Automatic Fault Localization for Property Checking[⋆]

Stefan Staber[1], Görschwin Fey[2], Roderick Bloem[1], and Rolf Drechsler[2]

[1] Graz University of Technology, 8010 Graz, Austria
[2] University of Bremen, 28359 Bremen, Germany

Abstract. We present an efficient, fully automatic approach to fault localization for safety properties stated in linear temporal logic. We view the failure as a contradiction between the specification and the actual behavior and look for components that explain this discrepancy. We find these components by solving the satisfiability of a propositional Boolean formula. We show how to construct this formula and how to extend it so that we find exactly those components that can be used to repair the circuit for a given set of counterexamples. Furthermore, we discuss how to efficiently solve the formula by using the proper decision heuristics and simulation based preprocessing. We demonstrate the quality and efficiency of our approach by experimental results.

1 Introduction

When a design does not fulfill its specification, debugging begins. There is little tool support for fault localization and correction, although industrial experience shows that it takes more time and effort than verification does.

In this paper we propose an approach for automatic localization of fault candidates for sequential circuits at the gate or HDL level for safety properties. The diagnosis uses a set of counterexamples that is obtained from either a formal verification tool or a simulator with functional checkers. Our approach builds on model based diagnosis [21]. A failure is seen as a discrepancy between the required and the actual behavior of the system. The diagnosis problem is then to determine those components that explain the discrepancy, when assumed that they are incorrect.

In [12], it is shown that for certain degenerate cases of sequential circuits, model based diagnosis marks all components as possible faults. Perhaps for this reason, there is little work on model-based diagnosis for sequential circuits, with the exception of [19], which does not take properties into account and has a different fault model than we do. Our experimental results show, however, that such degenerate cases rarely happen and that model based diagnosis can be used successfully in the sequential case.

Previous work in both the sequential and combinatorial case has assumed that a failure trace is given and the correct output for the trace is provided by the user. In our approach, instead of requiring a fixed error trace, we only assume that a specification is given in *Linear Temporal Logic* (LTL) [20]. Counterexamples to a specification can be extracted automatically and the user does not need to provide the correct output: the necessary constraints on the outputs are contained in the specification.

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

We formulate the diagnosis problem as a SAT problem. Our construction is closely related to that used in Bounded Model Checking (BMC) [5]. In our setting, a counterexample of length k is given. As in BMC, we unroll the circuit to length k and build a propositional formula to decide whether the LTL property holds. If we fix the inputs in the unrolled circuit to the values given in the counterexample and assert that the property holds, we arrive at a contradiction. The problem of diagnosis is the problem of resolving this contradiction.

To resolve the contradiction, we extend the model of the circuit. We introduce a set of predicates which assert that a component functions incorrectly. If an *abnormal predicate* is asserted, the functional constraints between inputs and outputs of the component are suspended. The diagnosis problem is to find which abnormal predicates need to be asserted in order to resolve the contradiction.

We can further restrict the set of satisfying assignments by requiring that the output of a gate must depend functionally on the inputs and the state of the circuit. Thus, we require the existence of a combinatorial correction. This allows us to extract a suggestion of the proper behavior of the suspect component from the satisfying assignments.

To improve the performance of the algorithm, we have experimented with decision heuristics for the SAT solver. In our setting a small set of decision variables suffices to imply the values of all other variables. Restricting the decision variables to this set leads to a considerable speedup and allows us to handle large and complex designs.

The search space can be further pruned by applying a simulation based preprocessing step. By calculating sensitized paths, the set of candidate error sites is pruned first. Only those components identified as candidates during the preprocessing step have to be considered during SAT based diagnosis.

The paper is structured as follows. In Section 2, we give an overview of related work. Section 3 gives the foundation of our approach and presents how we perform fault localization. The applicability of the approach on the source level is shown in Section 4. Then, Section 5 gives experimental evidence of the usability of our approach and we conclude in Section 6.

2 Related Work

There is a large amount of literature on diagnosis and repair. Most of it is restricted to combinatorial circuits. Also, much of it is limited to *simple faults* such as a forgotten inverter, or an AND gate that should be an OR. Such faults are likely to occur, for example, when a synthesis tool makes a mistake when optimizing the circuit. The work in [25] and [7] on diagnosis on the gate level, for example, combine both limitations.

Wahba and Borrione [26] treat sequential circuits on the gate level, but limit themselves to simple faults. The fault model of [13] is more general, and it addresses sequential circuits, but assumes that the correct outputs are given. Its technical approach is also quite different from ours.

Ali et al. proposed a SAT based diagnosis approach for sequential equivalence checking [3] and debugging combinational hierarchical circuits [2]. But the technique was only applied on the gate level and under the assumption that correct output values for counterexamples are given.

Both [10] and [29] work on the source code level (for hardware and programs, respectively). Both are based on the idea of comparing which parts of the code are exercised by similar correct and incorrect traces.

Only a few approaches have been proposed that are dedicated to fault localization or correction for property checking. In [9] a simulation based approach is presented which is less accurate than ours. Also, they do not consider functional consistency constraints. We use this simulation based technique as a preprocessing step to prune the number of components considered during diagnosis. In [14, 23] a game based approach is presented which locates a fault and provides a new function as a correction for a faulty component. Because it computes a repair, this approach is far less efficient than the one presented here.

3 Diagnosis for Properties

In this section we describe our approach. In 3.1 we give a description of the basic algorithm. We describe extensions of the algorithm for runtime and accuracy improvements in Section 3.2, 3.3, and 3.4. We conclude this section with a discussion in 3.5.

3.1 Computing Fault Candidates

In this section, we describe how to find fault candidates in a sequential circuit. To simplify our explanation, we assume that the components of the circuit are gates, that is, a fault candidate is always a single gate. We will return to the question of the proper definition of components in Section 4. We furthermore assume that the correct specification is given as a (single) LTL formula.

We proceed in four steps:

1. Create counterexamples,
2. build the unrolling of the circuit, taking into account that some components may be incorrect,
3. build a propositional representation of the property, and
4. use a SAT solver to compute the fault candidates.

The counterexamples to the property can be obtained using model checking or using dynamic verification. It is advantageous to have many counterexamples available as this increases the discriminative power of the diagnosis algorithm. Techniques for obtaining multiple counterexamples in model checking have been studied in [8, 11]. We will, however, first focus on the case where one counterexample (of length k) is present. We assume that the counterexamples are finite, that is, we ignore the liveness part of the specification.

The purpose of steps 2 and 3 is to construct a propositional formula ψ such that the fault candidates can easily be extracted from the satisfying assignments for ψ. As explained before, the procedure is closely related to BMC, and we will pay attention specifically to the differences.

Unrolling the Circuit We will assume that the reader knows how a propositional logic formula is obtained by unrolling the circuit. Let n be the number of gates in the circuit (before unrolling) and let $\varphi_{i,t}$ be the propositional representation of the behavior of gate i at time frame t. Then, $\bigwedge_{t\in\{0,\ldots,k-1\}} \bigwedge_{i\in\{0,\ldots,n-1\}} \varphi_{i,t}$ is the (standard) length-k temporal unrolling of the circuit.

In order to perform diagnosis, we introduce n new propositional variables, ab_0 through ab_{n-1}. We replace the description of gate i at time frame t by the formula $\varphi'_{i,t} = (\neg ab_i \rightarrow \varphi_{i,t})$. Intuitively, if ab_i is asserted, gate i may be incorrect, and we do not make any assumptions on its behavior at any time frame. If ab_i is not asserted, the gate works as required. Now assume that we have just one counterexample and we use the formula ξ to represent that the inputs of the unrolled circuit are as prescribed by our counterexample. Then the description of the unrolling is given by

$$\varphi' = \xi \wedge \bigwedge_{t\in\{0,\ldots,k-1\}} \bigwedge_{i\in\{0,\ldots,n-1\}} \varphi'_{i,t} \, .$$

Building the Property Next, we explain how to construct the propositional formula χ representing the specification.

Suppose a partial specification of the system is given in a LTL formula f. For each subformula g of f and for every time frame t we introduce a new propositional variable $v_{g,t}$. These variables are related to each other and to the variables used in the unrolling of the circuit as follows. For the temporal connectives, we use the well-known expansion rules [15], which relate the truth value of a formula to the truth values of its subformulas in the same and the next time frame. For instance, $\mathsf{G}\, f = f \wedge \mathsf{X}\,\mathsf{G}\, f$ and $\mathsf{F}\, f = f \vee \mathsf{X}\,\mathsf{F}\, f$. The Boolean connectives used in LTL are trivially translated to the corresponding constructs relating the propositional variables. Finally, the truth value of atomic proposition p at time frame t is equal to the value of the corresponding variable in the unrolling of the circuit. The final requirement is that the formula is not contradicted by the behavior of the circuit. That is, $v_{f,0}$, the variable corresponding to the specification in time frame 0, is true.

Propositional Formula Note that if we combine the description of the counterexample, the circuit, and the specification and we assume that all abnormal predicates are false, we arrive at a contradiction. Let $\zeta_0 = \bigwedge_{i=0}^{n-1} \neg ab_i$, then $\varphi' \wedge \chi \wedge \zeta_0$ is contradictory.

A diagnosis is obtained by asking which abnormal predicates can resolve the contradiction. For instance, for single fault candidates, let $\zeta_1 = \bigvee_{i=0}^{n-1} \bigwedge_{j\neq i} \neg ab_j$ guarantee that at most one abnormal predicate is true and let $\psi = \varphi' \wedge \chi \wedge \zeta_1$. If a is a satisfying assignment for ψ, and a asserts ab_i, then i is a fault candidate.

Multiple counterexamples can be used to reduce the number of diagnosed components: only an explanation that resolves the conflict for all counterexamples is a fault candidate. The propositional formula corresponding to this problem consists of one unrolling of the circuit for each counterexample. All sets of variables are disjoint, the abnormal predicates, which are shared, are an exception.

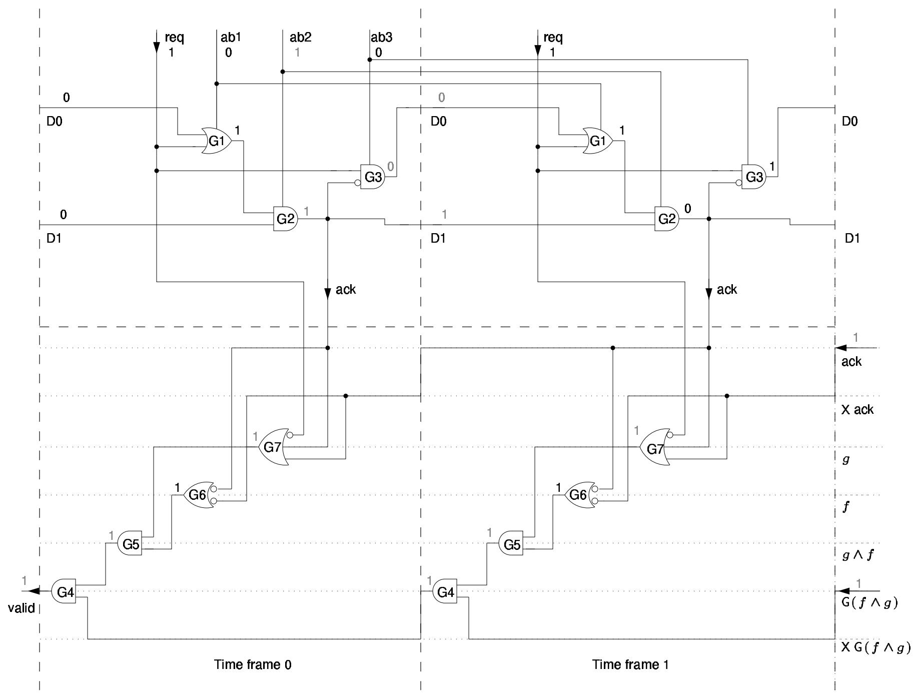

Fig. 1. Circuit with gate G2 as diagnosis ($g = \neg\mathrm{req} \vee \mathrm{ack} \vee \mathsf{X}\,\mathrm{ack}$, $f = \neg\mathrm{ack} \vee \neg\,\mathsf{X}\,\mathrm{ack}$).

Example In the following we illustrate the process using a simple arbiter with input req and output ack. The arbiter is supposed to acknowledge each request either instantaneously or in the next clock tick, but it may not emit two consecutive acknowledgements. In LTL, the specification reads

$$\mathsf{G}((\neg\mathrm{req} \vee \mathrm{ack} \vee \mathsf{X}\,\mathrm{ack}) \wedge (\neg\mathrm{ack} \vee \neg\,\mathsf{X}\,\mathrm{ack})).$$

Let D0 and D1 be latches. Latch D0 remembers whether there is a pending request, and D1 remember whether an acknowledge has occurred in the last step. The arbiter is defined by the following equations: $\mathrm{ack} = (\mathrm{D0} \vee \mathrm{req}) \wedge \mathrm{D1}$, $next(\mathrm{D0}) = \mathrm{req} \wedge \neg\mathrm{ack}$, and $next(\mathrm{D1}) = \mathrm{ack}$. Furthermore, the initial values of D0 and D1 are 0. Note that the circuit contains a fault: ack should be $\mathrm{G1} \wedge \neg\mathrm{D1}$ (see Figure 1).

The shortest counterexamples to the property have length two. For example, if we have requests in the first two time frames, ack is 0 in both frames, which violates the specification.

Figure 1 shows the unrolled circuit combined with the unrolled LTL specification. The abnormal predicates can remove the relation between the input and the output of a gate. For instance, the clauses for gate G2 are equivalent to $\neg\mathrm{ab}_2 \rightarrow (\mathrm{G2} \leftrightarrow (\mathrm{G1} \wedge \mathrm{D1}))$. Nothing is ascertained about the case where ab_2 is true.

The gates below the horizontal dashed line correspond to the unrolled formula. The signal corresponding to the truth of the specification is labeled "valid". For every time frame, the outputs of the gates in the unrolled formula correspond to a subformula of the specification. In the figure, the labels on the dashed horizontal lines indicate which subformula is represented by a gate output.

It is easily seen that valid is zero when two requests occur and all abnormal signals are set to zero. (Please ignore the gray numbers.) Note that signals corresponding to the valuation of ack and $\mathsf{G}\,(f \wedge g)$ in time frame 2 are inputs (bottom right). The fact that the specification is false can be derived regardless of the values of these signals, since the counterexample is finite.

The question we pose the SAT solver is whether there is a consistent assignment to the signals that makes the specification true and sets only one of the abnormal predicates to true. One solution to this question is shown in gray in the figure. Gate G2 is assumed to be incorrect (as expected). For the circuit to be correct, it could return 1 in time frame 0 and 0 in time frame 1. The corresponding value suggested by this satisfying assignment is that G2 should be 0 when G1 is 1 and D1 is 0, and 0 when both inputs to the gate are 1.

The contradiction cannot be explained by setting ab_1 or ab_3 to true, which means that G2 is our only fault candidate.

3.2 Functionality Constraints

There is another satisfying assignment to the example just discussed: let G2 be 0 in the first step and 1 in the second. Note that there is no combinational correction to the circuit that implements this repair, as the inputs and states in both steps would be the same, but the output of G2 is required to be different.

In fact, the approach may find diagnoses for which there is no combinational repair. It may even find diagnoses when the specification is not realizable as a circuit. (A similar observation is made in [28] for multiple test cases). We will now show that by adding Ackermann constraints to our propositional formula we can guarantee that for any diagnosis there is a fix that makes the circuit correct for at least the given set of counterexamples.

Let us say that a gate g is *repairable* if there is a Boolean function $b(i, s)$ in terms of the inputs and the state such the circuit adheres to the specification when g is replaced by $b(i, s)$. That is, a gate is repairable if we can fix the circuit by replacing the gate by some new cone of combinational logic.

We say that g is *repairable with respect to C*, where C is a set of sequences of inputs, if there is a Boolean function $b(i, s)$ such that none of the sequences in C are a counterexample to the property when g is replaced by $b(i, s)$.

Given a set of counterexamples C, the *Ackermann constraint* for a gate g says that for any (not necessarily distinct) pair of counterexamples c_1, c_2 and any pair of time steps i, j, if the state and the inputs of the circuit in time step i of counterexample c_1 equal the state and the inputs in time step j of counterexample c_2, then the output of g is the same in both steps.

Ackermann constraints can easily be added to the propositional formula by adding a number of clauses that is quadratic in the cumulative length of the counterexamples and linear in the number of gates.

We have the following result.

```
1   function staticDecision
2     for $i$ := 1 to $A$.size
3       let $ab$ be the variable $A[i]$;
4       if $ab$ == UNDECIDED then
5         $ab$ := 1;
6         return DECISION_DONE;
7       else if $ab$ == 1 then
8         for $t$ := 0 to $k-1$
9           if $H(ab)[t]$ == UNDECIDED
10            $H(ab)[t]$ := 0;
11            return DECISION_DONE;
12    return SATISFIED;
```

Fig. 2. Pseudocode of the static decision strategy

Theorem 1. *In the presence of Ackermann constraints, given a set of counterexamples C, any gate that is a diagnosis is repairable for C.*

It can be argued that our choice of what constitutes a repairable gate is somewhat arbitrary. Alternative definitions, however, are handled just as easily. For instance, one could require that a fix is a replacement by a single gate with the same inputs. The Ackermann constraints would change correspondingly. On the other extreme, one could allow any realizable function, in which case the Ackermann constraints would require that the output is equal if all the inputs in the past have been equal.

3.3 SAT Techniques

In practice, one wants all fault candidates, not just one. This can be achieved efficiently by adding blocking clauses [17] to the SAT instances stating that the abnormal predicates found thus far must be false. Note that we do not add the full satisfying assignment as a blocking clause, but just the fact that some abnormal predicates must be false, to exclude all other valuations of this assignment.

The efficiency of the SAT solver can be drastically improved using a dedicated decision strategy similar to [24]. By default, the solver performs a backtrack search on all variables in the SAT instance. In our case all variable values can be implied when the abnormal predicates and the output values of gates asserted as abnormal are given, since the inputs of the unrolled circuit are constraint to values given by the counterexample. Therefore, we apply a static decision strategy that decides abnormal predicates first and then proceeds on those gates that are asserted abnormal starting at time frame 0 up to time frame $k-1$.

Figure 2 shows the pseudo code for this decision strategy. The vector A contains all abnormal predicates. This vector is searched until a predicate ab with an undecided value is found. If no value was assigned, the predicate is set to 1 (Lines 4-6). Due to the construction of the SAT instance, this assignment implies the value 0 for all other abnormal predicates. If the first assigned predicate has value 1, the output variable of the gate influenced by ab is considered (Lines 7-11). The hash H maps abnormal predicates to output variables of gates. $H(ab)$ returns a vector of k propositional variables. Variable $H(ab)[t]$ represents the output of the gate that is asserted abnormal by ab at time frame t. Thus, the first gate with unknown output value that is asserted abnormal is set to the value 0. Gates in earlier time frames are considered first. If no unassigned variable is found, a satisfying assignment was found (Line 12). Note that only one value of each variable has to be assigned in the decision strategy because the other value is implied by failure driven assertions [16]. Note also that $H(ab)[t]$ is a list in the general case because we consider multiple counterexamples and components instead of gates, i.e. each abnormal predicate may correspond to multiple gates as explained in Section 4. In our implementation this list is searched for the first gate that is undecided.

The experiments show a significant speed up when this strategy is applied. We have not yet experimented with constraint replication, but this can obviously be used in our setting, especially when multiple counterexamples are present.

3.4 Simulation Based Preprocessing

When all gates or components of a circuit are considered as potential diagnoses the search space is very large. A first obvious method to reduce this search space is a cone-of-influence analysis or the calculation of a static slice. As a result, only those components that drive signals considered in the property are contained in the SAT instance.

Furthermore, we apply a simulation based preprocessing step [25, 9] to further reduce the number of components that have to be considered during diagnosis. Given a counterexample, all values are simulated on the unrolled circuit and the property in a linear time traversal. Then, starting at the output of the property, sensitized paths are traced towards the inputs and state at time frame 0 of the circuit [1]. This relies on the notion of controlling values of inputs for gates that determine the value of the output, e.g. the value 0 (1) is the controlling value for an AND gate (OR gate). First, the output is marked. Then, inputs with controlling values are marked recursively. If no input is controlling all inputs are marked recursively. Only components on a sensitized path are candidates for diagnoses. When using multiple counterexamples only components marked by each counterexample are candidates. Under a single failure assumption this procedure does not change the solution space for diagnosis, because changing a component that is not on a sensitized path cannot change the output value of the property.

The experimental results show that the overhead of this linear time preprocessing step is low. This step can prune the search space and, by this, reduces the overall run time.

3.5 Discussion

Just like multiple counterexamples, stronger specifications reduce the number of diagnoses. When more properties are considered, the constraints on the behavior are tightened. This observation is supported by our experiments.

In practical applications a hint how to repair the faulty behavior at a particular component is useful. The satisfying assignments not only provide diagnoses, but also the values that the faulty components should have. Thus, a correction is determined for the scenarios defined by the counterexamples.

Debugging the property – that might be faulty in practice – is also possible using the same approach. In this case abnormal predicates are associated to components of the property instead of the circuit.

The extension to liveness properties does not seem to be simple. In model checking, the counterexample to a liveness property is "lasso-shaped": after some initial steps, it enters an execution that repeats infinitely often. It is very easy to remove such a counterexample by changing any gate that breaks the loop without violating the safety part of the property. The recent observation that liveness properties can be encoded as safety [4] does not seem to affect this observation as it merely encodes the loop in a different way. Note however, that on an implementation level one probably has bounds on the response time and liveness can thus be eliminated from the specification, at least for the purpose of debugging.

4 Source Level Diagnosis

The previous section describes our approach by means of sequential circuits on the gate level. In this section we show the applicability of the approach on the source level. An expression on the source level may correspond to multiple gates. Therefore a single fault on the source level may correspond to multiple faults on the gate level. To avoid multiple fault diagnosis on the gate level, we can shift the diagnosis process to the source level and do not care about the gate level representation. Another possibility is to keep the information between source level and gate level and use it for the diagnosis process.

We present two principal techniques to calculate diagnoses at the source code level, discuss their advantages and point out the differences between them. Both techniques have been implemented for an evaluation.

4.1 Instrumentation Approach

The instrumentation approach directly includes the abnormal predicates in the source code of the design, this means components and reported diagnoses are parts of the source code.

We modify the design by introducing new primary inputs for abnormal predicates. Then, each component is enclosed by an if-statement that allows to override the value that is internally calculated by an arbitrary value from another new primary input. For example, when we consider the assignment `c = (a && !b)` as a component, we replace it by

```
    c = if(ab) then new_input else (a && !b).
```

In this implementation the mapping between source and gate level is not important. This makes the approach very easy to implement on top of an existing model checker.

The choice of components only depends on the instrumentation of the source code and can be adjusted to meet particular needs. Our choice regards any expression as a component, including right-hand sides of assignments and branching conditions.

We implemented the instrumentation approach on top of the VIS model checker [6]. We used two Perl scripts to instrument the Verilog design and the LTL property. We used the BMC package of VIS to generate counterexamples. As SAT solver for computing the diagnoses, we used zchaff [18] enhanced with the static decision heuristic discussed in Section 3.5. In the current version of the implementation multiple counterexamples need multiple calls to the SAT solver.

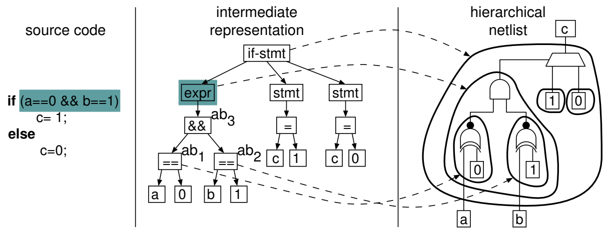

Fig. 3. Source code link in the Hierarchical Approach

4.2 Hierarchical Approach

In the second approach, the hierarchy that is induced by the syntactical structure of the source code is included in the gate level representation of the design and the property. This allows us to link the gate level to the source code.

The link between source code and gate level model is established during synthesis. Figure 3 shows this procedure. An *Abstract Syntax Tree* (AST) is created from the source code at first. Then, the AST is traversed and directly mapped to gate level constructs. During this mapping, the gates that correspond to certain portions of the source code can be identified. Thus, the AST induces regions at the gate level. These regions are grouped hierarchically.

Components are identified based on this representation. Each region corresponds to a component. E.g., the expression (a==1) && (b==0) corresponds to three components: (a==1), (b==0), and the complete expression. We introduce a single abnormal predicate for each region. All gates that do not belong to a lower region in the hierarchy are associated to this abnormal predicate. In the example the predicates ab_1, ab_2, and ab_3 are introduced.

Although this approach requires a modified synthesis tool, the diagnosis engine can take advantage of the hierarchical information. For instance, a correction of a single expression may not be possible but changing an entire module may rectify all counterexamples. When this hierarchy information is encoded in the diagnosis problem, a single fault assumption still returns a valid diagnosis.

The granularity of the diagnosis result can also be influenced. For example, we may choose only source level modules as components to retrieve coarse diagnoses, or, in contrast, we may consider all subexpressions and statements as components for a fine grained diagnosis result.

Finally, hierarchical information can be used to improve the performance of the diagnosis engine [2]. First, a coarse granularity can be used to efficiently identify possibly erroneous parts of the design. Then, the diagnosis can be carried out at a finer granularity with higher computational cost to calculate more accurate diagnoses for the previously diagnosed components.

The implementation of the hierarchical approach uses a modified version of the synthesis tool vl2mv from VIS and an induction-based property checker. The design *and* the property were described in Verilog. As a result, each can be considered during diagnosis. This environment can use multiple counterexamples for diagnosis and simu-

Table 1. Results for weak vs. strong specification (Columns 1,2, and 3: name of the design, number of gates and registers in design; Columns 4 and 5: length of the counterexample and time in seconds to calculate it; Column 6: number of components on source level; Column 7 and 8: results for static slice (percentage numbers are the ratio of the result to the total number of components); Columns 9 and 10: diagnosis results with weak specification; Columns 11 and 12: diagnosis results with strong specification, Column 13 time to solve SAT instance for a single and all diagnoses

Circuit			BMC		Diagnosis							
						slice		weak		strong		
Prop	gates	registers	len	time	#cmp	#cmp	%	#cmp	%	#cmp	%	time
b01_e1, *pOverfl*	98	7	5	0.01	40	32	80	8	20	5	13	0.04, 0.10
b02_e1, *pAltOut*	46	4	5	0.02	20	20	100	6	30	5	25	0.01, 0.02
b03_e1, *pGrantInv*	387	30	4	0.01	50	49	98	10	20	7	14	0.01, 0.14
b09_e1, *pLoadOld*	398	28	21	0.13	33	22	67	14	42	6	18	0.38, 0.83
b10_e1, *pRx2Tx*	318	20	7	0.02	61	53	87	16	26	10	16	0.73, 0.96
b11_e1, *pRsum*	770	31	6	0.17	44	39	89	16	36	9	20	0.11, 0.39
b13_e1, *pRelease*	505	53	5	0.11	96	72	75	6	6	3	3	1.88, 1.94
VsaR_e1, *pInv*	2956	154	15	0.5	56	50	89	14	25	8	14	1.99, 5.87

lation based preprocessing. The property checker is based upon a version of zchaff that supports incremental SAT [27] and is enhanced with the static decision heuristic. During diagnosis, one SAT instance is created that includes a copy of the design for each counterexample. We use the incremental interface of zchaff to calculate all diagnoses.

5 Experimental Results

For the experimental data, we used benchmarks provided with VIS. We manually introduced a bug in each of the designs by changing an operator or a constant. In the following, we will show how specific the diagnosis is and we will show the benefit of the modified decision heuristics and simulation based preprocessing. We are currently using two implementations, one for the instrumentation approach and one for the hierarchical approach. This is the reason that the designs in the two tables are not the same.

5.1 Accuracy

Diagnosis Results and Strong Specification For the analysis of the accuracy of the diagnosis we first consider the results of the instrumentation approach. We used a Pentium IV (Hyperthreading, 2.8 GHz, 3GB, Linux) for the experiments.

Table 1 contains the obtained experimental results. Since a verification engineer would only consider the expressions for debugging that are in the cone of influence of the failing property, we have calculated a static slice. For the diagnosis process we first used a weak specification, namely only the property which failed during bounded model checking. As shown in the table, the diagnosis results with the weak specification are far better than the slicing results. We repeated the experiments with a stronger specification. We added between three and ten additional properties to the property that failed during bounded model checking. With a stronger specification the number of diagnoses were reduced for every example, and for some examples results were significantly better. The small number of diagnoses underlines the usefulness of our approach to fault localization.

Table 2. Diagnosis results for multiple counterexamples and Ackermann constraints

Circuit				Diagnosis								
					slice		single		four		Ackermann	
Prop	gates	registers	len	#cmp	#cmp	%	#cmp	%	#cmp	%	#cmp	%
am2910_p1_e1,*pEntry5*	2257	102	5	227	205	90	66	29	36	15	36	15
am2910_p2_e1, *pStackPointer*	2290	102	5	230	87	37	37	16	26	11	26	11
bpbs_p1_e1, *pValidTransition*	1640	39	2	127	102	80	15	11	13	10	13	10
bpbs_p1_e2, *pValidTransition*	1640	39	2	127	102	80	15	11	4	3	4	3
counter_e1, *pCountValue*	25	7	3	11	10	90	4	36	4	36	1	9
FPMult_e1, *pLegalOperands*	973	69	4	119	105	88	3	2	3	2	3	2
FPMult_e2, *pLegalOperands*	973	69	4	119	105	88	54	45	47	39	47	39
gcd_e1, *pReadyIn22Cyc*	634	51	22	87	68	78	45	51	35	40	35	40
gcd_e2, *pReadyIn22Cyc*	634	51	22	87	68	78	34	39	32	36	32	36
gcd_e1, *pBoth*	634	51	23	87	71	81	46	52	36	41	36	41
gcd_e2, *pBoth*	634	51	23	87	71	81	33	37	33	37	33	37
gcd_e1, *pThree*	634	51	23	87	71	81	33	37	23	26	23	26
gcd_e2, *pThree*	634	51	23	87	71	81	39	44	22	25	22	25

Case Study This example shows the difference in accuracy between two specifications for example b09.

The original functionality of example b09 is a serial to serial converter. As a fault, we negated the condition of an if-statement. The resulting circuit violates the property that describes that in a certain state an input register must be zero. When we perform diagnosis using only the failing property, 14 components are identified.

The converter has four states: INIT, RECEIVE, EXECUTE, and LOAD_OLD. There are specific transitions that are possible between the states, for example from the INIT mode we must only reach the RECEIVE mode. If the permitted transitions between the states are included in the specification, the number of diagnoses is only six.

For the diagnosis corresponding to the actual fault we can conclude out of the new value that we have to invert the if-condition. One diagnosis is located in the branch of the if assignment that is executed because of the faulty if-condition. The suggested value for the input register is zero, as it is required in the property. The property that failed is an implication. In four of the six remaining diagnoses, the new values for the suspended components set the antecedent of the implication to false and therefore the property is satisfied.

Multiple Counterexamples and Ackermann Constraints Table 2 shows the influence of multiple counterexamples and Ackermann constraints on the diagnosis results. The implementation of these features demands full access to the generation of the SAT instance. We therefore integrated them in the hierarchical environment as explained in Section 4.2. Experiments were carried out on an AMD Athlon 3500+ (Linux, 2.2GHz, 1 GB, Linux).

The columns provide the same data as the previous table. Besides diagnosis results for static slicing, we report results for using a *single* counterexample, *four* counterexamples and four counterexamples together with *Ackermann* constraints. The use of multiple counterexamples can significantly improve the diagnosis result. In all but two cases the number of diagnoses was reduced.

Table 3. Run times for the different approaches (using four counterexamples)

	BMC	Diagnosis								
		zchaff default			static			simulation+static		
Circuit, *Property*	time	time	#cmp	#dec	time	#cmp	#dec	time	#cmp	#dec
am2910_p1_e1, *pEntry5*	0.54	11.87	205	165,247	2.63	205	8,047	1.62	69	7,855
am2910_p2_e1, *pStackPointer*	0.01	0.40	87	3,848	0.31	87	989	0.28	52	916
bpbs_p1_e1, *pValidTransition*	0.06	0.19	102	2,819	0.20	102	302	0.13	19	266
bpbs_p1_e2, *pValidTransition*	0.03	0.16	102	1,805	0.14	102	110	0.11	5	87
counter_e1, *pCountValue*	<0.01	0.01	10	259	0.01	10	131	<0.01	9	130
FPMult_e1, *pLegalOperands*	0.04	0.41	105	397	0.19	105	60	0.15	5	60
FPMult_e2, *pLegalOperands*	0.04	2.27	105	17,540	1.14	105	8,440	0.95	76	7,320
gcd_e1, *pReadyIn22Cyc*	18.7	1057.21	68	3,271, 957	53.98	68	479,526	54.35	67	479,525
gcd_e2, *pReadyIn22Cyc*	22.07	351.16	68	1,022,573	19.65	68	115,519	18.59	63	112,833
gcd_e1, *pBoth*	32.24	2213.35	71	3,468,162	91.74	71	425,438	90.08	67	425,436
gcd_e2, *pBoth*	24.20	453.83	71	1,058,165	55.23	71	237,104	50.19	59	232,334
gcd_e1, *pThree*	42.74	1626.07	71	2,617,354	201.76	71	723,180	198.44	65	730,191
gcd_e2, *pThree*	35.50	498.99	71	1,278,064	1306.90	71	3,586,181	1307.80	71	3,586,181

In contrast, Ackermann constraints do not yield the same improvement. Only in one case the number of diagnoses was reduced and the algorithm returned exactly the real error site. The overhead in runtime is quite high for Ackermann constraints. We observed an increase by up to a factor of 60 especially on large instances. Thus, Ackermann constraints should only be applied in a second stage of the diagnosis process due to their low influence on the accuracy.

5.2 Runtime

In Section 3 we suggested two techniques to improve the runtime of the overall algorithm: a static decision strategy for the SAT solver and the use of a simulation based preprocessing step. Both techniques were implemented within the hierarchical framework. Due to page limitation, we only report experimental results for the use of four counterexamples without Ackermann constraints in Table 3. The table shows runtimes for the different approaches. Additionally, the number of components considered during SAT based diagnosis (this is not the number of components returned as diagnoses that is shown in Table 2) and the number of decisions made by the SAT solver are reported.

The runtime decreases drastically when the static decision heuristic is applied. This is due to the reduction of the number of decisions that have to be done by the SAT solver. The only exception is the last benchmark, but when using only one counterexample the runtime was only 9.91 seconds at the cost of a lower accuracy (see above). Usually, the runtime does not exceed the time for BMC much — even when four counterexamples are applied for diagnosis. Here, incrementally applying more and more counterexamples as suggested in [22] can yield an even shorter runtime. The use of the simulation based preprocessing step also saves some runtime in those cases were the number of components considered during SAT based diagnosis can be reduced significantly. On the other hand the overhead is quite low when no components can be pruned.

The creation of counterexamples dedicated for diagnosis may improve the diagnosis results. This hypothesis is strengthened by following experimental results. We did 1000 diagnosis runs with 4 randomly chosen counterexamples on am2910_e1 for the property *pEntry5*. The number of diagnoses varied from 28 to 90 and the runtime varied between 1.75 and 3.75 seconds. Usually, a better diagnosis accuracy also had a shorter runtime.

In summary, the runtime was reduced drastically by the proposed techniques and makes the effort of diagnosis comparable to that of BMC.

6 Conclusions

We have presented an approach to automatically locate design faults at the gate level or the source code level. The approach handles safety properties written in LTL. A propositional logic formula is built in such a way that diagnoses can be derived from its satisfying assignments. We have shown how to extend the formula to make sure that a diagnosed component is actually repairable for the given input sequences.

We have proposed two techniques to implement the approach. One is easy to implement on top of an existing model checker, the other allows the diagnosis engine to exploit hierarchical information. We have shown that the use of multiple counterexamples and more comprehensive specifications provides a more accurate diagnosis result. We have drastically improved the efficiency of the approach by using a dedicated search strategy for the SAT solver and shown its applicability with experimental results.

Some ideas for future work have been discussed already. Furthermore, we would like to further investigate in how far the techniques presented here can be used to find faults in the specification rather than the system. Finally, we would like to attempt to use these ideas on models of a C program, and we would like to try the approach for all possible counterexamples, thus making it complete, by using quantified Boolean formulas.

References

1. M. Abramovici, P.R. Menon, and D.T. Miller. Critical path tracing - an alternative to fault simulation. In *Design Automation Conf.*, pages 214–220, 1983.
2. M.F. Ali, S. Safarpour, A. Veneris, M.S. Abadir, and R. Drechsler. Post-verification debugging of hierarchical designs. In *Int'l Conf. on CAD*, pages 871–876, 2005.
3. M.F. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir. Debugging sequential circuits using Boolean satisfiability. In *Int'l Conf. on CAD*, pages 204–209, 2004.
4. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2), July 2002. Formal Methods for Industrial Critical Systems (FMICS'02).
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.
6. 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.
7. P.-Y. Chung, Y.-M. Wang, and I. N. Hajj. Diagnosis and correction of logic design errors in digital circuits. In *Design Automation Conference (DAC'03)*, pages 503–508, 2003.
8. G. Fey and R. Drechsler. Finding good counter-examples to aid design verification. In *MEMOCODE*, pages 51–52, 2003.

9. G. Fey and R. Drechsler. Efficient hierarchical system debugging for property checking. In *Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'05)*, pages 41–46, 2005.
10. 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.
11. A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with `explain`. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 453–456. Springer-Verlag, Berlin, July 2004. LNCS 3114.
12. 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.
13. S.-Y. Huang and K.-T. Cheng. Errortracer: Design error diagnosis based on fault simulation techniques. *IEEE Trans. on CAD*, 18(9):1341–1352, 1999.
14. 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.
15. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems *Specification**. Springer-Verlag, 1991.
16. J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.
17. K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.
18. M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.
19. B. Peischl and F. Wotawa. Modeling state in software debugging of VHDL-RTL designs — a model based diagnosis approach. In *Automated and Algorithmic Debugging (AADEBUG'03)*, pages 197–210, 2003.
20. A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.
21. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.
22. A. Smith, A. Veneris, and A. Viglas. Design diagnosis using Boolean satisfiability. In *ASP Design Automation Conf.*, pages 218–223, 2004.
23. S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In D. Borrione and W. Paul, editors, *13th Conference on Correct Hardware Design and Verification Methods (CHARME '05)*, pages 35–49. Springer-Verlag, 2005. LNCS 3725.
24. O. Strichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 24(1):5–24, January 2004.
25. A. Veneris and I. N. Hajj. Design error diagnosis and correction via test vector simulation. *IEEE Trans. on CAD*, 18(12):1803–1816, 1999.
26. 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.
27. J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference*, pages 542–545, Las Vegas, NV, June 2001.
28. F. Wotawa. Debugging hardware designs using a value-based model. *Applied Intelligence*, 16:71–92, 2002.
29. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.