Diagnosis is Repair

Stefan Staber, Barbara Jobstmann, and Roderick Bloem
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2, 8010 Graz, Austria
{sstaber,bjobst,rbloem}@ist.tugraz.at

Abstract

We argue that for sequential circuits, fault localization and repair are one and the same problem. We assume that a specification is given in linear temporal logic and we solve the diagnosis and repair problem for finite-state programs using games. Our approach is sound and it is complete if the specification is an invariant. In contrast to known approaches, the repair we find is valid for all possible input sequences, not just for one given test case. We show the applicability of our approach, which has a complexity comparable to that of model checking, on a set of examples.

1 Introduction

Hamscher and Davis' observation that diagnosis is an inherently underconstrained problem [Hamscher and Davis, 1984] is only true for intermittent faults. When we assume that the failure is persistent, the problem is much better defined: A component is only a valid diagnosis if there is a single persistent alternative behavior that explains the observations. This behavior must be functional (the inputs fix the outputs) and therefore the usual technique of freeing the constraints corresponding to the component does not suffice.

In the usual diagnosis setting, intermittent faults may actually occur and they can also be used to model structural faults in which the component depends on a signal that is not supposed to influence it (e.g., because of a bridge fault). In the setting of fault localization [Console *et al.*, 1993], however, a diagnosis corresponds to an incorrectly chosen component, not to a faulty component, and the alternative behavior is the correct behavior that should be included in the circuit instead. The alternative behavior must thus be functional. In this setting, diagnosis is finding whether a consistent replacement explains the problem and repair is giving this replacement. If the logic that one uses has a constructive decision procedure, the two problems coincide.

Our approach to diagnosis builds on the repair approach suggested by [Jobstmann *et al.*, 2005]. The approach considers the repair problem as a game in which the protagonist is the circuit and the antagonist is the environment. The protagonist's goal is to fulfill the specification and the antagonist tries to foil that attempt. A winning strategy for the protagonist corresponds to a repair of the program. The approach builds a game from a program by leaving some decisions, which were previously fixed, for the protagonist to determine. Which decisions exactly are left to the protagonist depends on a suspicion of the fault location which is assumed to be delivered by an fault localization tool. This paper extends the approach of [Jobstmann *et al.*, 2005] by integrating fault localization with repair, closing the most important gap left in that work. Like Jobstmann et. al., we assume that the specification is given in linear-time logic (LTL) [Pnueli, 1981], and find a diagnosis and a repair that work for all possible inputs.

The applicability of our approach to structural faults depends on the fault model used. Although fault models can be arbitrarily general, allowing us to diagnose and fix an arbitrary fault, we typically assume that an expression or a left-hand side of an assignment is incorrect. We do not, however, limit the signals that influence the assignment. Hence, the new expression can contain arbitrary variables and the new left-hand side can be an arbitrary variable.

Applying model based diagnosis to fault localization has been studied for several years. Console, Friedrich, and Theseider Dupre [Console *et al.*, 1993] show the applicability of model based diagnosis to fault localization in logic programs. This approach has been extended to functional programs [Stumptner and Wotawa, 1999], hardware description languages [Friedrich *et al.*, 1996], and object oriented programs [Mateis *et al.*, 2000].

Stumptner and Wotawa [Stumptner and Wotawa, 1996] consider the problem of finding a diagnosis and a repair for a logic program. They find the diagnosis and repair considering only one input sequence, instead of all possible input sequences. This means that the diagnosis and repair found may not be applicable for a different input sequence.

Friedrich, Gottlob and Neidl [Friedrich *et al.*, 1992] give a formalization of the model-based diagnosis and repair process. They use the concept of *logically possible worlds*. These worlds depend on observations and actions from the past. A restricted set of repair operations are performed if particular observations and actions in the system have occurred. The repair operation change the actual world and new repair actions may arise.

Buccafurri et al. [Buccafurri *et al.*, 1999] integrate abductive reasoning into model checking for suggesting repairs.

Their approach is based on calls of a model checker. For every potential repair they invoke the model checker to verify the repair suggestion. Our approach is much more efficient: it uses one traversal of the state space to find and fix the problem.

Wahba and Borrione [Wahba and Borrione, 1995] present a method for finding single faults in sequential circuit. The given specification is another sequential circuit. Their algorithm iterates over time frames and in each step it removes gates from suspicion that would make a correct output incorrect or leave an incorrect output incorrect. Their algorithm has some limitations. They assume a limited type of faults (and/or gate switched, forgotten inverter, etc.) and they assume a given set of test patterns. Our approach use a more general fault model and we provide a correction for all possible inputs.

The remainder of the paper is structured as follows. Section 2 describes the definitions and methods used in our approach. In Section 2.1 the basic definitions for games are given. The construction of the game is described in Section 2.2. How diagnoses and repairs are provided by the game is described in Section 2.3. Characteristics, limitations and the complexity of our approach are presented in Section 2.4. In Section 3 we show the effectiveness of our approach on some examples. We conclude with Section 4.

2 Diagnosis and Repair

2.1 Games

A *game* over atomic propositions AP is a tuple $G = (V, v_0, I, C, \delta, \lambda)$, where V is a set of states, v_0 is the initial state, I is the set of environment inputs, C is the set of system choices, $\delta : V \times I \times C \to V$ is the transition function, and $\lambda : V \to AP$ is the output function. If C is a singleton, the game is a deterministic Moore machine.

The system choices model an incomplete implementation. The main question we ask about a game is what the system choices should be so that the game fulfills its specification. See [Thomas, 1995; 2002] for an overview of the theory of (infinite) games.

For the moment, we will assume that the specification S is an *invariant*, that is, S is a subset of 2^{AP}. The *bad states* B are the states v for which $\lambda(v) \notin S$. A Moore machine satisfies the specification if no bad state is reachable from v_0, for any input sequence.

A *strategy* is a function $\sigma : V \times I \to C$. A *play* of σ on G for the input sequence $i_0, i_1, \ldots$ is the infinite sequence $v_0, v_1, \cdots \in V$ such that $\delta(v_i, i_i, \sigma(v_i, i_1)) = v_{i+1}$. The strategy is winning if there is no input sequence such that the corresponding play visits a bad state. It should be clear that a winning strategy can be used to turn a game into a Moore machine that satisfies the specification.

We define the operator $\mathsf{IX} : V \to V$ as $\mathsf{IX}\, A = \{q \mid \exists i \in I \, \forall c \in C \, \exists q' \in A : \delta(q, i, c) = q'\}$, that is, the set of states from which the environment can force a visit to a state in A in one step. We define the set of lost states L for B as the least fixed point of the function $\lambda A.B \cup \mathsf{IX}\, A$. We should avoid the lost states because from them, the environment can force the system to a bad state in zero or more steps. If $v_0 \notin L$, this is possible, and a winning strategy is given by selecting, for every state $v \notin L$ and every input $i \in I$, a system choice $c \in C$ such that $\delta(v, i, c) \notin L$.

If the specification is not an invariant, but rather a formula in LTL, finding a repair is not as simple. In [Jobstmann *et al.*, 2005] it is shown how to solve the repair problem for LTL, which can be extended to the diagnosis problem in the manner described above for invariants. The approach is not complete for two reasons. First, the authors trade completeness for an exponentially more efficient algorithm. Second, in LTL, we can specify properties such as "The output at time 10 is equal to the input at time 0". If the program does not store the value of the input, we would need to add state to the program. One may argue for or against the desirability of such a repair. If one argues for, the repair is relatively easy. If one argues against, we can use a heuristic to solve the NP-complete problem of deciding whether a positional strategy exists.

The approach of [Jobstmann *et al.*, 2005] is complete for LTL properties that describe invariants. Note that most authors that consider the problem of fault localization through diagnosis limit their attention to invariants [Console *et al.*, 1993; Stumptner and Wotawa, 1996].

2.2 Construction of the Game

The first step in finding a repair for a program which does not fulfill its specification is to obtain a game from the program. A program corresponds to a Moore machine $G = (V, v_0, I, \{c\}, \delta, \lambda)$ where the system choice is a singleton. We will add system choices to this game that represent the alternatives corresponding to different diagnoses and repairs.

To find a winning strategy we have to decide what our system choice is. Suppose that we have a program with a fault and an expression *expr* that is a candidate diagnosis for the failure. We obtain a game by *freeing expr*, i.e., the expression is now the system choice. By freeing the expression we let the game decide a correct value. If we can find a winning strategy, it implies a repair for the program: regardless of the input sequence the specification is always fulfilled. For invariants a winning strategy exists iff a repair exists.

We can go a step further. Instead of manually deciding a candidate diagnosis for the game, we want to let the game decide which expression is the diagnosis. To this end we introduce a variable called `diagnosis` which is set by the game. The value of `diagnosis` corresponds to an expression in the program. If an expression is selected by `diagnosis`, it is freed as described before and determined by the game. All other statements remain the same.

If we want to extend our game in that way, the code has to be instrumented. For example, the assignment `a = b` in the original program becomes

```
if (diagnosis != this expression){
  a=b
} else {
  a=choose(range of a)
},
```

where `choose` is a system choice within the range of `a`. Other types of statements are changed in a similar way.

In order to identify faulty components, we need to decide what the components of the system are. For finite-state programs, our set of components consists of all expressions and the left-hand side of each assignment.

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.

2.3 Playing the game

The value of `diagnosis` is determined by the game as the first step in the instrumented program, even before the inputs for the program are read.

If we can find a winning strategy for the game it implies a repair: the strategy provides an assignment to `diagnosis` which corresponds to the faulty line in the program. Furthermore, the strategy provides an expression for the right-hand side or a variable for the left-hand side in the found line, such that by replacing the original value, the specification is always fulfilled. Because `diagnosis` is chosen by the system, the game is capable to find all single-fault repairs if there exists more than one. Note that the proper value for `free` may depend on the state of the program, which means that the repair is an expression in the state variables, and not necessarily a constant.

This approach is easily extended to multiple faults by choosing more than one diagnosis variable before the program starts.

2.4 Characteristics and Complexity

The approach is guaranteed to work for invariants. For LTL formulas, we may not find a repair, for reasons explained previously. The complexity of the approach is comparable to that of model checking. From a theoretical perspective, the complexity of the approach is exponential in the size of the program and the size of the specification. For single-fault diagnoses, model-based diagnosis using the approach of [Reiter, 1987] is also exponential because of the calls to the underlying theorem prover (a SAT solver). We have not undertaken any practical comparisons.

The symbolic implementation that we use is similar to that of symbolic model checking [McMillan, 1994], using standard algorithms for solving games [Thomas, 1995; 2002]. It is based on BDDs [Bryant, 1986] and typically avoids the exponential blowup by using a set-based approach, computing the validity of all diagnoses and repairs simultaneously.

3 Experiments

We have implemented our approach in VIS-2.1 [Brayton and others, 1996]. The programs in this section are instrumented as described in Section 2.2 and translated to Verilog. Assertions in the Verilog code are written in the form `if(...) error=1` and the invariants are stated as the LTL formula $\mathsf{G}(\text{error} = 0)$. In the current version of the program, the translations and instrumentations are done manually, but this process can easily be automated.

3.1 Minmax Example

Minmax is a simple program which should evaluate the maximum and the minimum of 3 input values [Groce, 2004]. The minimum is stored in `least`, the maximum is stored in `most`. The fault is located in Line 8 in Figure 1. Instead of assigning `input2` to `least` the value is assigned to `most`.

In order to present the instrumentation we start with a fault model which allows the algorithm to set the LHS of assignments. We free the left-hand side of the assignments in Line 4, 6, 8 and 10, i.e. the game decides if the value is assigned to `least` or `most`.

Figure 2 shows the instrumented version of the Minmax program. In Line 0, the game selects for variable `diagnose` one of four lines (represented by the function choose, which result is one of its parameters corresponding to a line in the source: 4, 6, 8 and 10). If a line is chosen, the game can decide the left-hand side for the assignment.

Note that `error` is initialized with `0` and is set to `1` in Line 11 if `!(least <= most)`. For the repair game we use the LTL invariant $\mathsf{G}(\text{error} = 0)$. Our algorithm is able to find the correct repair: it suggests to change the left-hand side of the assignment in Line 8 (Line 8.3 in the instrumented version) from `most` to `least`.

In an extended version of the example we change the fault model. We do not want to restrict the fault model to the LHS of assignments. Thus we additionally free the RHS of assignments and the expressions in the `if` conditions.

The algorithm provides two diagnoses and the corresponding repairs. 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.

In order to show the applicability of our approach to specifications that are not invariants, we change the program and the specification. In a modified program version we initialize `error` with `1` and set it to `0` if `(least <= most)`. We change the specification to the LTL formula $\mathsf{X}(\mathsf{X}(\mathsf{F}(\text{error} = 0)))$. The specification can be interpreted in the following way: "After two steps error must eventually be equal to 0". This is clearly not an invariant. Our algorithm is again able to find the correct repair.

3.2 Locking Example

Figure 3 shows an abstract program which realizes simple lock/unlock operations [Henzinger *et al.*, 2002; Groce, 2004]. Nondeterministic choices in the program are represented by

```
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);
```

Figure 1: MinMax Example

```
0    diagnose = choose{4, 6, 8, 10}
1    int least = input1;
2    int most = input1;
3    if(most < input2)
4      if (diagnose != 4)
4.1      most = input2;
4.2    else
4.3      choose{least, most} = input2;
5    if(most < input3)
6      if (diagnose != 6)
6.1      most = input3;
6.2    else
6.3      choose{least, most} = input3;
7    if(least > input2)
8      if (diagnose != 8)
8.1      most = input2;
8.2    else
8.3      choose{least, most} = input2;
9    if(least > input3)
10     if (diagnose != 10)
10.1     least = input3;
10.2   else
10.3     choose{least, most} = input3;
11   if(least <= most) error = 1;
```

Figure 2: Instrumented MinMax Example

`*`. 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 in the program is located in Line 6, which should be within the scope of the `if` command. The assertion in Line 21, for instance, is violated if the while loop is executed two times without calling `lock()`.

We instrumented the code in Line 1, 3, 4, 6 and 7. The algorithm finds three correction, using 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.

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.

```
  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; }
```

Figure 3: Locking Example

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. This is satisfactory repair because it does not restrict the conditions in the if and while statements.

Note that we are able to find a repair, although we use a different fault model than the one which caused the error.

In [Chen *et al.*, 2005] a model based diagnosis was performed on this program and located three candidates for the fault: Line 1, 6 and 7. Exactly the lines we have found and for which we have provided a repair.

3.3 Sequential Multiplier

The four-bit sequential multiplier shown in Figure 4 is introduced in [Hamscher and Davis, 1984]. 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 register A is the control input for the multiplexer. If it is high, the multiplexer forwards the value of register B to the adder, which adds it to the intermediate result stored in register Q. After four clock cycles register 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

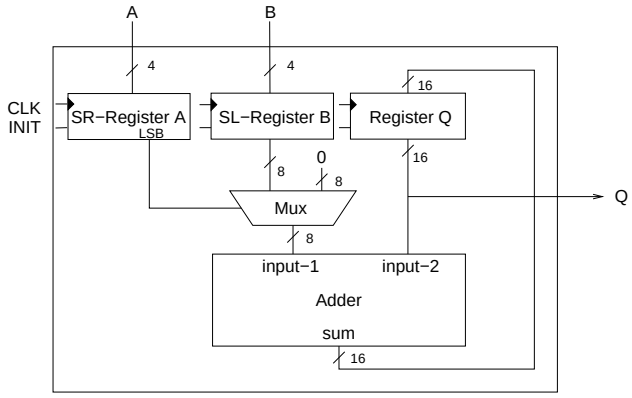

Figure 4: Sequential Multiplier

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. Similarly Q is a diagnosis for every test case.

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.

4 Conclusions

We have shown that the problem of diagnosing a fault in software coincides with the problem of finding a fix. We have shown an approach in which the specification is given as an LTL formula and we look for a diagnosis and a repair. The approach is sound, that is, when we find a repair, it is always valid. For invariants the approach is complete, and we are guaranteed to find a repair when one exists. Invariants are the only sort of specification considered by most other authors that study fault localization by diagnosis. We have shown experimental evidence supporting the applicability of our approach, which has a complexity comparable to that of model checking.

Acknowledgments

This work was supported in part by the European Union's PROSYD project.

References

[Brayton and others, 1996] 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.

[Bryant, 1986] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

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

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

[Console *et al.*, 1993] 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.

[Friedrich *et al.*, 1992] G. Friedrich, G. Gottlob, and W. Nejdl. Formalizing the repair process. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 709–713, Vienna, Austria, 1992.

[Friedrich *et al.*, 1996] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. In *European Conference on Artificial Intelligence*, pages 491–495, 1996.

[Groce, 2004] 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.

[Hamscher and Davis, 1984] 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, Austin, TX, 1984.

[Henzinger *et al.*, 2002] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th symposium on Principles of programming languages (POPL'02)*, pages 58–70. ACM Press, 2002.

[Jobstmann *et al.*, 2005] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. To appear at Computer Aided Verification '05, retrieve from www.ist.tugraz.at/verify/view/Projects/ProgramRepair, 2005.

[Mateis *et al.*, 2000] 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.

[McMillan, 1994] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[Pnueli, 1981] A. Pnueli. The temporal semantics of concurrent programs. *Theoret. Comput. Science*, 13:45–60, 1981.

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

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

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

[Thomas, 1995] 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.

[Thomas, 2002] W. Thomas. Infinite games and verification. In *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 58–64. Springer-Verlag, 2002. LNCS 2404.

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