F$. (If $\mu(q_n, s_n, i_n) = \emptyset$, 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.

A *memoryless strategy* is a finite state strategy with only one state. We will write a memoryless strategy as a function $\sigma : Q \times I \to 2^C$ and a play of a memoryless strategy as a sequence $s_0 \xrightarrow{i_0 c_0} s_1 \xrightarrow{i_1 c_1} \dots$, leaving out the state of strategy automaton.

We extend the labeling function λ to plays: the *output word* is $\lambda(\pi) = \lambda(s_0)\lambda(s_1)\dots$ Likewise, the *input word* is $\iota(\pi) = i_0 i_1 \dots$, the sequence of system inputs. The *output language (input language)* $L(G)$ ($I(G)$) of a game is the set of all $\lambda(\pi)$ ($\iota(\pi)$) with π winning.

A *safety game* has the condition $F = \{s_0 s_1 \cdots \mid \forall i : s_i \in A\}$ 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 visit 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. The

system choice models the nondeterminism of the automaton. Following the construction proposed in [SB00] we get a *generalized Büchi game*, which has more than one Büchi constraint. Our approach works with such games as well but for simplicity we explain it for games with a single constraint. Besides, we can easily get rid of multiple Büchi constraints with help of the well-known counting construction. The size of the resulting automaton is exponential in the length of the formula in the worst case.

In order to solve games, we introduce some notation. For a set $A \subseteq S$, the set $\mathsf{MX}\, A = \{s \mid \forall i \in I\, \exists c \in C\, s\ \in A : (s, i, c, s\ ) \in \delta\}$ is the set of states from which the system can force a visit to a state in A in one step. The set $\mathsf{M}A\,\mathsf{U}\,B$ is defined by the μ-calculus formula $\mu Y.\, B \cup \mathsf{MX}\,(A \cap Y)$. It defines the set of states from which the system can force a visit to B without leaving A. The *iterates* of this computation are $Y_0 = B$ and $Y_{j+1} = Y_j \cup (A \cap \mathsf{MX}\, Y_{j\ \ 1})$ for $j > 0$. From Y_j the system can force a visit to B in at most j steps. Note that there are only finitely many distinct iterates.

We define $\mathsf{MG}\, A = \nu Z.A \cap \mathsf{MX}\, Z$, the set of states from which the system can avoid leaving A. For a Büchi game, we define $W = \nu Z.\mathsf{MX}\,\mathsf{M}ZU(Z \cap B)$. The set W is the set of states from which the system can win the Büchi game. Note that these fixpoints are similar to the ones used in model checking of fair CTL and are easily implemented symbolically.

Using these characterizations, we can compute memoryless strategies for safety and Büchi games [Tho95]. For a safety game with condition A, the strategy $\sigma(s, i) = \{c \in C \mid \exists s\ \in \mathsf{MG}\, A : (s, i, c, s\ ) \in \delta\}$ is winning if and only if $s_0 \in \mathsf{MG}\, A$. For a Büchi game, let $W = \nu Z.\mathsf{MX}\,\mathsf{M}Z\,\mathsf{U}(Z \cap B)$. Let Y_1 through Y_n be the set of distinct iterates of $\mathsf{M}W\,\mathsf{U}(W \cap B) = W$. We defi ne the *attractor strategy* for B to be

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

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

3 Program Repair

This section contains our main contributions. In 3.1, we describe how to obtain a program game from a program and a suspicion of a fault. The product of the program game and the automaton for the LTL formula is a Büchi game. If the product game is winning, it has a memoryless winning strategy. In 3.2 we show how to construct a finite state strategy for the program game from the strategy for the product game and we discuss under which conditions we can guarantee that the product game is winning. A finite state strategy for the program game corresponds to a repair that adds states to the program. Since we want a repair that is as close as possible to the original program, we search for a memoryless strategy. In 3.3, we show that it is NP-complete to decide whether a memoryless strategy exists, and in 3.4, we present a heuristic to construct a memoryless strategy. This heuristic may fail to find a valid memoryless strategy even if one exists. Finally, we show how to extract a repair from a memoryless strategy.

3.1 Constructing a Game

Suppose that we are given a program that does not fulfill its LTL specification φ. Suppose furthermore that we have an idea which variables or lines may be responsible for the failure, for instance, from a diagnosis tool.

A program corresponds to an LTL game $K = (S, s_0, I, \{c\}, \delta, \lambda, \varphi)$. The set of system choices is a singleton (the game models a deterministic system) and the acceptance condition is the specification. Given an expression e in which the right-hand side (RHS) may be incorrect, we turn K into a *program game* G by *freeing* the value of this expression. That is, if Ω is the domain of the expression e, we change the system choice to $C = C \times \Omega$ and let the second component of the system choices define the value of the expression. If we can find a winning memoryless strategy for G, we have determined a function from the state of the program to the proper value of the RHS, i.e., a repair.

We can generalize the fault model by including the left-hand side (LHS). Thus, we convert the program to a game by adding a system choice that determines whether the LHS or the RHS should be changed and depending on that choice, which variable is used as the LHS or which expression replaces the RHS. Then we compute the choice that makes the program correct.

We do not consider other fault models, but these can be easily added. The experimental results show that we may find good repairs even for programs with faults that we do not model.

3.2 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 $G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, F)$, where $\delta((s,q), i_G, (c_G, c_A)) = \big(\delta_G(s, i_G, c_G), \delta_A(q, \lambda_G(s), c_A)\big)$, $\lambda(s,q) = \lambda_G(s)$, and $F = \{(s_0, q_0), (s_1, q_1), \cdots \mid s_0, s_1, \cdots \in F_G \text{ and } q_0, q_1, \cdots \in F_A\}$. 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.

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 memoryless winning strategy 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 $\lambda(\pi) \in I(A)$.*

The finite state strategy σ is the product of A and the memoryless (single state) strategy for $G \triangleright A$. If $F_G = S^\omega$, then σ is the winning strategy for the game G with the winning condition defined by A. The following result (an example of *game simulation*, cf. [Tho95]) follows from Lemma 2.

Theorem 1. *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 $I(A) = L(G)$. If there is a memoryless winning strategy for the Büchi game $G' \triangleright A$ then there is a finite state winning strategy for G.*

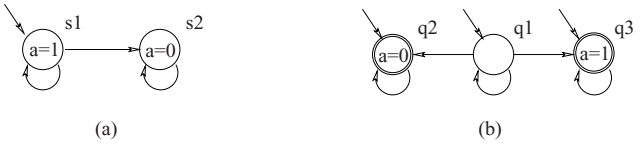

Fig. 1. (a) Game in which the environment can assign the value for variable a. (b) automaton for $\mathsf{F}\,\mathsf{G}(a = 1) \vee \mathsf{F}\,\mathsf{G}(a = 0)$

Note that the converse of the theorem does not hold. In fact, Harding [Har05] shows that we are guarantee to find a winning strategy iff the game fulfills the property and the automaton is *trivially determinizable*, i.e., we can make it deterministic by removing edges without changing the language.

For example, there is no winning strategy for the game shown in Fig. 1. If the automaton for the property $\mathsf{F}\,\mathsf{G}(a = 1) \vee \mathsf{F}\,\mathsf{G}(a = 0))$ moves to the state q_3, the environment can decide to move to s_2 (set $a = 0$), a move that the automaton 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 formula, we cannot give a strategy. Note that this problem depends not only on the structure of the automaton, but also on the structure of the game. For instance, if we remove the edge from s_1 to s_2, we can give a strategy for the product.

In general, the translation of an LTL formula to a deterministic automaton requires a doubly exponential blowup and the best known upper bound for deciding whether a translation is possible is EXPSPACE [KV98]. To prevent this blowup, we can either use heuristics to reduce the number of nondeterministic states in the automaton [ST03], or we can use a restricted subset of LTL. Maidl [Mai00] shows that translations in the style of [GPVW95] (of which we use a variant [SB00]) 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, $\varphi_1 \wedge \varphi_2$, $\mathsf{X}\,\varphi_1$, $(p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2)$, $(p \wedge \varphi_1)\,\mathsf{U}(\neg p \wedge \varphi_2)$ and $(p \wedge \varphi_1)\,\mathsf{W}(\neg p \wedge \varphi_2)$ are LTL^{det} formulas. Note that this set includes invariants ($\mathsf{G}\,p$) and $\neg p\,\mathsf{U}\,p = \mathsf{F}\,p$. LTL^{det} describes the intersection of LTL and CTL. In fact, deterministic Büchi automata describe exactly the properties expressibly in the alternation-free μ-calculus, a superset of CTL [KV98].

Alur and La Torre [AL01] 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 $\text{LTL}(\mathsf{F}, \wedge)$ and $\text{LTL}(\mathsf{F}, \mathsf{X}, \wedge)$. In contrast, for $\text{LTL}(\mathsf{F}, \vee, \wedge)$ and $\text{LTL}(\mathsf{G}, \mathsf{F})$ they show that the size of a corresponding deterministic automaton is necessarily doubly exponential in the size of the formula. Since trivially deterministic automata can be made deterministic by removing edges, they can be no smaller than the smallest possible deterministic automaton and thus there are no exponential-size trivially deterministic automata for the latter two groups.

3.3 Memoryless Strategies Are NP-Complete

As argued in the introduction, a finite state strategy may correspond to an awkward repair and therefore we wish to construct a memoryless strategy.

It follows from the results of Fortune, Hopcroft, and Wyllie [FHW80] that given a directed graph G and two nodes v and w, it is NP-complete to compute whether there are node-disjoint paths from v to w and back. Assume that we build a game G based on the graph G. The acceptance condition is that v and w are visited infinitely often, which can easily be expressed by a Büchi automaton. Since the existence of a memoryless strategy for G implies the existence of two node-disjoint paths from v to w and back, we can deduce the following theorem.

Theorem 2. *Deciding whether a game with a winning condition defined by a Büchi automaton has a memoryless winning strategy is NP-complete.*

It follows that for LTL games 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 P = NP, even if a finite state strategy is given.

3.4 Heuristics for Memoryless Strategies

Since we cannot compute a memoryless strategy in polynomial time, we use a heuristic. Given a memoryless strategy for the product game, we construct a strategy that is common to all states of the automaton, which is our candidate for a memoryless strategy on the program game. Then, we compute whether the candidate is a winning strategy, which is not necessarily the case. Note that invariants have an automaton consisting of one state and thus the memoryless strategy for the product game is a memoryless strategy for the program game.

Recall that the product game is $G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, B)$. Let $\sigma : (S \times Q) \times I_G \to 2^{C_G C_A}$ be the attractor strategy for condition B. Note that the strategy is immaterial on nodes that are either not reachable (under any choice of the system) or not winning (and thus will be avoided by the system). Let R be the set of reachable states of the product game, and let W be the set of winning states. We define

$$\tau\,(s, i_G) = \big\{c_G \;\big|\; \forall q \in Q : \big((s,q) \notin R \cap W \text{ or } \exists c_A \in C_A : (c_G, c_A) \in \sigma((s,q), i_G)\big)\big\}.$$

Intuitively, we obtain τ by taking the moves common to all reachable, winning states of the strategy automaton.[1]

If τ is winning, then so is σ, but the converse does not hold. To check whether τ is winning, we construct a game G from G by restricting the transition relation to adhere to $\tau : \delta = \{(s, i, c, s') \in \delta \mid c \in \tau\,(s, i)\}$. This may introduce states without a successor. We see whether we can avoid such states by computing $W = \mathsf{MG}\, S$. If we find that $s_0 \notin W$, we cannot avoid visiting a dead-end state, and we give up trying to find a repair. If, on the other hand, $s_0 \in W$, we get our final memoryless strategy τ by restricting τ to W, which ensures that a play that starts in W remains there and never visits a dead-end. We thus reach our main conclusion in the following theorem.

[1] We may treat multiple Büchi constraints, if present, in the same manner. This is equivalent to using the counting construction.

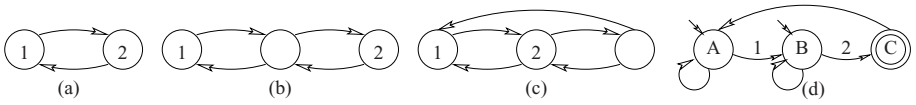

Fig. 2. Fig. a, b, c show three games with the winning condition that states 1 and 2 are both visited infinitely often. Multiple outgoing arcs from a state model a system choice. The winning condition is defined by the Büchi automaton shown in Fig. d. For Fig. a, the strategies for States A, B, and C coincide, and a memoryless strategy exists. For Fig. b, no memoryless strategy exists, and for Fig. c, a memoryless strategy exists, but it is not equal to the intersection of all the strategies for states A, B, and C. (The strategies are contradictory for the state on the right.)

Theorem 3. *If $s_0 \in W$ then τ is a winning strategy of G.*

3.5 Extracting a Repair

This section shows a symbolic method to extract a repair statement from a memoryless strategy. We determinize the strategy by finding proper assignments to the system choices that can be used in the suspect locations. For any given state of the program, the given strategy may allow for multiple assignments, which gives us room for optimization.

We may not want the repair to depend on certain variables of the program, for example, because they are out of the scope of the component that is being repaired. In that case, 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.

For each assignment to the system choice variables, we calculate a set $P_j \subseteq S \times I$ for which the assignment is a part of the given strategy. We can use these sets P_j to suggest the repair "`if P0 then assign0 else if P1 then ...`", in which `Pj` is an expression that represents the set P_j. The expression `Pj`, however, can be quite complex: even for small examples it can take over a hundred lines, which would make the suggested repair inscrutable.

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. Therefore, A_j is obtained from P_j by adding or removing states outside of a *care set*. The care set consists of all states that cannot be covered by A_j because they are not in P_j and all states that must be covered by A_j because they are neither covered by an A_k with $k < j$, nor by a P_k with $k > j$. We then replace `Pj` with an expression for A_j to get our repair suggestion.

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

Extracting a simple repair is similar to multi-level logic synthesis in the presence of satisfiability don't cares and we may be able to apply multi-level minimization techniques [HS96]; 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 `Aj`s, but in our experience, the suggested repairs are typically quite readable.

3.6 Complexity

The complexity of the algorithm is polynomial in the number of states of the system, and exponential in the length of the formula, like the complexity of model checking. A symbolic implementation needs a quadratic number of preimage computations to compute the winning region of a Büchi game, the most expensive operation, like the Emerson-Lei algorithm typically used for model checking [RBS00]. For invariants, model checking and repair both need a linear number of preimage computations. Although the combination of universal and existential quantification makes preimage computations more expensive and we have to do additional work to extract the repair, we expect that repair is feasible for a large class of designs for which model checking is possible.

In our current implementation, we build the strategy as a monolithic BDD, which may use a lot of memory. We are still researching ways to compute the strategy in a partitioned way.

4 Examples

In this section we present initial experimental results supporting the applicability of our approach on real (though small) examples.

We have implemented our repair approach in the VIS model checker [B$^+$96] as an extension of the algorithm of [JRS02]. The examples below are finite state programs given in pseudo code. They are translated to Verilog before we feed them to the repair algorithm. Suspect expressions are freed and a new system choice is added with the same domain as the expression. Assertions are replaced by `if(...) error=1` and the property $\mathsf{G}(\text{error} = 0)$. In the current version, this translation and the code augmentation are done manually.

4.1 Locking Example

We start with the abstract program shown in Fig. 3 [GV03]. This programs abstracts a class of concrete programs with different if and while conditions, all of which perform simple lock request/release operations. The method `lock()` checks that the lock is available and requests it. Vice versa, `unlock()` checks that the lock is held and releases it. The `if(*)` in the first line causes the lock to be requested nondeterministically, and the `while(*)` causes the loop to be executed an arbitrary number of times. The variable `got_lock` is used to keep track of the status of the lock (Lines 4 and 5). The assertions in Lines 11 and 21 constitute a safety property that is violated, e.g., if the loop is executed twice without requesting the lock. The fault is that the statement `got_lock--` should be placed within the scope of the preceding `if`.

Model-based diagnosis can be used to find a candidate for the repair [MSW00]. A diagnosis of the given example was performed in [CKW05] and localizes the fault in Lines 1, 6, or 7. We reject the possibility of changing Line 1 or 7 because we want the repair to work regardless of the if and while conditions in the concrete program. Instead, we look for a faulty assignment to `got_lock`. Thus, we free the RHS in Lines 3 and 6. The algorithm suggests a correct repair, `got_lock=1` for Line 3 and `got_lock=0`

```
  int got_lock = 0;
  do{
1   if (*) {
2     lock();
3     got_lock++; }
4   if (got_lock != 0) {
5     unlock();}
6   got_lock--;
7 } while(*)

  void lock() {
11  assert(L = 0);
12  L = 1; }

  void unlock(){
21  assert(L = 1);
22  L = 0; }
```

Fig. 3. 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. 4. MinMax Example

for Line 6. Note that we repair the program using a different fault model than the one which caused it, i.e., after the repair the program is correct, even though we did not suggest to move `got_lock--` inside the scope of the `if`.

4.2 MinMax

To present a more general fault model we show a simple program which assigns the minimal and maximal values out of three input values to `least` and `most`,resp.[Gro04].

The fault is located in Line 8 of Fig. 4, where `input2` is assigned to `most` (instead of `least`), which was one of five single fault diagnoses found by a model based debugger based on [MSW00]. To find the correct repair, we replace the assignments in lines 4, 6, 8, and 10 with switch-statements over the system choice that selects whether to assign to `least`, to `most`, or to replace the RHS. The algorithm correctly suggests to assign to variable `most` in Lines 4 and 6, and to `least` in Lines 8 and 10.

4.3 Critical Sections

Fig. 5 demonstrates how to cope with problems when testing properties that have no deterministic automaton (see Section 3.2). The example from [BEGL99] depicts two processes that share `flag` and `turn` variables, which are used to avoid concurrent access to the variables `x` and `y`. The process contains an arbiter (not shown) that non-deterministically 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`.

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;
```

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;
```

Fig. 5. Critical Section Example

To check if Process B is eventually allowed to access x when it is waiting for it, we check the property FairArbiter $\rightarrow \mathsf{G}(\text{Bwaiting} \rightarrow \mathsf{F}\,\neg\text{Bwaiting})$ where FairArbiter $= \mathsf{G}\,\mathsf{F}(\texttt{arbiter} = A) \wedge \mathsf{G}\,\mathsf{F}(\texttt{arbiter} = B)$) and Bwaiting is true whenever Process B is in Lines 3 or 7. As the implication leads to an negation of FairArbiter we get a nondeterministic automaton. Our algorithm cannot find a strategy for the product game of the program and this automaton (See Fig. 1).

We solve this problem by manually changing the arbiter to switch processes infinitely often. Freeing `turn1B` in Line 2 of Process A with domain {`false,true`} now leads to the correct answer, `turn1B = true`. Note that this repair also works for the original model. This repair can also be found by checking for violations of the critical section, which can be stated as a simple invariant and therefore does not require a modification of the system.

4.4 Processor

In order to compare the efficiency of repair algorithm to that of model checking, we have introduced a fault in a 16-bit version of a simple unpipelined DLX-style processor. The fault is in the ALU and the property checks that the ALU works correctly.

On a 2.8GHz Linux machine with 2GB of RAM, the model checking run needs 230 seconds to check that the property does not hold on the incorrect version. The repair algorithm finds a repair in 200 seconds, and the repair is verified to be correct by the model checker (an unnecessary precaution) in 210 seconds; all runs use around 1.2GB.

5 Conclusions

We have considered the problem of fixing a program to adhere to its specification, given a suspicion of the fault. We proceed by building the product of a game corresponding to the broken program and the automaton reflecting the specification. If the product game

has a winning strategy, we can repair the program. However, a strategy may not exist for the product even if a repair exists because of nondeterminism in the automaton. We could circumvent this problem by determinizing the automaton, but the cost is exponential and for many combinations of program and specification, nondeterminism turns out not to be problematic.

A winning finite state strategy correspond to a repair that introduces new state. We reject the possibility of changing the program logic and instead turn 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 described a heuristic that finds an efficient repair for a given memorless strategy.

The algorithm is of a complexity that is comparable to that 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 repairs in acceptable time, though improvements in the implementation are still possible.

The algorithm is complete for invariants as they have deterministic automata consisting of one state and in fact we can solve them using the linear algorithm for guarantee games.

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 see in how far we can minimize the negative effects of using a finite state strategy, e.g., by using a dependent variable analysis [HD93] to minimize the amount of added state. Finally, it would be interesting to see in how far the approach can be extended to push-down games that would result from an attempt to repair Boolean programs that appear in a SLAM-style abstraction/refinement approach [BR01]. We are looking into further improvements in the efficiency of the implementation.

References

[AL01] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. In *Symposium on Logic in Computer Science (LICS'01)*, pages 291–302, 2001.

[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.

[BEGL99] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by AI techniques. *Artificial Intelligence*, 112:57–104, 1999.

[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.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M.B. Dwyer, editor, *8th International SPIN Workshop*, pages 103–122, Toronto, May 2001. Springer-Verlag. LNCS 2057.

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

[CKW05] R. Chen, D. Köb, and F. Wotawa. A comparison of fault explanation and localization. unpublished, 2005.

[FHW80] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[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.

[Har05] A. Harding. *Symbolic Strategy Synthesis For Games With LTL Winning Conditions*. PhD thesis, University of Birmingham, 2005. Unpublished.

[HD93] A. J. Hu and D. Dill. Reducing BDD size by exploiting functional dependencies. In *Proceedings of the Design Automation Conference*, pages 266–271, Dallas, TX, June 1993.

[HS96] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, MA, 1996.

[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.

[KV98] O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

[Mai00] M. Maidl. The common fragment of CTL and LTL. In *Proc. 41th Annual Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[MSW00] C. Mateis, M. Stumptner, and F. Wotawa. A value-based diagnosis model for Java programs. In *Proceedings of the Eleventh International Workshop on Principles of Diagnosis*, 2000.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[RBS00] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, November 2000. LNCS 1954.

[RW89] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[ST03] R. Sebastiani and S. Tonetta. "more deterministic" vs. "smaller" büchi automata for efficient LTL model checking. In *Correct Hardware Design and Verification Methods (CHARME'03)*, pages 126–140, Berlin, October 2003. Springer-Verlag. LNCS 2860.

[SW96] M. Stumptner and F. Wotawa. A model-based approach to software debugging. In *Proceedings on the Seventh International Workshop on Principles of Diagnosis*, 1996.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13. Springer-Verlag, 1995. LNCS 900.