# Model-Based Reasoning with Multiple Test Cases and its Application to Debugging

**Bernhard Peischl**[1,2] **Naveed Riaz** [1] **Franz Wotawa**[1,2]
Technische Universität Graz
Institute for Software Technology
{peischl, nriaz, wotawa}@ist.tugraz.at

## Abstract

Today's simulation-centric hardware development process requires to leverage quality test suites for fault localization rather than employing them solely for detecting malfunctioning. In this article we (1) propose an extension of the model-based debugging theory to address the treatment of test suites in a well-founded way and (2) relate this novel approach to an algorithmic technique known as filtering. For multiple test cases revealing a certain fault (3) we propose an iterative computation of diagnoses as an extension to Reiter's diagnosis algorithm, and (4) notably report on empirical evaluations of this algorithm taking into account even dual-fault diagnosis.

## 1 Introduction

Today's increasing demand on semiconductors and software-enabled systems is accompanied by increasing design complexity, high quality attributes, cost pressure and shrinking time to market. Thus detecting, localizing, and fixing faults is a crucial issue in today's fast paced and error prone development process. In general, detecting and repairing misbehavior in an early stage of the development cycle reduces costs as well as development time considerably.

Hardware description languages like VHDL [Navabi, 1993; IEEE, 1988] and Verilog [IEEE, 1995], and the availability of mature simulation technology allow for verification and systematic testing in early stage in the development cycle. Unlike to the early 90s, the need for physical prototypes is more and more replaced by appropriate models of the chip under development. Once a fault is detected, the presence of a physical prototype suggests to conduct supplementing measurements for gaining further observations to locate the misbehavior's root cause. In this scenario, typically the input remains the same, and the engineer looks for further measurement points. The classical literature on model-based diagnosis (MBD) [Reiter, 1987; de Kleer and Williams, 1987] addresses this process and provides foundational support for locating faulty components by incorporating additional measurements.

With the advent of various models in today's development process, particularly exhaustive simulation of digital semiconductors has become an inherent and well-established technique. Model-based test pattern generation furthermore allows for straightforward generation of test suites fulfilling various coverage goals. However, the availability of these quality test suites to our best knowledge is not addressed by MBSD (model-based software debugging) theory so far. In contrast to the scenario in the early 90s, we need to take advantage of numerous individual test cases rather than capturing additional observations obtained from the prototype. Extending classical model-based debugging theory thus might provide the theoretical underpinning to master the fault location process (which accounts for 50 to 80 percent of the time used for verification depending on the level of automation of the employed verification tools [Auerbach *et al.*, 2005]) and promises to provide remedy in terms of sophisticated tool support.

In this article we briefly introduce MBSD and (1) propose to extend MBSD to leverage whole test suites rather than individual tests (see Section 3). Moreover, (2) we relate this approach to the previously proposed method of filtering (also see Section 3) and (3) present a novel extension to Reiter's hitting set algorithm that efficiently takes account of multiple error-revealing test cases (see Section 4). We (4) outline empirical results (notably single- as well as dual-fault diagnosis) obtained from the ISCAS'89 benchmark suite (see Section 5). Finally we discuss related literature (see Section 6), and conclude this article (see Section 7).

## 2 Model-based Software Debugging

The basic idea of model-based software debugging (MBSD) is to employ a single test case together with knowledge about the program's syntax and semantics to locate the misbehavior's possible causes. Obviously, a correctly functioning program cannot produce an incorrect value for a given test case. Therefore, to make the program consistent with this specific test case revealing the faulty behavior, we have to assume

[1] Authors are listed in alphabetical order.

[2] The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsfrderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

some subset of the program's components to work incorrectly. We report these components as diagnosis candidates for the specific test case under consideration.

MBSD, as the discipline of applying model-based diagnosis (MBD) [Reiter, 1987; de Kleer and Williams, 1987] to locate bugs in software, has a well-founded theory and several case studies indicate its maturity in the context of HDLs [Peischl and Wotawa, 2006; Wotawa, 2002]. In the following we restate the definition of the classical diagnosis problem [Reiter, 1987; de Kleer and Williams, 1987] and point out the most notable differences to MBSD.

**Definition 2.1 (Diagnosis Problem)** *[Reiter, 1987; de Kleer and Williams, 1987] A diagnosis system is a pair $(SD, COMP)$, where*

- *$SD$, the system description, in [Reiter, 1987] is a set of first-order sentences*
- *$COMP$, the system components, is a finite set of constants.*

*An observation of a system is a finite set of first-order sentences, and together with the system description $(SD, COMP)$ forms the classical diagnosis problem $(SD, COMP, OBS)$.*

In contrast to a diagnosis problem, where $SD$ specifies the correct behavior and the observations specify the input and the response of the actual system, the system description $SD$ of a debugging problem describes the (faulty) behavior and the tests (partially) specify the intended (correct) behavior. Moreover, to overcome scalability issues, we employ a Horn-clause like encoding for both, $SD$ and the test cases.

**Definition 2.2 (Debugging Problem)** *A debugging problem is a tuple $(SD, COMP, TC)$ where $SD$ is a logical model of the given program (typically incorporating structure and behavior), $COMP$ is a finite set of statements or expressions of this program, and $TC$ refers to a logical sentence representing the given test case.*

## 3 Test Suites for Fault Localization

Although some research work considers multiple test cases (or test suites) for enhancing the proposed model's discrimination capability in terms of filtering the diagnosis obtained from a specific model [Wotawa, 2002], we notably lack a general approach for the treatment of whole test suites. In the following we extend the MBSD framework towards the universal treatment of test suites. Within this framework we establish a relationship to the filtering approach proposed in [Wotawa, 2002].

Rather than considering a single test case solely, a test suite contains multiple test cases $TC_1$, $TC_2$, ..., $TC_n$, while the system description remains the same. As outlined in Section 1, test cases may either reveal a fault - that is, the specific test case's logical encoding is inconsistent with the system description, or fail to detect misbehavior. For the remainder of this article, the first case refers to a negative test case, whereas we refer to a positive test in case of consistency. Thus we can partition the set of test cases $TC$ in positive (referred to as $TC_{pos}$ in the following) and negative ones (formally referred to as $TC_{neg}$).

We continue with a simple example illustrating the potential of positive test cases.

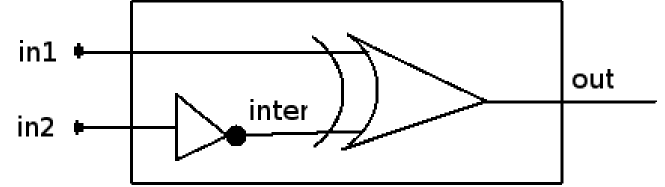



Figure 1: Depicting a small faulty circuit

**Example 3.1 (Positive test cases)** *Figure 1 illustrates a part of a circuit comprising an $XOR$ and a $NOT$ gate. We further assume that this circuit is faulty and that both components are reported as diagnosis candidates by employing negative test cases. Suppose we have already received the following information after the application of a negative test case ($in1$ = '1', $in2$ = '0', $out$ = '1'). Considering the $NOT$ gate abnormal and $XOR$ gate correct, the value of 'inter' is computed '0' by the model. Now we apply a positive test case ($in1$ = '0', $in2$ = '0', $out$ = '1'). Once again considering the $NOT$ gate abnormal and $XOR$ gate correct, the value of 'inter' is now computed '1' by the model. We immediately see that abnormal component $NOT$ is required to map 'inter' = '0' for negative test case and 'inter' = '1' for the positive test case for the same input value $in2$ = '0'. Obviously, no deterministic component can fulfill this requirement. Thus the not component can no longer be considered as a valid diagnosis candidate.*

In the following we generalize the idea motivated by our example and show how to incorporate positive test cases into the MBSD framework. Afterwards we relate our novel system description to the algorithmic filtering approach presented in [Wotawa, 2002].

**Definition 3.1 (Test Suite Integration)** *Given a set of test cases $TC = TC_1, TC_2, ..., TC_n$, a system description $SD$ and a set of components, the diagnosis problem considering all test cases in $TC$ is obtained as follows:*

- *for each $TC_i \in TC do$*
  - *generate a new $SD_i$, where all component and connections are uniquely identified with a new index $i$*
- *let $SD^*$ be $\bigcup_{i=1}^{k} SD_i \cup \{\neg AB(C) \rightarrow \neg AB(C_1) \wedge \neg AB(C_2) \wedge ... \wedge \neg AB(C_k) | C \in COMP\}$, where $C_j$ denotes the corresponding components in $SD_i$.*
- *let $TC^*$ be a renaming of $TC$, such that every test case is associated with connections in $SD_i$*

*The new diagnosis problem incorporating the test cases $TC$ is given by the tuple $(SD^*, COMP, TC^*)$.*

Note that the size of $SD$ increases with the number of test cases linearly and the individual components $C_i, i = 1, 2, .., k$ are treated as independent components. Applying Reiter's algorithm [Reiter, 1987] in a straightforward way

is thus rather inefficient. In Section 4 we thus propose a novel, iterative construction of the hitting-set DAG. However, although this novel algorithm handles additional conflicts rather efficient, positive test cases do not yield to further conflicts and thus are not able to discriminate diagnosis further on. Under absence of structural faults, the following, novel extension allows even for taking advantage of positive test cases by relying on Ackermann constraints [Ackermann, 1954].

As positive test cases do not yield to additional conflicts, we capture their specific information on diagnoses in terms of the system description with Ackermann constraints $SD^A$. To our best knowledge the authors of [Raiman *et al.*, 1991] were the first employing Ackermann constraints to express that our components behave deterministically. By adding these consistency constraints we formalize the fact that the same combination of input values applied to a component $C$ produces the same output for every instance $C_i$. This specifically allows for exploiting valuable information captured in terms of the many test cases not revealing any faulty behavior.

**Definition 3.2 (System Descr. with Ackermann constraints)** *Given a set of positive test cases $TC_{pos}$, we assume $in(C_i) = \{i^1_{c_i}, ..., i^m_{c_i}\}$ to denote the inputs of component $C_i$, and $out(C_i) = \{o^1_{c_1}, ..., o^n_{c_i}\}$ the outputs of component $C_i$. By extending the system description $SD^*$ in terms of the Ackermann Constraints $\neg AB(C) \wedge CON_A = \{\forall^m_{l=1} i^l_{c_i} = i^l_{c_j}) \rightarrow \forall^n_{p=1}(o^p_{c_i} = o^p_{c_j})|i \neq j\}$, where i and j denote indices of test cases, we obtain a diagnosis problem incorporating Ackermann constraints. The diagnosis problem is thus given in terms of the tuple $(SD^A, COMP, TC^*_{pos})$ where $SD^A = SD^* \cup CON_A$ denotes the Ackermann constraint system description.*

An algorithmic approach to these constraints is filtering [Wotawa, 2002]. Filtering refers to discarding certain diagnosis by taking advantage of further test cases $TC_i$ in a dedicated post processing phase. A diagnoses $\Delta$ states that $\Delta \cup SD \cup TC_i \cup \{\neg AB(C)|C \in COMP \setminus \Delta\}$ is consistent. This implies that there is a replacement, that is - there exists a function $replace(C)$ for every component $C \in \Delta$ - repairing the program for the given test case $TC_i$. The function $replace(C)$ allows for producing the correct output values for the considered test case $TC_i$. However, considering a set of test cases $TC = \{TC_1, TC_2, ..., TC_n\}$ such an replacement does not exist for all test cases in $TC$ necessarily.

We briefly restate filtering as follows. Since all components in $COMP \setminus \Delta$ are assumed to behave correctly, we can compute the input values $in(C)$ and $out(C)$ for every component $C \in \Delta$ (Using simulation). According to this computed input-output relation obtained from all test cases $TC$, component $C$ is possibly required to map the same input to different output values. This corresponds to an inconsistency and the specific diagnosis $AB(C)$ is not repairable w.r.t the test cases $TC$: As there is no function $replace(C)$ as stated previously component $C$ can be removed from the set of diagnosis candidates. In this vein, we basically evaluate the Ackermann constraints in an iterative fashion at a meta-model level by checking for different input values for a certain output value.

We formalize the procedure mentioned previously in terms of the procedure filter($\Delta$,$TC$), where $\Delta$ denotes the set of diagnosis candidates and $TC$ represents the test suite:

1. forall test cases $TC_i \in TC$ do
2. (a) forall $D \in \Delta$ do
   (b) Let $i_{d_i}$ denote the values at the input and $o_{d_i}$ be the values at the output of component $D$ obtained by assuming $AB(D) \wedge \{\neg AB(C)|C \in COMP \setminus D\}$
   (c) if there exist indices $i, j$, $i \neq j$, such that $i_{d_i} = i_{d_j} \wedge o_{d_i} \neq o_{d_j}$
   (d) remove $D$ from $\Delta$
3. return $\Delta$

**Claim 1 (Claim 1)** *The procedure $filter(\Delta, TC)$ reduces the diagnosis candidates $\Delta$ obtained from $(SD, COMP, TC)$ exactly to those given in terms of the diagnosis problem $(SD^A, COMP, TC^*)$, where $SD^A$ again denotes the system description $SD$ with Ackermann constraints as given in Definition 3.2 and $TC^*$ refers to the renaming as given in Definition 3.1.*

**Proof 1 (Proof of Claim 1)** *After applying $filter(\Delta, TC)$ to the obtained diagnoses, there is no component $D$ at which we obtain different input values for a certain output. Following the notion given in Definition 3.2, we formally conclude*

1. $\nexists i, j, i \neq j \cdot (\forall^m_{l=1} i^l_{d_i} = i^l_{d_j}) \wedge (\forall^n_{p=1} o^p_{d_i} \neq o^p_{d_j})$
2. $\neg \exists i, j, i \neq j \cdot (\forall^m_{l=1} i^l_{d_i} = i^l_{d_j}) \wedge (\forall^n_{p=1} o^p_{d_i} \neq o^p_{d_j})$
3. $\forall i, j, i \neq j \cdot \neg(\forall^m_{l=1} i^l_{d_i} = i^l_{d_j}) \vee (\forall^n_{p=1} o^p_{d_i} = o^p_{d_j})$
4. $\forall i, j, i \neq j \cdot \neg(\forall^m_{l=1} i^l_{d_i} = i^l_{d_j}) \Rightarrow (\forall^n_{p=1} o^p_{d_i} = o^p_{d_j})$
5. $\forall i, j \cdot CON_A$

□

The procedure $filter(\Delta, TC)$ thus imposes the Ackermann constraints on the set $\Delta$.

# 4 Negative Test Cases: Iterative Computation of Diagnosis

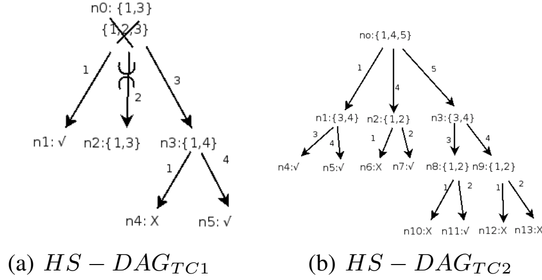
For negative test cases we rely on reusing Reiter's well known Minimal Hitting Set Tree Algorithm [Reiter, 1987] for computing diagnosis. This algorithm was later revised by Greiner [Greiner *et al.*, 1989]. First we generate a Minimal Hitting Set Directed Acyclic Graph (referred to as HS-DAG in the following) for the first test case (similar to actual Greiner's algorithm for single test case). The nodes labeled by $\surd$ represent the minimal diagnosis for current test case. Further test cases traverse, reuse and extend the same directed acyclic graph.

This directed acyclic graph is traversed using breadth first strategy and further test cases modify this acyclic directed graph by adding new nodes or closing, pruning or reusing old nodes. The main idea is that, first an acyclic directed graph $D$ is created, the nodes which represent minimal diagnosis and are labeled by $\surd$ are replaced with conflict sets $CS$ returned from next test cases, if such conflict sets exist . If no such

conflict set exists then this node also represents the minimal diagnosis for next test cases as well.H($\surd$) is the set of edge labels on the path from root down to node labeled by $\surd$.

We use a simple example to illustrate our approach. Suppose we have two test cases, $TC1$ and $TC2$, $F1 = \{\{1,2,3\},\{1,3\},\{1,4\}\}$ and $F2 = \{\{1,4,5\},\{3,4\},\{1,2\}\}$ denote the corresponding conflict sets.

Figure 2 represents their corresponding minimal HS-DAG.




(a) $HS-DAG_{TC1}$ (b) $HS-DAG_{TC2}$

Figure 2: MHS directed acyclic graphs for TC1 and TC2

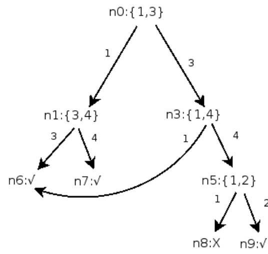



Figure 3: $DAG_{mult}$, the DAG generated by our novel alg.

Suppose $D1 = \{\{1\},\{3,4\}\}$ and $D2 = \{\{1,3\},\{1,4\},\{2,4\},\{2,3,5\}\}$ are diagnoses returned from acyclic directed graphs obtained from $F1$ and $F2$ respectively.

By following the well-known procedure proposed in [Greiner *et al.*, 1989] we obtain a DAG $DAG_{TC_1}$ as depicted in Figure 2. For incorporating $F2$, we start traversing this DAG in breadth first order. We find node n1, marked with $\surd$ at level 1 with H(n1) = $\{1\}$. We can see that this node can be labeled with conflict set $\{3,4\}$ of $F2$, so we replace the label $\surd$ of n1 with $\{3,4\}$ and create two downwards arcs with labels 3 and 4 respectively.

Now we have created two new unlabeled nodes n6 and n7 with H(n6) =$\{1,3\}$ and H(n7)= $\{1,4\}$. In the same level there is no other node which can be re-processed so we move to next level.

We have unlabeled nodes n6 and n7. As there is no conflict set available in $F2$ which can become their label, these are marked with label $\surd$. Moving forward in the same level we find node n4 marked as closed with H(n4) = $\{1,3\}$. This node cannot remain closed any longer because the previous minimal node n1 with H(n1) = $\{1\}$ has been replaced with two new minimal nodes n6 and n7 and H(n6) is also $\{1,3\}$. The new minimal node n6 is at the same level as n4, so we can reuse n6.

Moving forward we find node n5 previously marked with label $\surd$ and H(n5) = $\{3,4\}$. Now this label can be replaced with conflict set $\{1,2\}$ of $F2$, so we replace the label $\surd$ of n5 with $\{1,2\}$. On the next level we have two unlabeled nodes represented as n8 and n9 and H(n8) = $\{1,3,4\}$ and H(n9) = $\{2,3,4\}$. As we already have minimal node n6 with H(n6) = $\{1,3\}$ so we can close the first node. For the second node, as there is no conflict set available in $F2$ which can become it's label, we mark it with label $\surd$. Our final DAG is depicted in Figure 3. We get the following diagnosis from these two test cases $D_{mult}$ = $\{1,3\}$,$\{1,4\}$,$\{2,3,4\}$.

## 4.1 Algorithm

To efficiently treat multiple test cases, we follow Greiner's original algorithm [Greiner *et al.*, 1989] constructing the HS-DAG for an ordered collection of sets $F_i$. We assume to use the same pruning rules, i.e., node closing, node re-use, and node pruning. The latter is used in cases where a conflict set is found which is a subset of a conflict set used earlier in the construction of the HS-DAG. All pruning rules except node closing remain the same in our variant of Reiter's HS-DAG algorithm.

Node closing is used in cases where a node $n$ is processed and there exists another node $m$ labeled with $\surd$ and $H(m) \subset H(n)$. In this case $m$ already reveals a minimal hitting-set and $n$ would only lead to a non-minimal one. Thus $m$ can be closed which is indicated by labeling $m$ with $\times$. When changing a HS-DAG incrementally using multiple test cases a previously generated minimal hitting set might become invalid because a new previously not considered conflict is detected. Hence, the corresponding node $n$ has to be extended and its label has to change. This of course has consequences for nodes $m$ which has been closed because of $n$. In order keep track of closed nodes $m$, we introduce a new function $closed$ for nodes $n$ which stores all nodes that are closed because of $n$. Hence, when changing the label of $n$ from $\surd$ to a conflict set, we immediately know the closed nodes we have to re-process.

The following iterative version of Reiter's HS-DAG algorithm assumes that we have given a set $F$ which comprises set of conflicts $F_i$, $i = 1,..,n$. For each test case $TC_i$ the set $F_i$ stores the conflicts obtained from $TC_i$ and the system description. Note that it is not necessary to compute all conflicts for given test case in advance. They can be computed whenever required. We use the same technique described by Reiter [Reiter, 1987] for this purpose.

1. Construct the root node $n_0$ of $D$. Let $H(n_0)$ be the empty set and label the node with $\surd$.
2. For each element $F_i$ from $F$ do the following:
   (a) Traverse $D$ in breadth first order starting from the root node $n_0$.
   (b) if node $m$ is previously labeled by $\surd$
      i. If for all x $\in F_i$, x $\cap$ $H(m) \neq \{\}$ then leave this node as it is and move to next node.
      ii. Else label m by $\sum$ where $\sum$ is the first member of $F_i$ for which $\sum \cap H(m)$ = $\{\}$ and for

each $\sigma \in \sum$, generate a new downward arc labeled by $\sigma$. This arc leads to a new node $o$ with $H(o) = H(m) \cup \{\sigma\}$. The new node $o$ will be processed (labeled and expanded) after all nodes in the same generation as m have been processed. Re-open all nodes $n \in closed(m)$ by removing their label.

(c) If a node $m$ is unprocessed, i.e., its label is empty, do the following:

i. If for all $x \in F_i : x \cap H(m) \neq \{\}$ then label $m$ by $\surd$

ii. Else, label $m$ by $\sum$ where $\sum$ is the first member of $F_i$ for which $\sum \cap H(m) = \{\}$. For each $\sigma \in \sum$, generate a new downward arc labeled by $\sigma$. This arc leads to a new node $o$ with $H(o) = H(m) \cup \{\sigma\}$. The new node $o$ will be processed (labeled and expanded) after all nodes in the same generation as m have been processed

3. Return D.

After all test cases in $F$ have been processed the HS-DAG $D$ comprises all minimal hitting sets which are the nodes labeled by $\surd$. The pruning rules of Greiner et al. are used in order to ensure $D$ to be as small as possible. Moreover, because of the breadth first computation the algorithm computes hitting sets of increasing size. We might stop computation for hitting sets exceeding a given limit.

## 5 Empirical Results

In this section we evaluate our approach using circuits from ISCAS 89 benchmark suite [Brglez *et al.*, 1989]. For every circuit we introduced a fault by substituting a randomly selected statement with another one, e.g., changing an *and* operator with an *or* operator. We obtained the error revealing-inputs - the negative test cases - by performing a circuit equivalence check with the model checker VIS [R. K. Brayton *et al.*, 1996]. Whenever we invoke VIS, we obtain a different negative test case for the specific circuit. To supplement our results on the filtering approach [Wotawa, 2002], we focus our empirical research work on negative test cases.

Table 1 outlines the obtained results for the first 8 circuits in the ISCAS 89 benchmark: In column one the table lists the circuit name, column two shows the length of the error-revealing input sequence, and column three refers to the number of components building up the model. Column four opens a sub-table which, for a number of test cases (ranging from one to five for every circuit), lists the number of fault candidates, the number of faulty lines (a single source may correspond to several faults), and the percentage of reduction (last column) in terms of source code lines. The last column indicates, that under presence of five error-revealing test cases, the proposed extension of Reiter's algorithm allows for excluding 94 percent (considering 4 cycles) respectively 93 percent (considering 8 cycles) of all source lines. For every individual circuit, we verified that the introduced fault is among the reported ones.

As pointed out in Section 4 our algorithm extension allows for retrieving diagnosis in increasing order of cardinality. Similar to Table 1, Table 2 outlines our most recent results for dual-fault diagnosis (alongside with the total number of source code lines for every circuit). Notably, for functional faults, whilst guaranteeing that the introduced fault is among the diagnosis candidates, our algorithm allows for excluding more than 92 (for single-fault diagnosis) and 85 percent (for dual-fault diagnosis) of the source code considering up to at most five negative test cases.

Due to semantic differences between Verilog and VHDL, these results for Verilog are slightly weaker than the results for VHDL RTL presented in [Peischl and Wotawa, 2006]: Our Verilog model differs from the VHDL approach in an additional evaluation component to reflect the correct semantics of Verilog's blocking and non-blocking assignments [Peischl *et al.*, 2008]. For the ISCAS 89 test suite, on average, 187 of these evaluation components degrade the model's discrimination capabilities.

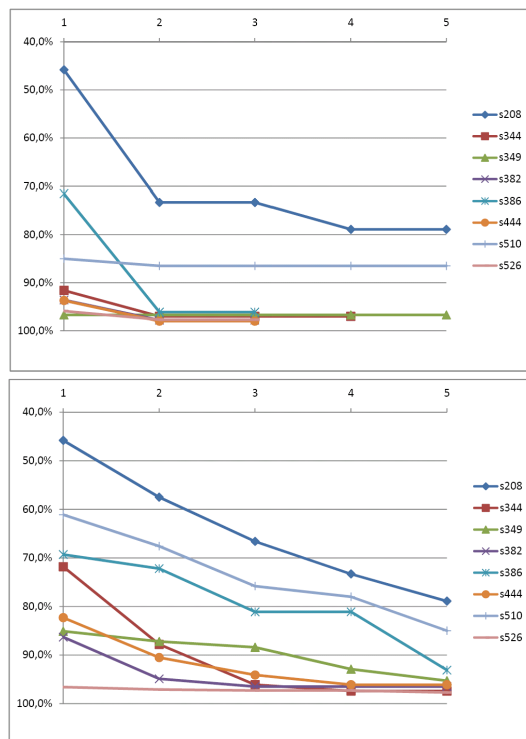



Figure 4: Results ISCAS89, multiple test cases, single fault, 4 (above) and 8 cycles (below)

Note that - particularly in a practical setting - negative test cases may provide different discrimination capabilities with respect to the obtainable diagnoses. Regarding our experiments, the circuit equivalence checking approach solely guarantees to reveal the introduced fault (constraining the test sequence generation by, for example, requiring a test case to affect disjoint output signals - might further excel the presented results).

Table 1: Empirical results single fault diagnosis, multiple test cases, ISCAS 89 benchmark suite

| circuit | cyc. no. | comp. | no. tc | fault candid. | faulty lines | %reduction |
|---|---|---|---|---|---|---|
| s208 | 4 | 2360 | 1 | 219 | 130 | 45.8 |
| | | | 2 | 105 | 64 | 73.3 |
| | | | 3 | 105 | 64 | 73.3 |
| | | | 4 | 75 | 50 | 78.9 |
| | | | 5 | 75 | 50 | 78.9 |
| | 8 | 5720 | 1 | 219 | 130 | 45.8 |
| | | | 2 | 171 | 102 | 57.5 |
| | | | 3 | 134 | 80 | 66.6 |
| | | | 4 | 105 | 64 | 73.3 |
| | | | 5 | 83 | 50 | 78.9 |
| s344 | 4 | 3748 | 1 | 59 | 36 | 91.6 |
| | | | 2 | 20 | 13 | 97.0 |
| | | | 3 | 20 | 13 | 97.0 |
| | | | 4 | 20 | 13 | 97.0 |
| | 8 | 7496 | 1 | 202 | 122 | 71.8 |
| | | | 2 | 86 | 53 | 87.8 |
| | | | 3 | 28 | 17 | 96.1 |
| | | | 4 | 21 | 11 | 97.4 |
| | | | 5 | 21 | 11 | 97.4 |
| s349 | 4 | 3688 | 1 | 27 | 18 | 96.7 |
| | | | 2 | 27 | 18 | 96.7 |
| | | | 3 | 27 | 18 | 96.7 |
| | | | 4 | 27 | 18 | 96.7 |
| | | | 5 | 27 | 18 | 96.7 |
| | 8 | 7376 | 1 | 139 | 82 | 85.1 |
| | | | 2 | 118 | 70 | 87.2 |
| | | | 3 | 107 | 64 | 88.4 |
| | | | 4 | 62 | 39 | 92.9 |
| | | | 5 | 41 | 26 | 95.3 |
| s382 | 4 | 3904 | 1 | 53 | 35 | 93.6 |
| | | | 2 | 20 | 12 | 97.8 |
| | | | 3 | 20 | 12 | 97.8 |
| | 8 | 7808 | 1 | 134 | 75 | 86.3 |
| | | | 2 | 50 | 28 | 94.9 |
| | | | 3 | 34 | 19 | 96.5 |
| | | | 4 | 34 | 19 | 96.5 |
| | | | 5 | 34 | 19 | 96.5 |
| s386 | 4 | 3568 | 1 | 234 | 145 | 71.5 |
| | | | 2 | 31 | 20 | 96.1 |
| | | | 3 | 31 | 20 | 96.1 |
| | 8 | 7136 | 1 | 252 | 156 | 69.3 |
| | | | 2 | 224 | 141 | 72.2 |
| | | | 3 | 151 | 96 | 81.1 |
| | | | 4 | 151 | 96 | 81.1 |
| | | | 5 | 54 | 35 | 93.1 |
| s444 | 4 | 4488 | 1 | 70 | 38 | 93.7 |
| | | | 2 | 22 | 12 | 98.0 |
| | | | 3 | 22 | 12 | 98.0 |
| | 8 | 8976 | 1 | 193 | 106 | 82.3 |
| | | | 2 | 108 | 57 | 90.5 |
| | | | 3 | 67 | 36 | 94.1 |
| | | | 4 | 43 | 23 | 96.1 |
| | | | 5 | 43 | 23 | 96.1 |
| s510 | 4 | 4928 | 1 | 189 | 99 | 85.0 |
| | | | 2 | 167 | 89 | 86.5 |
| | | | 3 | 166 | 89 | 86.5 |
| | | | 4 | 166 | 89 | 86.5 |
| | | | 5 | 166 | 89 | 86.5 |
| | 8 | 9856 | 1 | 491 | 257 | 61.1 |
| | | | 2 | 399 | 214 | 67.6 |
| | | | 3 | 299 | 160 | 75.8 |
| | | | 4 | 273 | 145 | 78.0 |
| | | | 5 | 189 | 99 | 85.0 |
| s526 | 4 | 4844 | 1 | 40 | 26 | 95.9 |
| | | | 2 | 28 | 15 | 97.7 |
| | | | 3 | 28 | 15 | 97.7 |
| | 8 | 9688 | 1 | 40 | 22 | 96.6 |
| | | | 2 | 33 | 18 | 97.1 |
| | | | 3 | 31 | 17 | 97.3 |
| | | | 4 | 31 | 17 | 97.3 |
| | | | 5 | 28 | 15 | 97.7 |
| Average | 4 | 3941 | | 74 | 44 | 93.6 |
| Average | 8 | 7882 | | 123 | 70 | 92.5 |

Table 2: Empirical results ISCAS 89, dual-fault diagnosis, 4 cycles

| circuit | no. lines | no. tc | fault cand. | faulty lines | %reduction |
|---|---|---|---|---|---|
| s208 | 240 | 1 | 275 | 159 | 33.7 |
| | | 2 | 710 | 152 | 36.7 |
| | | 3 | 1101 | 156 | 35.0 |
| | | 4 | 827 | 140 | 41.7 |
| | | 5 | 420 | 119 | 50.4 |
| | | 6 | 401 | 111 | 53.8 |
| | | 7 | 382 | 99 | 58.8 |
| s344 | 432 | 1 | 6338 | 138 | 68.1 |
| | | 3 | 1822 | 59 | 86.3 |
| | | 2 | 1917 | 129 | 70.1 |
| | | 4 | 757 | 48 | 88.9 |
| | | 5 | 757 | 48 | 88.9 |
| s349 | 550 | 1 | 152 | 27 | 95.1 |
| | | 3 | 338 | 40 | 92.7 |
| | | 2 | 110 | 22 | 96.0 |
| | | 4 | 110 | 22 | 96.0 |
| | | 5 | 110 | 22 | 96.0 |
| s382 | 546 | 1 | 1164 | 78 | 85.7 |
| | | 3 | 513 | 66 | 87.9 |
| | | 2 | 315 | 64 | 88.3 |
| | | 4 | 315 | 64 | 88.3 |
| | | 5 | 315 | 64 | 88.3 |
| s386 | 408 | 1 | 723 | 192 | 62.2 |
| | | 3 | 4433 | 179 | 64.8 |
| | | 2 | 4360 | 175 | 65.6 |
| | | 4 | 4683 | 170 | 66.5 |
| | | 5 | 1648 | 146 | 71.3 |
| s444 | 600 | 1 | 1152 | 145 | 75.8 |
| | | 3 | 333 | 53 | 91.2 |
| | | 2 | 333 | 33 | 96.1 |
| | | 4 | 108 | 22 | 96.3 |
| | | 5 | 108 | 22 | 96.3 |
| s526 | 639 | 1 | 551 | 40 | 93.7 |
| | | 3 | 436 | 26 | 95.9 |
| | | 2 | 366 | 23 | 96.4 |
| | | 4 | 366 | 23 | 96.4 |
| | | 5 | 366 | 23 | 96.4 |
| Average | 488 | | 1057 | 88 | 85.2 |

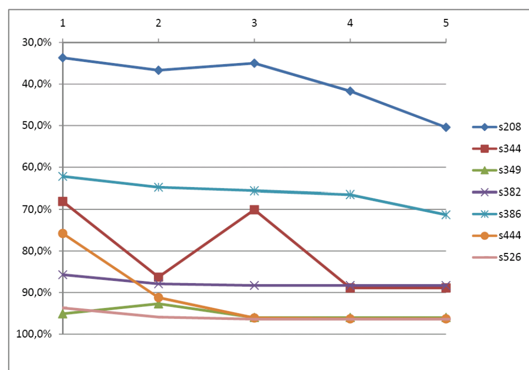



Figure 5: Results ISCAS89, double fault, cycle no. 4

## 6 Related Work

We can divide related work into four areas. First we discuss and compare the different debugging techniques used for sequential circuits. Second, we point out work related to treatment of multiple test cases. Third, we discuss the different approaches for diagnosing multiple faults and finally we discuss the work related to Ackermann constraints.

Pitchumani, Mayor, and Radia describe a diagnosis tool for VHDL that employs so-called functional fault models and reasons from first principles by means of constraint suspension [V.Pitchumani *et al.*, 1991; 1992]. They employ a hierarchical approach using stuck-at fault modes at the first level and an arbitrary failure model at the second level. As the authors do not provide experimental results, it is impossible to evaluate whether their approach outperforms ours in terms of no. of fault candidates. Kuchcinski et al. [Krzysztof *et al.*, 1993] discuss an application of algorithmic debugging to automatic fault localization in VLSI designs and propose a smooth combination of different diagnosis techniques.

Our empirical results differ from other published results [Cheng *et al.*, 1998; Alexander *et al.*, 2004] mainly in two points. We perform evaluation on unoptimized RTL representations and we compute possible fault locations at the expression level rather than at the gate level. Authors of [Cheng *et al.*, 1998] also present experimental results for the ISCAS 89 benchmark suite. The satisfiability-based approach [Alexander *et al.*, 2004] also employs optimized versions of the benchmarks and leverages recent achievements in Boolean satisfiability techniques for computing diagnoses. Notably, the authors of that approach point out that an unoptimized version makes diagnosis harder because of circuit redundancies. In source-level debugging, we inherently deal with unoptimized representations because any optimization might remove redundancies and possible fault candidates. Moreover, these researchers claim that fault-mode-free diagnosis is a desirable characteristic because fault effects can have non deterministic behavior.

Second, we discuss work related to treatment of multiple test cases. Reiter [Reiter, 1987] discusses the concept of incorporating additional measurements and Hou [Hou, 1991] explores this concept further on. F.Levy [F.Levy, 1992] has also described an approach for incremental treatment of different conflict sets. Moreover, the authors of [Meerwijk and Priest, 1992] recognized this lacking aspect and propose an approach for employing multiple test cases, however, their approach imposes unreasonable assumptions on the relationship between the circuit's input and output.

Third, there have been different approaches taken by different researchers for the diagnosis of multiple faults. The approach in [de Kleer and Williams, 1987] is based on conflict recognition and candidate generation. [H.T.Ng., 1990] provided the extension for the consistency-based diagnosis approach described by Reiter [Reiter, 1987] for the diagnosis of devices whose behavior changes over time. [S.Subramanian and R.J.Mooney, 1996] presented a similar approach, they combined this standard diagnostic approach [de Kleer and Williams, 1987] with a hypothesis checker. The authors of [Daigle *et al.*, 2006] use a fault isolation scheme, where fault effects are represented as qualitative fault signatures. As we are not aware of any publications reporting on empirical results, on the proposed techniques , it is impossible to compare our approach with their approach in terms of effectiveness.

Finally, To our best knowledge the authors of [Raiman *et al.*, 1991] were the first employing Ackermann constraints to express that components behave deterministically. We pursue a similar idea in the context of positive test cases in a simulation-driven development process. Notably [Staber *et al.*, 2006] employed Ackermann constraints for locating faults in Verilog programs by using a model checker.

## 7 Conclusion

Today's simulation-centric hardware development process requires to leverage quality test suites for locating a misbehavior's cause rather than solely detecting it. In contrast to the early 90s, we therefore need to take advantage of numerous individual test cases rather than narrowing the fault location in terms of additional measurements on a prototype.

Regarding typical test suites only a fraction of these test cases effectively reveals a fault. The remaining ones - although carrying valuable diagnosis information - do not contribute to locate the misbehavior.

This article contributes to MBD research in (1) proposing an extension to the classical model-based diagnosis theory to address the treatment of test suites, (2) relates this extension to a previously proposed filtering approach, (3) shows how to iteratively extend Reiter's algorithm to leverage multiple error-revealing faults, and (4) reports on an concrete debugging application in terms of an empirical evaluation notably taking into account dual-fault diagnoses.

## References

[Ackermann, 1954] W. Ackermann. *Solvable Cases of Decision Problems*. North Holland, 1954.

[Alexander *et al.*, 2004] Smith Alexander, Veneris Andreas, and Viglas Anastasios. Design diagnosis using boolean satisfiability. In *Proc. of the IEEE Asian-South Pacific Design Automation Conference 2004*, 2004. to appear in 2004.

[Auerbach *et al.*, 2005] G. Auerbach, M. Moulin, B. Jobstmann, and R. Bloem. Property driven design and implementation, deliverable 2.1/1. Technical report, Institute for Software Technology, Technische Universität Graz, 2005.

[Brglez *et al.*, 1989] F. Brglez, D. Bryan, and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *IEEE International Symposium on Circuits and Systems*, 1989.

[Cheng *et al.*, 1998] Kwang-Ting Cheng, Shi-Yu Huang, Kuang chien chen, and Juin-Yeu joseph lu. Fault-simulation based design error diagnosis for sequential circuits. In *35th Design Automation Conference*, pages 632–637. ACM Press, 1998.

[Daigle *et al.*, 2006] M. Daigle, X. Koutsoukos, and G. Biswa. Multiple fault diagnosis in complex physical systems. In *Proceedings of the 17th International Workshop on Principles of Diagnosis*, 2006.

[de Kleer and Williams, 1987] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[F.Levy, 1992] F.Levy. Reason maintenance systems and default theories. Technical report, Universite Paris Nord, 1992.

[Greiner *et al.*, 1989] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[Hou, 1991] Aimin Hou. A theory of measurement in diagnosis from first principles. *Artificial Intelligence*, pages 281–328, 1991.

[H.T.Ng., 1990] H.T.Ng. Model based, multiple fault diagnosis of time-varying, continuous physical devices. In *6th Conference on Artificial Intelligence Applications*, volume 1, pages 9–15, May 1990.

[IEEE, 1988] IEEE. IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987, 1988. Institute of Electrical and Electronics Engineers, Inc. IEEE.

[IEEE, 1995] IEEE. IEEE Standard Verilog Language Reference Manual LRM Std 11364-1995, 1995. Institute of Electrical and Electronics Engineers, Inc. IEEE.

[Krzysztof *et al.*, 1993] Kuchcinski Krzysztof, Drabent Wlodzimierz, and Maluszynski Jan. Automated diagnosis of VLSI digital circuits using algorithmic debugging. In Peter Fritzon, editor, *Proceedings of the first International Workshop on Automated and Algorihmic Debugging*, pages 350–367, 1993.

[Meerwijk and Priest, 1992] Arthur Meerwijk and Chris Priest. Using multiple tests for model-based diagnosis. In *Proceedings of the Third International Workshop on Principles of Diagnosis*, pages 30–39, Washington, 1992.

[Navabi, 1993] Zainalabedin Navabi. *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.

[Peischl and Wotawa, 2006] Bernhard Peischl and Franz Wotawa. Automated source-level error localization in hardware designs. *IEEE Des. Test*, 23(1):8–19, 2006.

[Peischl *et al.*, 2008] Bernhard Peischl, Naveed Riaz, and Franz Wotawa. *Advances in Automated Debugging of Verilog Designs*, volume 134 of *Studies in Computational Intelligence*. Springer, 2008.

[R. K. Brayton *et al.*, 1996] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. -T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[Raiman *et al.*, 1991] Olivier Raiman, Johan de Kleer, Vijay Saraswat, and Mark Shirley. Characterizing non-intermittent faults. In *Proceedings AAAI*, pages 849–854, Anaheim, July 1991. Morgan Kaufmann.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[S.Subramanian and R.J.Mooney, 1996] S.Subramanian and R.J.Mooney. Qualitative multiple-fault diagnosis of continuous dynamic systems using behavioral modes. In *13th National Conference on Artificial Intelligence Applications*, pages 965–970, August 1996.

[Staber *et al.*, 2006] S. Staber, G. Fey, R. Bloem, and R. Drechsler. Automatic fault localization for property checking. In *Second International Haifa Verification Conference*, pages 50–64, Haifa, October 2006.

[V.Pitchumani *et al.*, 1991] V.Pitchumani, P.Mayor, and N.Radia. A system for fault diagnosis and simulation of vhdl descriptions. In *28th ACM/IEEE Design Automation Conference*, pages 144–150. ACM Press, 1991.

[V.Pitchumani *et al.*, 1992] V.Pitchumani, P.Mayor, and N.Radia. A vhdl fault diagnosis tool using functional fault models. *IEEE Design & Test of Computers*, 9(2):33–41, April 1992.

[Wotawa, 2002] Franz Wotawa. Debugging hardware designs using a value-based Model. *Applied Intelligence*, 16(1):71–92, 2002.