On Exploiting Passing and Failing Test Cases in Debugging Hardware Description Languages

Bernhard Peischl
Softnet Austria
Graz, Austria
bernhard.peischl@soft-net.at

Naveed Riaz
College of Comp. Science and IT
University of Dammam,
Dammam, Saudi Arabia
nrmohammed@ud.edu.sa

Franz Wotawa
Institute for Software Technology
Graz University of Technology
Graz, Austria
franz.wotawa@ist.tuGraz.at

Abstract - **In this manuscript, we outline how to use test suites for software debugging of hardware description languages. We propose an algorithmic improvement for dealing with numerous failing test cases and show how to exploit passing test cases in terms of a technique called filtering. We report on results obtained on a well-known benchmark suite. The results clearly show that both passing and failing tests are capable of increasing the diagnoses accuracy in the field of software debugging.**

Model-based debugging; software debugging; debugging of hardware description languages; fault isolation.

I. INTRODUCTION

This article is an extension to previous research work [1] and reports on recent results in software debugging of Verilog designs. In contrast to the Very High Speed Integrated Hardware Description Language (VHDL) [2], Verilog [3] has a formal semantics and thus is amendable to research in verification and debugging, e.g., its synthesis semantics is formally specified in Gordon [4]. Whereas VHDL is a strongly and richly typed language, Verilog is a weakly and limited typed language [5].

Most of the research in verification deals with the detection of faults and does not address the fact that debugging involves locating and correcting the fault. In detecting faults (software/hardware testing), we make use of numerous test cases. In the recent past, numerous test cases have been employed for localizing faults, e.g., in terms of employing spectrum-based diagnosis [6, 7, 8, 9, 10].

Spectrum-based techniques, however, allow for logical reasoning at the level of dependencies and do not consider the semantics of the language in terms of value-level models [11, 12]. Our work exploits synthesis semantics and makes use of test suites. This article shows that there is solid empirical evidence that taking into account test suites improves the fault localization in HDLs considerably.

Over the last 25 years, the Artificial Intelligence (AI) community has developed a framework for system diagnosis called model-based diagnosis (MBD). This framework covers a broad range of capabilities including the isolation of faulty components and the handling of multiple fault locations [13, 14]. A specific problem solving system is automatically generated by applying task-specific, but domain-independent problem solving algorithms (e.g., Greiner et al.'s algorithm [15]) to the system model. Harnessing these techniques in software engineering tools may help to master the development of complex circuits and software-enabled systems. The state of the art in this field can be characterized by prototypes that are starting to become part of industrial applications.

In this article, we extend previous work [1, 11, 12] in the field of debugging Hardware Description Languages (HDLs) by (1) introducing an iterative version of Greiner et al.'s hitting set algorithm and (2) presenting an empirical evaluation of the impact of passing test cases. Both aspects contribute to further establish AI-based techniques in the software engineering field.

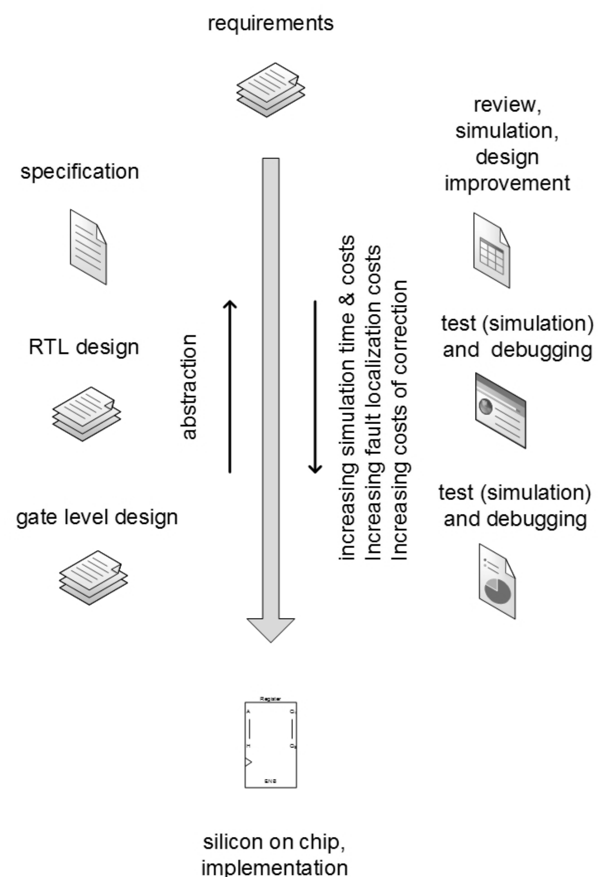

Figure 1: Design process with HDLs.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

II. SIMULATION, TEST AND DEBUGGING

Figure 1 outlines an overview of the hardware design cycle employing the Verilog HDL. The designer starts with an initial specification that primarily captures the functional requirements for the circuit being designed. Usually, this is followed by a detailed design on the register transfer level (RTL). Both designs are executable and thus are amenable to automated verification. In general, the RTL design is verified very thoroughly in terms of testing and various other analysis techniques, e.g., hazard analysis. Since there is a fixed window for start of production, these verification steps typically are conducted under time pressure and thus the time for debugging – detecting, localizing, and repairing the misbehavior – becomes a key performance indicator.

Typically, the design process iterates through several steps: design and programming is followed by a simulation of the circuit. The outcome of the simulation is compared to the specification, that is, it is checked whether the waveform traces on a higher abstraction level (the specification) deviate from the waveforms obtained from the test run on the RTL level. Previous research work, carried out in the VHDL domain, gives an intuitive understanding on how to leverage model-based diagnosis for fault localization in HDL designs (see www.ist.tugraz.at/staff/peischl/HDLDebugging.wmv).

Moreover, to reduce costs and the time to market, it is of utmost importance to detect the faults as early as possible. Thus, as testing is a viable economical technique to assure functional correctness, testing is also subject of numerous research and innovation projects. However, in order to resolve a bug, it is equally important to localize and finally remove the fault. In terms of process maturity this is captured by the defect backlog metrics (which counts the number of removed bugs) rather than the defect arrival curve that captures solely the detection rate of faults [16]. In today's software/hardware engineering processes such key performance indicators often are made available by extracting data from the underlying development tools [17, 18] thus offering the potential to quantify the effect of introducing fault isolation tools.

According to a study conducted at IBM Haifa, 50 to 80 percent of the overall development is attributed to verification activities including localization and correction [19]. Thus, particularly under local or temporal separation of the design and the test team, the automation of fault localization (and correction) is a sustainable topic for ongoing and future R&D work as it contributes to make the development process more efficient.

III. DEBUGGING SEQUENTIAL VERILOG DESIGNS

In contrast to our previous research dealing with VHDL [12, 20, 21, 22] the semantics of Verilog has been analyzed rigorously, and thus provides the necessary theoretical underpinning in language semantics and circuit synthesis. Gordon [4] provides a formal description of various semantic interpretations of Verilog like event-semantics and trace-semantics. In event-semantics (which is the semantics employed for fine-grained simulations) the change of a variable necessitates the recalculation of depending procedures.

In contrast to that, the trace semantics of Verilog computes solely the quiescent values at the end of a simulation cycle. That is, trace semantics abstracts over transient states and computes the steady values at the end of the simulation cycle. For computing these quiescent values, each procedure is evaluated only once per cycle [4]. Procedures are evaluated in a certain order such that a procedure is not evaluated until all its driving procedures have been evaluated. In other words, a procedure's outputs are computed only when all its inputs are known (or can be computed). So we build up our representation of the design by starting with processes solely dependent on known inputs and variables (e.g., the primary inputs, including clock). Afterwards, the outputs of these processes are attached to the list of already known inputs and variables. This process continues until all the procedures in the design are levelized [22]. In this way, we build up a chain of procedures and their inputs and outputs, thus allowing one for an evaluation of all the variables used in the design at the end of the simulation cycle.

Synchronous sequential circuits change their states and output values at discrete instants of time, which are specified by the rising and falling edge of a clock signal. In electrical engineering, sequential circuits are often viewed as a sequence of connected combinational circuits. This can be done by selecting specific connections (e.g., one can use minimal-cut set computation [23] for identifying these connections) and splitting them in two separated connections. The output of a stage of a specific cycle is connected to the corresponding input of the next cycle. We have adopted the same idea for providing an appropriate debugging model for sequential designs. Our representation can be broken into two phases, one in which latches change state, and one in which all the combinational blocks are evaluated. We effectively break the design at latches by treating the outputs of the latches as they were inputs and inputs of the latches as they were outputs.

In our representation, we first identify variables that we have to synthesize into latches. By splitting these variables and treating them as additional inputs and outputs, we ensure that our representation remains acyclic. Then we levelize the graph according to the levelization strategy discussed above. Thus, we receive a sequence of procedures depicting the data flow from the given primary inputs to the primary outputs. Our next step is to unroll the sequential circuits to incorporate

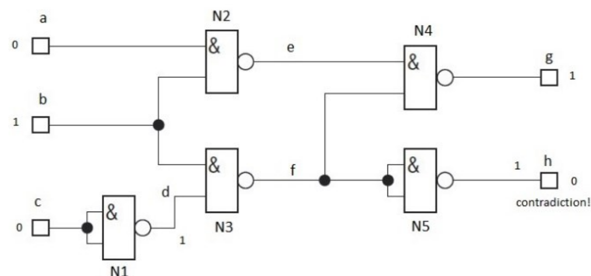

Figure 2: Illustration of a simple diagnosis problem.

multiple cycles (input sequence length). We assume that we know the number of unrollings to be performed in advance. After the levelization of all the procedures, we create the debugging model which represents our model at level 1 (cycle number 1). For every component C, we attach a timestamp i during the creation of the model to ensure a unique identification, where C_i represents the instance of component C at cycle i. Thus, we make n copies of every component involved, where n is the total number of cycles or unrollings. In this way, we create n number of instances for each component.

Diagnosis problem: A diagnosis problem considering circuit unrolling over n cycles is a triple *(SD, COMP,OBS)* where

1. $SD = \bigcup_{i=1..n} SD_i$ where SD_i is the system description for cycle i
2. $COMP = \bigcup_{i=1..n} C_i$ where C_i are the components in cycle i, and
3. $OBS = \bigcup_{i=1..n} OBS_i$ and OBS_i denote the observations in cycle i.

For every component of our model that is associated with the source code, we add an assumption $\neg AB(C)$. From a semantics point of view, this assumption denotes that the component C is assumed to work corretly. In other words this means it is assumed to be not abnormal. If we set this assumption to false, this means that the component is erroneous.

Example: Consider the digital circuit in Figure 2, which comprises five digital *NAND* gates, N_1 to N_5 and only a single cycle. We further assume that we have observed the following values on the digital circuit's inputs and outputs: $a=0$, $b=1$, $c=0$, $g=1$, and $h=0$. These values correspond to the observations *OBS*. The system description *SD* corresponds to the syntax and the semantics of the circuit (e.g., a constraint model, or horn-clause encoding of the circuit). Obviously, *SD* and *OBS* are contradictory. We can prove this by computing the values for every gate's outputs (and inputs). From $a=0$ and $b=0$, we conclude that the output of gate N_2 becomes *1*. From $c=0$ follows that the second input of gate N_3 must be *1*. This value together with $b=1$ leads to $f=0$. Consequently, $h=1$ contradicts the observed value for h. So, we know that something must be wrong and that the assumption that all components are working correct can no longer be valid.

The above given definition captures a diagnosis model for a single test case (of length n). Given this definition the diagnosis problem considering a test suite is given as follows, where we refer to the predicate *AB(C)* to denote abnormality of component C (correspondingly $\neg AB(C)$ refers to a correctly functioning component).

Diagnosis problem, test suite: Given a test suite comprising the test cases $TC_1, TC_2, \ldots, TC_k$. Let the system description SD_j be the system description considering test case TC_j and let C_i^j be the instance of component C at cycle i in test case number j. Correspondingly OBS_i^j denote the observations in cycle i of test case TC_j. The diagnosis problem $(SD^*, COMP^*, OBS^*)$ considering this test suite is given as follows:

1. $SD^* = \bigcup_{j=1..k} SD_j \cup \{\neg AB(C_0{}^j) \rightarrow \neg AB(C_1^j) \wedge \ldots \wedge \neg AB(C_n^j)\}$
2. $COMP^* = \bigcup_{j=1..k, i=0..n} C_i^j$
3. $OBS^* = \bigcup_{j=1..k, i=1..n} OBS_i^j$

IV. Iterative Computation of Diagnoses

In computing the diagnosis candidates we determine all inconsistent sub models (i.e., parts of the given design causing discrepancies). In the terminology of model-based diagnosis (MBD) these sub-models are referred to as conflicts. Since the assumption that all components of a conflict behave correctly causes the discrepancy, at least one of these components must be responsible for the misbehavior. Thus, once we have obtained all inconsistent sub models, for every component, we have to check, whether assuming this component to be abnormal allows one for getting rid of the given discrepancy in every sub model. We collect those assumption(s) that allow one for removing the given discrepancies and report the associated components as diagnosis candidates.

Recalling the previous definitions, the computation of diagnosis candidates is a consistency check for first-order sentences. In theory, one can compute diagnoses by generating all subsets Δ of *COMP* in increasing order of cardinality and checking whether $SD \cup OBS \cup \{\neg AB(C) \mid C \in COMP \setminus \Delta\}$ is consistent. Central to this algorithm is the concept of a contradictory sub model referred to as conflict in the classical MBD literature. A conflict for a diagnosis problem *(SD, COMP, OBS)* is a set $CO \subseteq COMP$ such that $SD \cup OBS \cup \{\neg AB(C) \mid C \in CO\}$ is contradictory. A conflict set is minimal iff no proper subset of it is a conflict set for *(SD, COMP, OBS)*. A set of conflicts is referred to as conflict-set $F=\{CO_1, CO_2, \ldots, CO_n\}$.
A conflict $CO = \{C_1, C_2, C_3, \ldots, C_k\}$ says that the assumption that all components are correct – that is,
$\neg AB(C_1) \wedge \neg AB(C_2) \wedge \neg AB(C_3) \wedge \ldots \wedge \neg AB(C_n)$ is true – is inconsistent with *SD* and *OBS*. However, *SD* together with *OBS* is consistent. Thus, the correctness assumptions $\neg AB(C_i)$ are responsible for the contradiction and must be altered to eliminate the conflict. This means that we must invert at least one of the $\neg AB(C_i)$ assumptions. If we now have more than one conflict, we must invert at least one (not necessarily different) assumption from every conflict. These inverted assumptions are a diagnosis because they resolve all

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

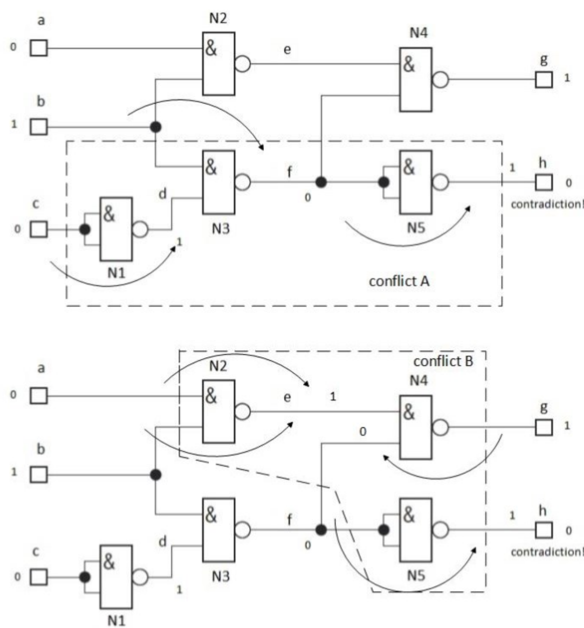

Figure 3: Example illustrating the computation of conflicts.

conflicts. So, a diagnosis is a set of components that, when assumed to behave incorrectly, leads to a consistent system state.

Continuing our example, we obtain two minimal conflicts. Figure 3 depicts them together with the computation of the contradiction values. There are two conflicts: *A*, whose components are N_1, N_3 and N_5, and *B*, whose components are N_2, N_4, and N_5. From this follows immediately that $\{N_5\}$ is a single-fault diagnosis candidate because $AB(N_5)$ resolves both conflicts *A* and *B* [12].

However, this rather inefficient brute-force approach does not work for debugging, as the number of components becomes huge. Reiter et al. [14] provide an algorithm for finding a set of minimal diagnoses and Greiner et al. [15] provide a correction to Reiter's proposal. Reiter et al. [14] and Greiner et al. [15] show how to efficiently compute diagnoses given a single conflict-set in terms of the hitting set algorithm.

The classical MBD literature is primarily focused on how to compute the diagnoses from a single conflict-set. However, in model-based software debugging, every failing test case results in one or several conflicts, i.e., a conflict-set. When considering several test cases TC_1, TC_2, TC_3, …, TC_k, we obtain a conflict-set for every test case. The resulting set C of conflict-sets therefore is $C = \{F_1, F_2, F_3, …, F_k\}$.

In theory we therefore can compute diagnoses by computing all minimal hitting sets for the union of the conflict-sets $\bigcup_0^k F_i$. However, in debugging HDLs, conflicts appear iteratively, e.g., first we execute test cases TC_1 (resulting in conflict-set F_1) and afterwards (when conflict-set F_2 becomes available) we execute a second test case TC_2 (resulting in conflict-set F_2). Following the classical literature, one can compute the diagnoses resulting from conflict-set F_1 and afterwards compute the diagnoses for the conflict-set $F_1 \cup F_2$. This results in building up the hitting set dag for the conflict-set F_1 twice as this dag needs to be built for test case TC_1 (conflict-set F_1) and for both test cases TC_1 and TC_2 (conflict-set $F_1 \cup F_2$).

In developing our automated debugging tool we managed to overcome this challenge by using an iterative variant of the original algorithm from Greiner et. al [15]. This algorithm answers the research question how to efficiently compute diagnoses in an iterative manner. Our algorithm consists of four main parts.

The procedure *Iterative_HS(C)* takes a set of conflict-sets $C=\{F_1, F_2, .., F_n\}$ and returns a dag. By collecting the edge labels *H(n)* at all nodes labeled with √ we can retrieve all (subset-minimal) diagnoses in increasing order of cardinality, i.e., all single-diagnoses can be retrieved prior to computing dual-fault diagnosis. For example, by retrieving all edge labels *H(n)* up to level three of the graph, we obtain all single- and dual-fault diagnoses. Note that the order in which the conflict-sets appear is determined by the availability of the test cases and the specific decision procedure for computing conflicts (e.g., the procedure given in [21]). Two different orderings of the same conflict-sets will result in different dags, however, from both dags we retrieve the same set of diagnoses.

The procedure *HSDAG(D, N, F)* is a modified version of the algorithm proposed in Geiner et. al. It differs from the original algorithm as it not only operates on the dag *D* and the conflict-set *F* but relies on an ordered set of nodes *N*. We use these nodes to control which nodes need to be modified in the case that the already existing dag (e.g., dag resulting from conflict-set F_1) becomes inconsistent with the new conflict being added (e.g., dag resulting from conflict-set F_1 is inconsistent with respect to conflict-set F_1 and F_2).

In order to determine these nodes, we use two further procedures. The procedure *Check_√ (D, F)* checks whether there are nodes marked with √ in the dag *D*, that according to the given conflict-set *F* are no longer valid. To establish the invariant of the algorithm, we need to label these nodes with the first set ∑ from *F* and store these nodes for later processing within *HSDAG*. The second procedure *Check_× (D, F)* checks whether there are closed nodes that need to be re-opened due to adding the conflict F. In this case, the respective node is re-opened and either labeled with the first set from F or marked with √. Again, we store this node for later processing as it might be subject of further pruning according to Greiner's algorithm.

Iterative_HS (C)
1. *Let DAG represent the growing dag. Let H(n) be the set of edge labels on the path in DAG from the root down to node n.*
2. *Generate a DAG_0 with root node n_0 with $label(n_0)=\Sigma$, where ∑ is the first set in conflict-set F_1 and $H(n_0) = \phi$.*
3. *Let N_0 be the nodes from DAG_0 in breath-first order*
4. *DAG = DAG_0; N=N_0.*
5. *For i=1 to |C|-1*
6. *DAG=HSDAG (DAG, N, F_i)*

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

7. N=Check_√ (DAG, F_{i+1}) ∪
Check_× (DAG, F_{i+1})
8. Return HSDAG(DAG, N, F_{i+1})

Comments:

1. Definition *DAG*, *H(n)* denotes the set of edge labels
2. The initial dag DAG_0 contains the root node n_0 labeled with the first element from conflict-set F_1, and two children
3. N_0 is the ordered set of nodes in DAG_0
4. Creation of the initial *DAG*
5. Iteration through all conflict-sets
6. Invoke *Check_√* and *Check_×* to retrieve those nodes from the *DAG* that need to be modified in order to be consistent with the passed conflict set F_i
7. For each conflict-set we invoke *HSDAG* explicitly given the set of nodes *N* that need to be modified
8. Finally, invoke *HSDAG* to return the pruned *DAG*

Check_√ (D, F)
1. R= ϕ
2. For all nodes $n \in D$ where label (n)= √ in breath-first order do
3. If there is $x \in F, H(n) \cap x = \phi$ then
4. $label(n) = \sum$ where $\sum$ is the first element from F for which $\sum \cap H(n) = \phi$
5. R=R ∪ {n}
6. Return R

Comments:

1. Initially, the set of nodes to be processed is void
2. Traverse nodes labeled with √ in breath-first order
3. Check if node needs to be re-labeled
4. Label node n with the first element in *F* which is not in *H(n)*
5. Store node for further processing in *HSDAG*

Check_× (D, F)
1. R= ϕ
2. For all nodes $n \in D$ where label (n) = × in breath-first order do
3. If there is a node $n' \in D$ which is labeled by √ and $H(n') \subset H(n)$ then
4. If for all $x \in F, x \cap H(n) \neq \emptyset$ then label(n)= √
5. Otherwise, $label(n) = \sum$ where $\sum$ is the first element from F for which $\sum \cap H(n) = \phi$
6. R=R ∪ {n}
7. Return R

Comments:

1. Initially, the set of nodes to be processed is void
2. Traverse nodes labeled with × in breath-first order
3. Check if node needs to be re-opened
4. Re-label node with √
5. Re-label node with the first element in F which is not in *H(n)*
6. Store node for further processing in *HSDAG*

HSDAG (D, N, F)
1. For all nodes $n \in N$ do
2. if for all $x \in F, x \cap H(n) \neq \emptyset$ then label(n)= √
3. Otherwise, $label(n) = \sum$ where $\sum$ is the first element from F for which $\sum \cap H(n) = \phi$
4. If n is labeled by a set $\sum$, for each $\sigma \in \sum$, generate a new arc with $label(n) = \sigma$. This arc leads to a new node m with $H(m) = H(n) \cup \{\sigma\}$. The new node m in D will be processed after all nodes in the same generation as n have been processed.
5. [REUSE]
a. If there exists a node n' with H(n')=H(n) ∪ { σ } then generate a directed arc from n to n'. Hence n' will have more than one parent.
b. Otherwise, generate a new node m at the end of this σ-arc
6. [CLOSING]
If there exists a node n' labeled with √ where $H(n') \subset H(n)$, then set label(n') to × for closing n. A label is not computed for n nor any successor nodes generated.
7. [PRUNING]
If the set $\sum$ is to label a node and it has not been used previously then attempt to prune D as follows:
a. If there exists a node n' which has been labeled by a set $S \in F$ where $\sum \subset S'$, then relabel n' with $\sum$. For any α in $S' \backslash \sum$ the α-arc under n' is no longer allowed. The node connected by this edge and all of its descendants are removed, except for those with another ancestor which is not being removed.
b. Interchange $label(n)$ and $label(n')$
8. Return D

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Comments:

1.-8. *HSDAG* computation according to Greiner [15]. The authors of [15] in detail describe strategies for closing, pruning and reuse of nodes.

The functions *Check_√ (D, F)* and *Check_× (D, F)* return those nodes in the dag that are not consistent with the new conflict-set F. Instead of applying the *HSDAG* procedure to the set $F_1 \cup F_2$, we apply it to conflict-set F_1 and identify those nodes that are not consistent with the new set $F_1 \cup F_2$. Afterwards, we alter, respectively, extend the dag obtained from conflict-set F_1 by applying the *HSDAG* procedure only for the identified nodes. Within *HSDAG*, the strategies for pruning, closing and re-using nodes are the same as proposed in Greiner et al. [15].

Example: Let $F_1=\{\{1,2,3\}, \{1,3\}, \{1,4\}\}$ and $F_2=\{\{1,4,5\}, \{3,4\}, \{1,2\}\}$ be conflict sets obtained from two test cases. Figure 4 represents the hitting set dag for F_1 and Figure 5 represents the corresponding hitting set graph for $F_1 \cup F_2$. In the following, we assume the existence of an ordered collection of conflict-set *Fi* for every test case *TCi* and show how to obtain the hitting set dag for the conflict-set *F1* ∪ *F2*.

Note that it is not necessary to compute all the conflicts for a given test case in advance, rather the algorithm allows one for computing the conflicts on demand. Furthermore, the algorithm allows one for computing the hitting sets in order of increasing cardinality. Thus, we can stop the computation of diagnoses once a specified depth has been reached (e.g., we retrieve solely single and dual-fault diagnosis by stopping computation at depth level 3). In the following, we illustrate the iterative variant of Greiner's hitting set algorithm that takes the set of conflict sets as an argument.

According to our algorithm, we have to build up the hitting set DAG_0. This initial dag consists of a node n_0 with two edges and $H(n_0)=\phi$. We continue with the computation of the hitting set dag $HSDAG_{F1}$ for conflict-set F_1 by invoking $HSDAG(DAG_0, N_0, F_1)$, where the collection N_0 represents the nodes from DAG_0 in breath first order.

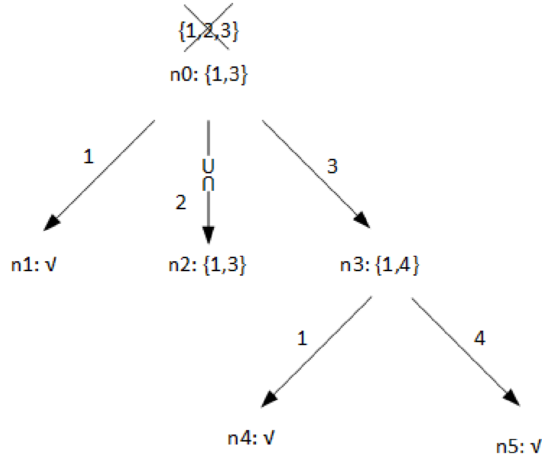

Figure 4: Hitting set dag HSDAGF1 for the conflict-set F1

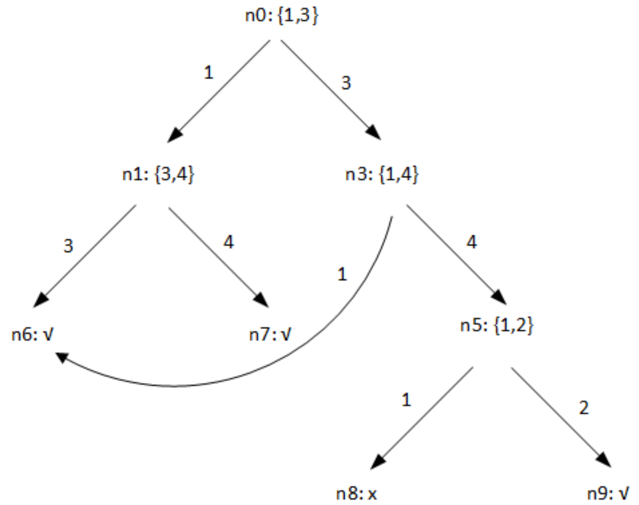

Figure 5: Hitting set for the union of the set $F_1 \cup F_2$

This step corresponds to applying the algorithm as proposed by Greiner et al. [15].

As specified in the algorithm, in line 7, we determine those nodes in $HSDAG_{F1}$ that need to be recalculated as their labeling in no longer consistent with the extended conflict $F_1 \cup F_2$. As illustrated in Figure 2 (we used circular arcs to denote pruning of node n_2) the nodes n_1, n_5 (*Check_ ✓)* and node n_4 (*Check_ ×)* are the potential candidates for re-labeling or re-opening. After having executed *Check_ ✓* and *Check_ ×* we invoke *HSDAG* (DAG_1, N_1, F_2) to finally obtain $HSDAG_{F1 \cup F2}$. This dag is consistent with the conflict-set $F_1 \cup F_2$. Figure 5 illustrates the final dag from which we retrieve the hitting sets *{1,3}*, *{1,4}* and *{2,3,4}* as the set of diagnoses.

V. EXPLOITING PASSING TEST CASES

Since passing test cases do not cause a logical contradiction, we do not obtain conflicts from passing test cases. However, passing test cases contribute in isolating faults in two ways.

First, we need passing test cases to bring the program into a state, in which another (failing) test case can reveal a misbehavior. In general, to exhibit misbehavior, sequential designs need to traverse a chain of intermediate states. In each of these states, the circuit does not exhibit erroneous behavior, and thus passing test cases contribute to finally reach a state in that the circuit exhibits misbehavior.

Second, as the different instances of our components behave independently, as we create an independent component for every unfolding of the circuit. We can use passing test cases to incorporate the notion of deterministic components into our debugging model. To illustrate the potential of using passing test cases to locate the root cause for detected misbehavior we continue with a simple example.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE I: ASSUMPTIONS, AND TEST CASES FOR OUR RUNNING EXAMPLE.

assumption	in1	in2	out	inter	verdict
AB(not), ¬ AB(exor)	1	0	1	0	fail
AB(not), ¬ AB(exor)	0	0	1	1	pass

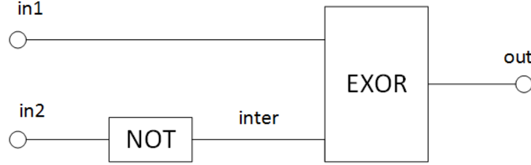

Figure 6: Part of a circuit as our running example.

As a part of a circuit, Figure 6 illustrates an exclusive or and a not gate together with a passing and failing test case for this circuit. We further assume that the circuit is faulty, that is, our test suite has identified misbehavior and we obtain both components (the *EXOR* and the *NOT* gate) as possible diagnosis candidates.

Suppose we have the test cases given in Table I. Considering the first (failing) test case in the first line, and assuming the *NOT* gate to be abnormal but the *EXOR* gate correct, we can deduce that variable *inter* becomes *0*. However, under the same assumption, the passing test case in line 2, forces variable *inter* to become *1*. We immediately see that the *NOT* gate is required to map the variable *inter* to *0* and to *1* for the same input value in_2=0. Obviously, no deterministic component can fulfill this requirement. Thus, the *NOT* gate can no longer be considered as a valid diagnosis candidate. To our best knowledge, the authors of [24] were the first who used this idea for discriminating diagnosis candidates. Unfortunately, the article gives no further insights whether the technique can be employed in practice as the authors do not provide an empirical evaluation to evaluate scalability and the improvement potential with respect to accuracy.

In the following, we propose an extension to that which – under absence of structural faults – allows one for taking advantage of passing test cases. As passing test cases do not yield to additional conflicts, we capture their specific information about diagnoses in terms of Ackermann constraints [25]. By adding these consistency constraints we incorporate the fact that the same combination of input values applied to a deterministic component *C* produces the same output for every instance of *C*. This allows one for exploiting the many test cases that do not reveal a fault. The system description with Ackermann constraints SD_A is given as follows:

System description with Ackermann constraints: Let *TC* be a set of test cases form a test suite *TC*, let $in(C_i) = \{ i_{Ci}^1, \ldots, i_{Ci}^m \}$ denote the inputs of component C_i, let $out(C_i) = \{ o_{Ci}^1, \ldots, o_{Ci}^n \}$ denote the outputs and let *SD** denote the system description of a diagnosis problem considering a test suite.

The system description with Ackermann constraints SD_A is given by,

$$SD_A = SD^* \cup CON_A,$$
$$CON_A = \neg AB(C_i) \wedge \forall_{l=1}^{m} i_{ci}^l = i_{cj}^l \rightarrow \forall_{p=1}^{n} o_{ci}^p = o_{cj}^p$$

where, $i \neq j$ and *i,j* denote indices of the test cases.

As we will show in the next section, Ackermann constraints increase the complexity of the model considerably. Therefore, we used a post processing technique proposed by the authors of [26]. As shown at the end of this section filtering allows one for iteratively applying the Ackermann constraints to the obtained diagnoses. Instead of compiling the constraints into the debugging model, we apply the constraints in terms of a dedicated post-processing phase.

Filtering refers to discarding certain diagnoses by taking advantage of further test cases TC_i. A diagnosis Δ states that $\Delta \cup SD \cup TC_i \cup \{\neg AB(C) \mid C \in COMP \setminus \Delta\}$ is consistent. This implies that there is a replacement, that is, there exists a function *replace(C)* for every component $C \in \Delta$ that allows one for repairing the program for the given test case. The function *replace(C)* allows one for producing the correct output values for the considered test case. However, considering a test suite such a replacement does not exist for all test cases in the test suite TC necessarily. Since all components *COMP \ Δ* are assumed to behave correctly, we can compute the input values *in(C)* and *out(C)* for every component *C* from *Δ* (employing forward propagation). According to this computed input/output relation, the component *C* may be required to map the same input- to different output values. This corresponds to an inconsistency and the specific diagnoses *AB(C)* is not repairable wrt. the specific test case. As there is no function *replace(C)* as stated previously, the component *C* can be removed from the set of diagnosis candidates. In this vein, we evaluate the Ackermann constraints in an iterative way by checking for different input values for a certain output value.

Algorithm 1 (Filtering): Let Δ denote a set of diagnosis candidates and let TS be a test suite.

1. *For all D ∈ Δ do*
2. *For all test cases TC_i ∈ TC do*
    a. *Let i_{Di} denote the input values and let o_{Dj} denote the output values of component D by assuming* $AB(D) \wedge \{\neg AB(C) \mid C \in COMP \setminus D\}$
    b. *If there exits i,j, i≠j, such that* $i_{Di} = i_{Dj} \wedge o_{Di} \neq o_{Dj}$ *then remove D from Δ*
3. *Return Δ*

Claim: Algorithm 1 applies the Ackermann constraints CON_A to a set of single-diagnosis candidates.

After applying Algorithm 1 to the set of single-fault diagnosis candidates, there is no component *D* at which we obtain different input values for a certain output value. Thus, we

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

conclude, that

1. $\neg\exists i,j,i\neq j \bullet (\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \wedge (\forall_{p=1}^{n} o_{Di}^{p} \neq o_{Dj}^{p})$
2. $\forall i,j,i\neq j \bullet \neg(\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \vee (\forall_{p=1}^{n} o_{Di}^{p} = o_{Dj}^{p})$
3. $\forall i,j,i\neq j \bullet (\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \rightarrow (\forall_{p=1}^{n} o_{Di}^{p} = o_{Dj}^{p})$

Thus, Algorithm 1 imposes the Ackermann constraints on the set of single-fault diagnosis candidates. For our evaluation of the approach we therefore took advantage of the filtering algorithm previously presented.

VI. PRACTICAL EXPERIENCES AND EVALUATION

Our evaluation and practical experiences answer two research questions. First, we strive to quantify the impact of exploiting passing test cases for debugging. This is done by referring to a former experiment [11] and comparing these results with our novel results considering passing test cases. Second, we evaluated the running times of the algorithm proposed herein. For both research questions, we rely on the ISCAS'89 benchmark suite [27].

We conducted our experiments on a Dell Power Edge 1950 II - 2x Quad Core with 2.0 GHz and 10GB of RAM. For computing diagnoses we relied on the extension of Reiter's algorithm described herein. Note that, for the efficient computation of diagnoses, we convert the rules given in the previous sections into a specific Horn-like encoding [21]. As the computation of conflict sets is a time critical issue, the (minimal) conflict sets are computed according to the procedure explained in [21]. The diagnosis engine and the proposed extension are implemented in the Java programming language.

Our debugging tool parses the Verilog code, builds up the model as described in this article and converts a test suite to the logical representation. Afterwards, the tool computes diagnosis candidates in increasing order of cardinality and visualizes the results by highlighting the corresponding statements, expressions or operators.

A. Time Complexity of Computing Diagnoses

For our empirical evaluation we use a Horn-like encoding of the rules presented herein. By relying on this encoding we make use of an efficient procedure to compute all minimal conflicts [21]. From the obtained conflicts we retrieve diagnoses by computing the minimal hitting sets in increasing order, where for practical purposes, primarily single- and dual-fault diagnoses are of interest. In general, searching for all diagnoses has a worst time complexity of the order $O(|MODES|*|COMP|^{s})$, where $|MODES|$ is the number of fault modes, $|COMP|$ is the number of components and s is the maximal size of the diagnoses [23]. Since we use two fault modes ($AB(C)$ and $\neg AB(C)$) and search for single and double fault diagnoses, our worst time complexity is of the order $O(|COMP|^{2})$. Note that we consider the components in every cycle as independent and thus the number of components increases with the length of the test case. However, the average running time complexity is much better because diagnoses with smaller cardinality (particularly single-fault diagnoses) are more likely than higher order diagnoses. For example, finding all single diagnoses is of order $O(|COMP|)$ assuming the decision procedure can be executed in unit time.

B. Generation of Test Suites

We obtained the test suite by injecting a single-fault (respectively a dual-fault for the second series of experiments) into the RTL design. Afterwards, we identified the faults in terms of running a simulation until we obtained five test cases revealing the introduced fault. The faults are introduced in a random way by picking a statement from every circuit and replacing this statement by another statement. That is, for every circuit, we replaced an arbitrary statement with a structurally equivalent statement (same no. of input parameters). For example, in a specific circuit we randomly selected a *NOR* statement and replaced it by an *AND* statement. Further, we implicitly removed/added negations as we substituted a logical statement by the negated counterpart (e.g., *NAND* by *AND* or vice versa). These error types are not necessarily complete w.r.t. functional errors, but as they are believed to be common in the design process, we capture the most common scenarios [28]:

- Mistakenly replacing one gate/statement by another gate/statement with the same number of inputs.
- Incorrectly adding or removing a gate/statement.

All empirical evaluations are conducted on the Verilog RTL version of the ISCAS'89 benchmark suite [27]. Further, the gate-level representations of the ISCAS'89 benchmarks have been used to obtain the correct waveform traces since our simulator allowed only for simulation of gate-level circuits.

Regarding all experiments, we verified that the injected fault (the root cause) is among the retrieved diagnosis candidates.

C. Empirical Evaluation

In our experimental setting, we assumed that an engineer only knows the correct values of the primary inputs for every simulation cycle and the outputs at the end of the final simulation cycle. That is, specified information captures the primary inputs v_{in} and their corresponding values val_{in} for every instant of time $t=1..n$, (v_{in}, val_{in}, t), together with the primary outputs and the corresponding value at time n, (v_{out}, val_{out}, n). The observations are given in terms of the primary input variables for every cycle and the primary output variables at the end of the simulation cycle (i.e., at time point n, where n is the length of the test case). Table II lists the number of primary inputs, primary outputs, and the number of gates and D-type flip-flops for the circuits we considered in our experiment. The last column is not published in Brglez et al. [27] but lists the number of lines of code in the source code representation of the Verilog RTL design.

In our first experiment, we evaluate the discrimination capability of the software debugger with an increasing number of failing test cases. Figure 7 summarizes the number of obtained single-fault diagnoses for a part of the ISCAS'89 benchmark suite.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE II: STRUCTURAL CHARACTERISTICS OF THE PROGRAMS FOR THE EMPIRICAL EVALUATION.

circ. name	no. prim. inp.	no. prim. outp.	no. D-type flip-flops	no. gates	no. lines
s208	11	2	8	96	240
s349	9	11	15	161	545
s382	3	6	21	158	544
s386	7	7	6	159	508
s444	3	6	21	181	602
s510	19	7	6	211	660
s526	3	6	21	193	634

For every circuit we randomly introduced a single fault and computed all single-fault diagnoses using the algorithm introduced in Section IV. Regarding this experiment, the introduced fault always was among the set of retrieved single-fault diagnoses.

The experiment is of practical relevance as in practice, engineers have a limited amount of failing test cases only (in our case up to five) and numerous passing test cases. The failing test cases relate the primary inputs to the (quiescent) value of the primary outputs. Our experiment does not assume any intermediate values to be known (e.g., the expected temporal response for the primary outputs). We solely rely instead on the input values for every cycle and the expected value of the primary outputs at the end of the simulation.

Figure 7 underpins the findings discussed in previous research articles as the number of single-fault diagnoses being obtained depends on both, the concrete test case and the structural complexity of the program. With an increasing number of failing tests we can considerably reduce the number of obtained single-fault diagnoses.

Afterwards, we repeated a similar experiment but in addition applied the filtering algorithm to exploit the numerous passing test cases for debugging. Figure 8 illustrates the improvement we gained from applying these test cases together with the failing test cases. In the figure, no passing test case (0) refers to using exactly five failing test cases. In addition, we applied the filtering procedure by using up to four passing test cases. As in the previous experiment, the introduced fault always is among the retrieved set of single-fault diagnoses.

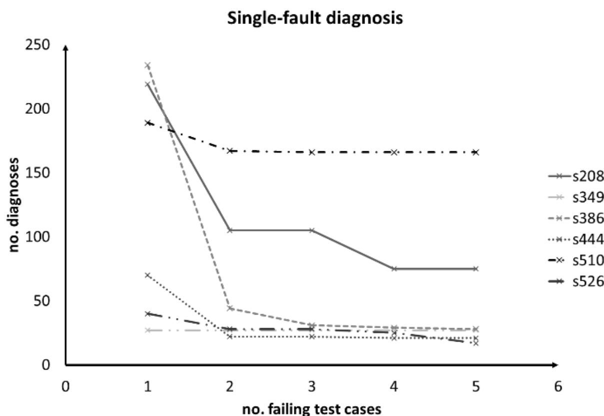

Figure 7: Single-fault diagnoses with increasing number of failing test cases.

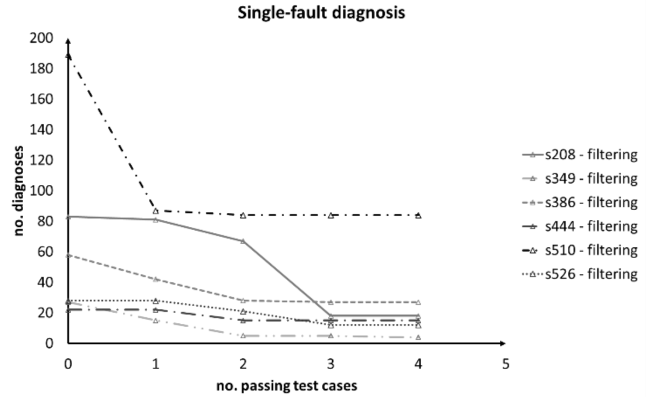

Figure 8: Improvements due to the filtering technique.

Regarding our experiments, passing test cases were able to further reduce the number of single-fault diagnoses for every program we considered.

Figure 9 outlines the running times for our algorithm we obtained for computing all single-fault diagnoses including the application of the filtering procedure. Remarkably, the random fault introduced in circuit s510 yields to a significant number of diagnoses and thus higher response times when compared to the remaining circuits. It appears that, (1) the structural complexity, (2) the random fault we introduced, and (3) the specific test cases revealing the introduced fault results in a (at least in relation to the other circuits) computationally expensive problem. On average we obtained 74 single-fault diagnoses corresponding to 44 faulty lines in the source code. Regarding our experiments, a designer can exclude over 93 percent of the statements and expressions from being faulty.

In a second series of experiments, we randomly injected two faults into every circuit we considered for our experiment.

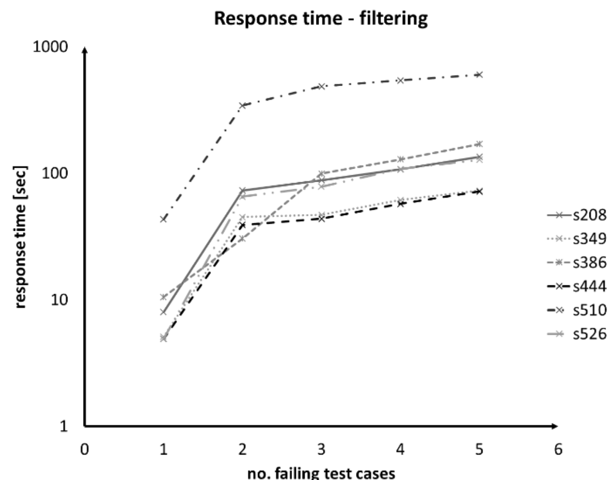

Figure 9: Running times for computing single-fault diagnoses.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

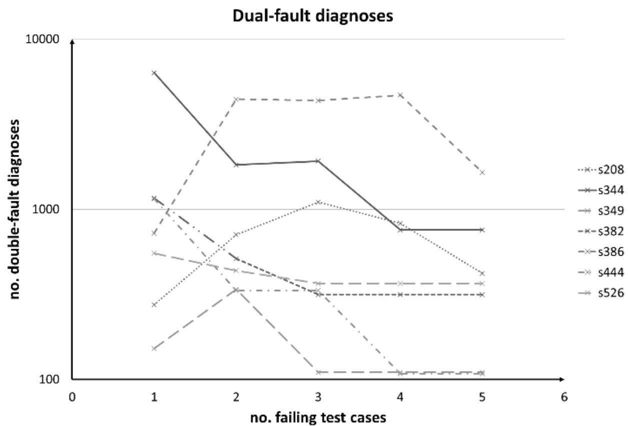

Figure 10: Dual-fault diagnoses for a part of the ISCAS'89 benchmarks

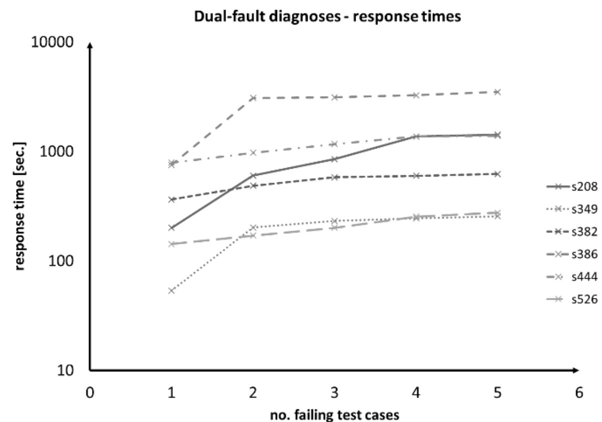

Figure 11: Running times for computing dual-fault diagnoses

Again, we generated up to five failing test cases and computed all single- and dual-fault diagnoses by using the algorithm introduced in Section IV. Figure 10 outlines the number of dual-fault diagnoses we obtained with increasing number of failing test cases.

For some circuits (e.g., s208, s349 and s386) the obtained number of dual-fault diagnoses is not monotonically decreasing. Unlike to computing single-fault diagnoses, this may happen due to the fact that test cases may mask some faults. For example, the first test case might reveal the first fault being introduced but may mask the second fault we introduced. As a consequence the second fault is not among the retrieved list of diagnoses. The second test case might reveal the second fault, therefore, after computing diagnoses, the second fault will also appear in the list of diagnoses. As a result, in the presence of multiple faults, when adding further test cases, in some cases, the number of diagnoses might increase. However, for most of the circuits in our experiment the number of dual-fault diagnoses decreases with increasing number of failing test cases. Again, for every circuit being considered in the experiment, the introduced pair of faults appeared among the computed dual-fault diagnoses

Figure 11 outlines the running times for computing dual-fault diagnoses. Note that one usually computes diagnoses in increasing order of cardinality. That is, we first use our algorithm to compute single-fault diagnoses and only if the real cause of misbehavior cannot be explained by a single-fault we continue with computing dual-fault diagnoses.

VII. Discussion and Related Work

The research work regarding fault localization is hardly comparable due to the variety of benchmarks and different approaches and abstraction levels being used. For this reason we predominately discuss the work where empirical results on the ISCAS'89 (and purely combinational ISCAS'85) benchmark suites have been obtained.

The authors of [29] propose a method that uses the crosstalk-induced pulse fault simulation to identify a set of suspected faults that are consistent with the observed responses. The authors propose two simulation-based methods to diagnose a crosstalk-induced pulse fault which may occur between the clock line of a flip-flop and a signal line in the sequential circuit. The first method is a basic method to diagnose the crosstalk-induced pulse faults. It uses information about the first and the last timeframe at which a crosstalk-induced pulse fault is detected. The second method uses additional information to reduce the computational complexity for diagnosing a crosstalk induced pulse fault. In order to reduce computational effort, the second method uses stored state information to calculate the primary output values at the present timeframe.

The authors of Boppana et al. [30] introduce a state information-based technique for supporting fault diagnosis. Storage of faulty state data corresponding can be used to reduce stored data. It has been demonstrated that the technique can support fast fault diagnosis in partial scan environments, particularly if the resulting design is acyclic. In doing so, the simplified structure of the partial scan circuit has been exploited. Arguably, the most important contribution of this work is the useful storage of faulty circuit data corresponding to flip-flops. Therefore, the circuit is no longer considered as a black-box element at diagnosis time and hence offers increased flexibility.

The authors of Smith et al. 2004 [31] present a SAT-based solution to design diagnosis of (also dual) faults where existing solvers can be utilized. The authors also examine different implementation trade-offs and heuristics. Like our work, experiments with dual faults demonstrate the efficiency and practicality of the approach.

Peischl and Wotawa 2006 [22] present work on VHDL and present results on the ISCAS'89 benchmarks regarding single test cases. This article introduces a model based on the MBD approach that abstracts over individual events within a single simulation cycle and allows one for performing source level debugging.

So far, the work on comparing the various debugging approaches is rather limited. In Finder et al. 2010 [32], a methodology is presented to evaluate debugging algorithms from a qualitative perspective. Notably, the authors of [32] lift the fault model originally defined for gate level net lists to higher level descriptions like HDLs and conclude that some

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

types of bugs can be handled using a simulation-based approach, while other types of bugs cannot be handled. Peischl and Wotawa [20] argue that simulation-based techniques may miss faults. Thus, the comparison of MBD-based techniques with simulation-based techniques can only be done in terms of case studies under a given set of assumptions (e.g., fault types) and the conclusions are restricted to the respective benchmarks being considered. Further, the different metrics to measure the quality or adequacy of the fault candidates (e.g., some notions of neighborhood, number of fault candidates, etc.) at the various levels of granularity (statements, expressions, operators etc.) make a generalized and meaningful comparison almost impossible.

The authors of [33] outline the results obtained with a fault-simulation based technique. The main differences to our experimental settings are as follows:

- In contrast to our experiments, the results described in [33] have been obtained on the gate level as this work does not focus on locating the erroneous statement or expression at the source code level.
- The fault localization is performed on optimized circuits rather than on the original design. The optimized circuits only comprise *AND* and *OR* gates. In contrast to that, our work deals with RTL designs and with locating the root cause on the source code level.
- The authors of [33] only specify the minimal length of the test cases and only give an upper limit of the number of failing and passing test cases.
- Most notably, and in contrast to our work, the technique introduced in [33] requires the primary input values and correct values for the primary output for every time frame. To our experience, a designer does only use limited correctness information (e.g., the expected signal values at the end of a test case) rather than having knowledge about all the intermediate values.
- The approach pursued in [33] fails for the circuits' s208 and s444 due to the high complexity for long failing test cases.
- Rather than allowing every statement, expression or signal to be faulty, only signals are considered as potential root cause for the observed misbehavior. This is a major difference to our approach, as our technique allows one for obtaining diagnosis candidates at the level of statements, and expressions including individual variables.
- Erroneous implementations are generated in terms of injecting a gate type error randomly after decomposition into *AND* and *OR* gates [33, 34].

Although the response times and the number of obtained diagnoses can hardly be compared due to the points mentioned above, our empirical results correspond with the results outlined in [33] in two respects:

- The number of potential single-fault diagnoses is reduced substantially when employing a couple of failing test cases.
- It appears that further increasing the number of failing test cases yields to saturation, i.e., the number of diagnosis candidates does not appear to become considerably smaller even when increasing the number of failing test cases substantially. In this respect, the decision to empirically investigate techniques that allow one for incorporating passing as well as failing test cases gains even more importance.

In the models used herein, we abstract over time (we use trace semantics) and variable values (we only operate with values *0* and *1*). This kind of abstraction is particularly suited for designs that can be synthesized. For mainstream programming languages (for example, the Java programming language) other abstractions like, for example, functional dependences or abstract interpretation models can be beneficial [35]. Finding a suitable abstraction is the key in successfully applying model-based software debugging, as this allows one for trading off computational complexity and accuracy of the obtained diagnoses. Today, it is an open issue, how to systematically find adequate abstractions and this issue requires further research.

Recent work approaches the fault localization problem merely from an algorithmic point of view. These articles differ from our research in two aspects. First, none of the works addresses source level debugging (in the sense of automatically highlighting the potential fault candidates at the level of expressions and statements at the HDL RTL level) and second, the evaluation of the novel algorithms and techniques is only performed on combinational circuits (mostly on the ISCAS'85 benchmark suite) and does not address sequential circuits (and thus the notion of state).

Siddiqi and Huang [36] propose a heuristic measurement point selection that can be computed efficiently. Furthermore, the technique introduced in [36] makes use of hierarchical diagnosis. For the largest system, where even this approach fails, the authors make use of specific abstractions. Unlike our approach (HDL trace semantics is an abstraction of the finer grained event semantics) this abstraction is not related to HDL language semantics. Experiments with the (combinational) ISCAS'85 benchmark suite indicate that this approach scales to all circuits in the suite except for one.

Feldman et al. [37] combine passive monitoring, probing and test sequencing with automated test pattern generation. Within their framework (FRACTAL), the authors empirically evaluate the trade-offs of three algorithms by performing experiments on the ISCAS'85 combinational benchmark circuits.

The same authors further propose a stochastic fault diagnosis algorithm called SAFARI [38], which trades off guarantees of computing minimal diagnoses for computational efficiency. In terms of the ISCAS'85 benchmarks, the authors empirically demonstrate that SAFARI achieves several orders of magnitude speedup over two well-known deterministic algorithms. The authors argue

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

that SAFARI can be of broad practical significance, as it can compute a significant fraction of minimal cardinality diagnoses for systems too large or too complex to be diagnosed by existing deterministic algorithms.

Abreu and van Gemund [39] use a heuristic approach to approximate the computation of minimal hitting sets and present a low-cost approximate minimal hitting set algorithm called STACCATO. The authors use a heuristic function that is particularly tailored to MBD problems. One difference to our work is that the candidates are not retrieved in increasing order of cardinality. Whether STACCATO can be tailored to software debugging (e.g., making use of STACCATO only for large problem spaces) is an open issue and subject of future research.

Like in this article, Bailey and Stuckey [40] present an incremental approach to compute hitting sets and show that circuit error diagnoses requires finding all minimal unsatisfiable subsets to compute minimal diagnoses. However, the proposed hitting set algorithm works in the context of constraints whereas the proposed algorithm herein is used in our automated debugging tool and is a variant of Reiter's hitting set algorithm dealing with sets of items.

VIII. CONCLUSION

In this article, we showed how to employ the well-founded theory of model-based diagnosis to fault localization in Verilog designs. We discussed today's simulation driven development lifecycle and proposed a model that can handle test suites comprising passing and failing test cases. To exploit passing test cases we used a technique called filtering and related this technique to Ackerman constraints. Regarding failing test cases, we used an iterative version of Reiter's hitting set algorithm.

We present an empirical evaluation of the impact of passing test cases alongside with an analysis of the running times of an iterative variant of Greiner et al.'s hitting set computation in the context of our automated debugging tool for HDLs. In this respect, we reported on exhaustive empirical results on one of the mostly employed benchmarks in the area of HDLs, the ISCAS'89 benchmark suite. Our results clearly indicated that exploiting test suites (comprising passing as well as failing test cases) considerably may improve the accuracy of the obtained diagnoses.

REFERENCES

[1] B. Peischl, N. Riaz, and F. Wotawa, "Using filtering to improve value-level debugging of verilog designs," In VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle, pp. 49–54, 2013.

[2] Z. Navabi, "VHDL: Analysis and Modeling of Digital Systems," McGraw-Hill, 1993.

[3] IEEE, IEEE Standard Verilog Language Reference Manual (LRM), IEEE STD 11364-1995, 1995.

[4] M. J. C. Gordon, "Relating event and trace semantics of hardware description languages," The Computer Journal, vol. 45, no. 1, pp. 27–36, 2002.

[5] S. Bailey, "Comparison of VHDL, Verilog and SystemVerilog," Digital Simulation White Paper, http://boydtechinc.com/btf/archive/att-1977/01-LanguageWhitePaper.pdf (last accessed on 09.05.2014).

[6] R. Abreu, P. Zoeteweij, and A. J C Van Gemund, "On the accuracy of spectrum-based fault localization," In Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007, TAICPART-MUTATION 2007, pp. 89–98, 2007.

[7] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, ICSE, pp. 82–91, ACM, 2006.

[8] D. Hao, L. Zhang, T. Xie, Hong Mei, and J. Sun, "Interactive fault localization using test information," Journal of Computer Science and Technolgy, vol. 24, no. 5, pp. 962–974, 2009.

[9] B. Liblit, M. Naik, A. X Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," ACM SIGPLAN Notices, vol. 40, no. 6, pp. 15–26, 2005.

[10] Y. Yu, J.A. Jones, and M.J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," In Software Engineering, 2008. ICSE '08, ACM/IEEE 30th International Conference on, pp. 201–210, 2008.

[11] B. Peischl, N. Riaz, and F. Wotawa, "Automated Debugging of Verilog Designs," International Journal of Software Engineering and Knowledge Engineering, vol. 22, no. 5, pp. 695–723, 2012.

[12] B. Peischl and F. Wotawa, "Model-Based Diagnosis or Reasoning from First Principles," IEEE Intelligent Systems, vol. 18, no. 3, pp. 32-37, IEEE Computer Society, May/June 2003.

[13] J. de Kleer, A. K. Mackworth, and R. Reiter, "Characterizing diagnoses," In AAAI , Howard E. Shrobe, Thomas G. Dietterich, and William R. Swartout, editors, pp. 324–330, AAAI Press / The MIT Press, 1990.

[14] R. Reiter, "A theory of diagnosis from first principles," Artificial Intelligence, vol. 32, no. 1, pp. 57-95, 1987.

[15] R. Greiner, B. A. Smith, and R. W. Wilkerson, "A correction to the algorithm in Reiter's theory of diagnosis," Artificial Intelligence, vol. 41, no. 1, pp. 79-88, 1989.

[16] S. H. Kan, Metrics and models in software quality engineering, Addison-Wesley, 1995.

[17] B. Peischl, V. R. Torrents, A. Kalchauer, S. Lang, "Business intelligence in software qualiy monitoring: Experiences and lessons learnt from an insutrial case study," In Proceedings of the 6th Software Quality Days (SWQD 2014), pp. 34-47, 2014.

[18] L. Lavazza and M. Mauri, "Software process measurement in the real world: Dealing with operating constraints," In Lecture Notes in Computer Science, Qing Wang, Dietmar Pfahl, DavidM. Raffo, and Paul Wernick, editors, Software Process Change, vol. 3966, pp. 80–87, Springer Berlin Heidelberg, 2006.

[19] B. Jobstmann, R. Bloem, A. Cimatti, G. Auerbach, and M. Moulinn, "Prosyd: Property-based system design, deliverable 2.1/1," PROSYD Techical Report, FP6-IST-507219, 2005.

[20] B. Peischl and F. Wotawa, "Error traces in model-based debugging of hardware description languages," In Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG'05, pp. 43–48, New York, NY, USA, ACM, 2005.

[21] B. Peischl and F. Wotawa, "Computing diagnosis efficiently: A fast theorem prover for propositional horn theories," 14th Internaltional Workshop on Principles of Diagnosis (DX-03), pp. 175–180, June 2003.

[22] B. Peischl and F. Wotawa, "Automated source-level error localization in hardware designs," IEEE Design & Test of Computers, vol 23, no. 1, pp. 8-19, January 2006.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[23] F. Wotawa, “Applying Model-Based Diagnosis to SoftwareDebugging of Concurrent and Sequential ImperativeProgramming Languages,” PhD thesis, Technische Universität Wien, 1996.

[24] O. Raiman, J. de Kleer, V. A. Saraswat, and M. Shirley, “Characterizing non-intermittent faults,” In AAAI, Thomas L. Dean and Kathleen McKeown, editors, pp. 849–854, AAAI Press / The MIT Press, 1991.

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

[26] F. Wotawa, “Debugging hardware designs using a value-based Model,” Applied Intelligence, vol. 16, no. 1, pp. 71-92, 2002.

[27] F. Brglez, D. Bryan, and K. Kozminski, “Combinational profiles of sequential benchmark circuits,” In IEEE International Symposium on Circuits and Systems, pp. 1929–1934, 1989.

[28] D. Nayak and D. M. H. Walker, “Simulation-based design error diagnosis and correction in combinational digital circuits,” In VTS, pp. 70–79, IEEE Computer Society, 1999.

[29] H. Takahashi, M. Phadoongsidhi, Y. Higami, K.K. Saluja, and Y. Takamatsu, “Simulation-based diagnosis for crosstalk faults in sequential circuits,” In Proceedings to the 10th Asian Test Symposium, pp. 63–68, 2001.

[30] V. Boppana, I. Hartanto, and W.K. Fuchs, “Fault diagnosis using state information,” In Proceedings of Annual Symposium on Fault Tolerant Computing, pp. 96–103, 1996.

[31] A. Smith, A. Veneris, and A. Viglas, “Design diagnosis using boolean satisfiability,” In Proceedings of the 2004 Asia and South Pacific Design Automation Conference, ASP-DAC ’04, pp. 218–223, Piscataway, NJ, USA, 2004, IEEE Press.

[32] A. Finder and G. Fey, “Evaluating debugging algorithms from a qualitative perspective,” In Forum on Specification Design Languages (FDL 2010), pp. 1–6, 2010.

[33] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and J.-Y. J. Lu, “Fault-simulation based design error diagnosis for sequential circuits,” In Proceedings of the Design Automation Conference, 1998, pp. 632–637, 1998.

[34] A. Kuehlmann, D. I. Cheng, A. Srinivasan, and D. P. LaPotin, “Error diagnosis for transistor-level verification,” In Proceedings of the 31st Annual Design Automation Conference, DAC ’94, pp. 218–224, New York, NY, USA, 1994, ACM.

[35] W. Mayer and M. Stumptner, “Model-based debugging using multiple abstract models,” In Fifth International Workshop on Automated and Algorithmic Debugging, CoRR, pp. 55-70, cs.SE/0309030, 2003.

[36] S. Siddiqi and J. Huang, “Sequential diagnosis by abstraction,” Journal of Artificial Intelligence Research, vol. 41, no. 2, pp. 329–365, May 2011.

[37] A. Feldman, G. Provan, and A. van Gemund, “A model-based active testing approach to sequential diagnosis,” Journal of Artificial Intelligence Research, vol. 39, no. 1, pp. 301–334, September 2010.

[38] A. Feldman, G. Provan, and A. Gemund, “Approximate model-based diagnosis using greedy stochastic search,” In of Lecture Notes in Computer Science, Abstraction, Reformulation, and Approximation, Ian Miguel and Wheeler Ruml, editors, volume 4612, pp. 139–154, Springer Berlin Heidelberg, 2007.

[39] R. Abreu and A. J. C. van Gemund, “A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis,” In SARA, Vadim Bulitko and J. Christopher Beck, editors, AAAI, pp. 2-9, 2009.

[40] J. Bailey and P. J. Stuckey, “Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization,” In Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages, PADL’05, pp. 174–186, Berlin, Heidelberg, 2005, Springer-Verlag.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org