Automated Source Level Error Localization in Hardware Designs

Bernhard Peischl and Franz Wotawa*
Technische Universität Graz
Institute for Software Technology (IST)
{peischl,wotawa}@ist.tu-graz.ac.at

contact author: Bernhard Peischl
postal address: Inffeldgasse 16b/2
A-8010 Graz, Austria
phone: +43 316 873 5730
fax: +43 316 873 5706
e-mail: peischl@ist.tu-graz.ac.at

keywords: design error diagnosis, software debugging, model-based diagnosis, fault localization

* We listed the authors in alphabetical order.

Abstract

Recent achievements in formal verification techniques allow for fault detection even in larger-sized real world designs. However, rather than supporting a designer in detecting a design’s malfunctioning, tool support for localizing the faulty statements is of uttermost importance, too. This might reduce development time and thus overall project costs considerably. In this article we promote the idea of automated source-level debugging and introduce our novel debugging model that, in contrast to our previous models, allows for source-level debugging of even larger sized VHDL designs at the granularity of statements and expressions. We present empirical results for our cycle-based modeling approach for the well known ISCAS’89 benchmark circuits. Notably, our technique is fully-automated and does not require that an engineer is familiar with formal verification techniques.
1 Introduction and Motivation

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 and development time considerably. In the area of hardware design, shrinking design windows, ever increasing design complexity, and expensive prototyping are pushing companies towards putting considerable efforts onto design analysis, verification and debugging at the source code level.

Production use of source code based methodology enables designers to create very complex systems on chip, thereby employing the source code as the basis for logic synthesis. Usually developers detect faults in hardware designs by means of verification tools, for example, model checkers [5] or by means of test cases together with the extensive use of simulators and waveform trace tools. In both cases the hardware designer identifies test vectors leading to the unexpected results.

After detecting a design’s malfunctioning we have to locate the failure’s real cause and correct the design. The input of this task is the test case revealing the erroneous behavior as well as the source code. The output of the fault localization process is a number of source-code artifacts which might be responsible for the specific misbehavior the design exhibits for the given test case. Figure 1 outlines the design flow together with the underlying representations as well as the input and output artifacts for the case of employing a model checking tool.

In this article we introduce a new model for diagnosing VHDL designs. Our approach focuses on the model-based diagnosis (MBD) [30, 7] paradigm and directly employs structure and behavior for software debugging and thus reasons from first principles. MBD offers several advantages compared to other fault localization techniques [23], however, as for example pointed out in [26], reasoning from first principles is time consuming. Therefore creating a model that represents the problem domain in a way that allows for the desired coverage and quality in localizing faulty source-code artifacts while remaining computationally tractable is the crucial issue in applying MBD.

In the recent past we’ve developed models for the VHDL domain at different abstraction levels. Some of them allow for detailed fault localization at the statement and expression level even in designs that synthesize into sequential circuits. Theses models accurately represent VHDL’s event semantics including the detailed semantics of events and processes in order to represent state information inherently present in sequential designs. Although such an accurate modeling approach offers good fault localization capabilities it suffers from computational intractability for larger designs.

In this article we show how to overcome this by introducing a new model that abstracts over individual events within a single simulation cycle. We can incorporate the new model in our MBD approach to source level debugging and thus all advantages adhere. These advantages of our approach to source-level debugging are: (1) we can derive the diagnosis model automatically from the source code, (2) there is no requirement to use fault models, although our approach allows for incorporating them, (3) our approach is not restricted to single-fault diagnosis, (4) we can incorporate abstract domains such as arrays or integers, (5) the approach is very flexible in exploiting knowledge about correct behavior.

2 Related Work on Error Localization

Existing work on design error diagnosis primarily deals with gate-level descriptions. In this area ba-
sically there are two approaches to fault localization [11]: simulation-based approaches [19, 17, 28, 32] and symbolic approaches [2, 18, 19]. In comparison symbolic approaches are accurate but suffer from combinatorial explosion whereas simulation-based approaches, although scalable with design size, require numerous test vectors to be accurate enough.

Moreover, there is a bulk on recent work that deals with source-level debugging of mainstream programming languages [35, 1, 9]. In contrast to these languages, where debugging models have to reflect complex object-oriented structures and the dynamic behavior of objects, the major challenge in creating a debugging model for HDLs is the treatment of simulation time. However, to our best knowledge only a few publications deal with fault localization in HDLs. [3, 4] reports the application of static program slicing to VHDL. In [26, 27] the authors describe a diagnosis tool for VHDL that employs so called functional fault models and reasons from first principles by means of constraint suspension. Kuchcinski et al. [16] discusses an application of algorithmic debugging to automatic fault localization in VLSI designs and proposes a smooth combination of different diagnosis techniques.

Regarding the modeling approach proposed herein, to our best knowledge, we are not aware of similar work in the domain of fault localization and diagnosis. In the area of circuit simulation, so called levelized compile-code simulators employ a similar modeling approach for synchronous circuits. These levelized compile-code simulators [33] have the potential to provide much higher simulation performance than event-driven simulators because they eliminate much of the run-time overhead associated with propagating and handling events and their effects.

From an language semantics point of view our approach exploits the equivalence between event and trace semantics for synchronous designs. Gordon [8] discusses this relationship on a simplified, Verilog-like language HDL. The author explicates a semantic pseudo code for this language, gives a formal event semantics via a simulation cycle and from this a trace semantics. For a class of syntactically restricted programs, commonly known as the RTL subset, together with some reasonable assumptions Gordon [8] shows that the trace semantics derived from the simulation cycle corresponds to the event semantics.

The work we present herein rests upon our own research [23, 34, 24, 25]. In [23] we outline the challenges in applying MBD for software debugging in general. In [34] we present an abstract model that relies on functional dependencies [13] and abstracts from individual values referring to signals or variables merely as being correct or incorrect with respect to a given test case. Although this model allows for fault localization in combinational VHDL designs up to 10 MB, we only can locate the faulty process rather than the faulty statement or expression.

Another so called value-level modeling approach allows for a detailed localization of the faulty source-code artifact but suffers from computational intractability for larger designs [24]. This model accurately represents VHDL’s event and process semantics in a static fashion. In [25] we show that solely relying on the counterexample trace of an event-based simulation to built up a dynamic model for a specific test-case is not reasonable due to the presence of so called conditional dependencies in VHDL. The novel value-level modeling approach we present herein abstracts over individual events and relies on functional dependencies to represent the data-flow in the naively synthesized circuit. In this way it is feasible to overcome the intricacies caused by conditional dependencies and to improve scalability compared to previous value-level models [24] considerably.

3 A Running Example

First and foremost we outline the idea of the MBD approach to design error diagnosis by means of a small example. Figure 2 shows a VHDL design and the corresponding circuit. Besides of the clock signal $CLK$, we have a primary input $I$, a primary output $O$, and a variable holding an internal state $S$. Moreover, we store the last three bits of the output stream in signals $y_2$, $y_1$, and $y_0$ respectively. When input $I$ is set to ‘0’ the JK-flip-flop (JK-FF) is in toggle mode, thus process $p_1$ of our design acts as frequency by two divider. When setting $I$ to ‘1’ the JK-FF is in non-change mode, hence its output represents the state’s present value. Our circuit furthermore indicates, on its three outputs $z_2$, $z_1$, and $z_0$, the difference between the number of ones and the number of zeros in the last three
bits of the output $O$. We consider this difference to be positive if the number of ones in the last three bits exceeds the number of zeros and is negative otherwise and encode this difference in two’s complement form.

On the source-code level we explain the design’s behavior as follows. Process $p1$ executes whenever $CLK$ changes, the first if statement (line 13) checks whether there is a falling edge of $CLK$, whereas the second one (line 14) assures that the state’s value toggles if input $I$ is set to ‘0’ only. Statement 18 transfers the state’s value to the distinguished output $O$. Consequently, the output $O$ corresponds to the state’s value at the end of the simulation cycle.

Process $shiftReg$ implements the 3-bit shift register which holds the last three bits of the stream received, so that the combinational logic specified by process $comb$ can determine the number of ones and zeros and produce an appropriately encoded output $z2z1z0$.

In order to demonstrate our design diagnosis approach, we introduce two bugs. In line 18, instead of assigning the state $S$ to output $O$, we erroneously assign the conjunction of state $S$ and input $I$. Our second bug is in line 41, where we assign the constant ‘000’ instead of ‘001’ to variable $z$.

Figure 3 outlines the traces we obtain from simulating our design; dashed lines indicate the expected and thus correct signal values. At time $t1$ the negative edge of $CLK$ and $I$ causes $S$ to change its value from ‘0’ to ‘1’. Consequently, since $I$ is ‘0’, the output $O$ goes to ‘0’. This setting contradicts the specification saying that signal $O$ and $S$ have the same values after process execution. Moreover, we expect $z$ to be ‘001’ rather than ‘111’ at the simulation’s end. To localize the faulty source-code artifact, we need to have at least a partial specification of the correct behavior. For this example, we assume that we’re solely aware of the correct temporal behavior of signal $O$ and the variable $z$ at the end of the second cycle.

Our approach to design error diagnosis focuses on adopting MBD [30, 7] to the domain of hardware description languages, in particular to VHDL (Very High-Speed Integrated Circuit Hardware Description Language) [12]. In searching for possibly faulty source code artifacts, henceforth referred to as components, (e.g. expressions, statements or whole processes) we identify the smallest number of componen-
nents, that, when assumed to behave correctly, yield to incorrect output values. Essentially, this captures the notion of an inconsistent sub-model, also called a conflict in the terminology of MBD. Assuming that this sub-model’s components work correctly causes a contradiction, thus at least one of those components must behave faulty.

Once we know all inconsistent sub-models, for every component we check whether assuming this component to behave abnormal allows for getting rid of the given inconsistencies in every sub-model. We collect those (single) assumptions that allow for removing the given contradiction(s) and report them as (single) fault candidates.

To clarify this procedure, we again consider our running example alongside with a so-called diagnosis model. We anticipate this model in Figure 4, and give a detailed discussion on the underlying abstraction and the automated model creation process in Section 4. In the figure we indicate specified signal values by surrounding boxes. Note that we solely employ these values in order to compute our fault candidates.

In localizing a malfunctioning’s possible causes, we first identify all minimal sub-models yielding to contradiction with the given test-case. In Figure 4, if we assume the and operator and the succeeding signal-assignment statement to behave correct, the input $I[1]$ causes the and operator’s output and consequently the distinguished output $O[2]$ to be ‘0’. The set \{and[2], <= [2]\} (referred to as conflict 1 in Figure 4, where the number in brackets refers to the number of the clock cycle for which we assume a specific component to behave correctly) therefore is a minimal sub-model causing inconsistency. Note that we treat components which belong to the same source location but to different simulation cycles independently.

We can explain our second inconsistent sub-model, namely conflict 2 in the figure, as follows: In the first simulation cycle, by assuming the three signal-assignment statements to be correct, we can deduce the values for $y_2[1], y_1[1], \text{ and } y_0[1]$ from the specified value $O[1]$, and the initial values for $y$. In the same fashion, if we assume the signal-assignment statements in the second simulation cycle to work properly, and employ the values for $y$ obtained from the first simulation cycle, we know that $y[2] = '101'$. However, provided the remaining components of conflict 2 work properly, the cond input of the if-statement becomes true and thus the constant ‘000’ propagates to the output $z$, where we obviously encounter an inconsistency. Since no proper subset of this conflict causes this contradiction, we’ve identified a second minimal conflict.

In the next step we illustrate how to compute diagnosis candidates from the given conflicts. Our assumption that a specific component is faulty – that is, it behaves abnormal, allows for selecting its output value independent of the components’ inputs. In our example, we can choose the and operator’s output to be ‘1’ which obviously removes the given contradiction with respect to the output signal $O$. By assuming the subsequent signal-assignment statement of conflict 1 to behave abnormally we can establish consistency with the output $O$, too.

In order to get rid of the remaining inconsistency we have to assume at least a second component to work incorrectly. If we restrict our search for fault candidates to dual-fault diagnoses we solely can blame one further component to behave faulty because we already assume one component to work incorrectly in order to resolve conflict 1.

On a conceptual level, we can try to restore the contradiction by establishing an appropriate value for $y$, so that we can propagate the correct value to variable...
Figure 4. The diagnosis model for our running example.
z. Therefore, we require the value for \( y \) to become \('011'\) (see line 35 in Figure 2) or \('110'\) (line 37). However, for the given test-case, we neither can establish \( y \) to be \('001'\) nor \('110'\) by assuming just a single signal-assignment statement to behave incorrectly. Thus none of the signal-assignment statements of conflict 2 is a diagnosis candidate.

Nevertheless, all remaining components in conflict 2 are candidates. If we assume the \( =_{[2]} \) operator to behave faulty, we have to propagate the previous value of variable \( z \) alongside the else-paths of the if-statement chain. This yields the final value of \( z \) to be \('001'\) and thus removes the given contradiction. Furthermore, by assuming the constant \('000'\) to behave faulty, we can resolve the contradiction by assuming \('001'\) on this component’s output. The same holds for the output of the if statement and the variable assignment statement. Recapitulating this procedure we have obtained 8 dual-fault diagnoses: 

\[(\text{and},=), (\text{and},'000'), (\text{and},\text{if}), (\text{and},:=), (<=,=), (<=,'000'), (<=,\text{if}), (<=,:=).\]

These diagnosis candidates include the real cause of misbehavior, namely the pair \((\text{and},'000')\). However, note that we can obtain correct behavior for our design for the given test-case by appropriately modifying all statements of one of the remaining diagnoses.

4 Modeling for Design Error Diagnosis

Our modeling approach rests on the idea that evaluating a process’ statements relates a process’ inputs to its output. Thus, in order to compute a process’ output values, we have to know the process’ inputs. To create a debugging model we deduce a sequence of statement block. There is an edge from

(i) every port \((p, s)\) to the signal \(s\) of that port.

(ii) every signal \(s1\) to every port \((p, s2)\) whose signal \(s2\) depends on signal \(s1\).

Formally, we introduce the notion of functional dependencies [13] to capture the stated dependencies: An output \( out \) depends functionally on a set \( I \) if changing some input signals \( i \in I \) at time \( t \) may change the output signal \( out \) at time \( t' \geq t \). For example, consider the statement \( O \leftarrow \text{not} \ Q \). In this statement signal \( Q \) influences signal \( O \) since \( Q \) determines \( O \)’s value.

The Functional Dependency (FD) graph \( G = (V, E) \) is a directed graph where signals and ports represent vertices \( V \) and edges \( E \) represent dependencies. A(n) (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal. An output signal is a signal appearing on the left-hand side of a signal-assignment statement. A port is functionally dependent on the signals appearing in its sensitivity list and might depend on variables defined in the associated process’ sequential statement block. There is an edge from

\[V, E \text{ represent vertices } V \text{ and edges } E \text{ represent dependencies.} \]

\[A(n) \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]

\[A \text{ (output) port is an ordered pair consisting of a process and an output signal and represents a process writing to that signal.} \]
full circles denote ports. For the whole example we obtain 4 port nodes at level 0, a single port node at level 1, and 3 port nodes at level 2. For example, the output port \((O, p1)\) depends on \(CLK\) and \(I\) (since those variable appear in \(p1\)'s sensitivity-list) as well as on the variable \(S\) defined in the sequential statement part.

In the model creation process we start with those statements and expressions required to evaluate those ports that are one level deep. As a result of this evaluation, we obtain new signal values, which in turn, allow for computation of those ports that are two levels deep. Proceeding in this way, we create a diagnosis model by rearranging source code artifacts according to the data flow from primary inputs to primary outputs of the (naively) synthesized circuit.

For circuits with state the functional-dependency graph contains loops and thus the process of levelization fails. However, as proposed in [33], it is possible to modify the process to handle such designs.

For synchronous sequential circuits it is reasonable to assume that every feedback loop contains at least one synchronous flip-flop. Conceptually, the creation of debugging representation is broken into two phases, one in which latches change state, and one in which we create all the model fragments for the remaining gates (the purely combinational block). The outputs of the latches are treated as if they were inputs, and latch inputs are treated as outputs. Effectively, we split the design at the latches. Since every feedback loop must contain at least one synchronous flip-flop, the resultant circuit is combinational, and we can levelize the FD graph. Provided we assume a synchronous sequential design, we can identify such signals either by performing a syntactical analysis, an unoptimized synthesized representation, or by computing the so called golden cut set [14].

In VHDL designs variables represent a process’ local state explicitly. As the FD graph in Figure 5 indicates, due to variable assignments we may encounter cycles that do not contain a flip-flop. We can get rid of the the cycle by splitting variable \(S\) into two variables \(S'\), representing an additional input, and \(S''\) denoting an additional output. Besides of the declared output signals, for variables we always create an additional process output to store the variable value after process execution.

To compute conflicts and diagnosis candidates as we outlined for our example in Figure 4 we have to represent the model’s structure and the sub-component’s behavior in terms of rules that allow for forward (from inputs to outputs like in a classical simulator) as well as backward reasoning (from the outputs to the inputs). Denoting the inputs of the \(\text{and}\) operator with \(\text{in1}(\text{and})\) and \(\text{in2}(\text{and})\) and referring to the output by \(\text{out}(\text{and})\), we describe the correct behavior of the \(\text{and}\) operator in terms of propositional Horn clause like rules as depicted in Figure 6, where \(\neg AB(\text{AND}) = T\) denotes the assumption that the and operator works properly (or not abnormal). Note that the bracketed expressions in the figure denote single literals, e.g., \(\text{in1}(\text{and})\), is a single literal. Furthermore we explicitly add rules saying that such a literal cannot be true and false at any moment in time, e.g., for the and operator’s output signal we add the following rule, where \(\bot\) denotes logical contradiction: \(\text{out}(\text{and}) = T \land \text{out}(\text{and}) = F \rightarrow \bot\). This allows for computing conflicts in an efficient way. For example, if our test-case says that \(\text{out}(\text{AND}) = T\) and we can deduce \(\text{out}(\text{AND}) = F\), we know that this is a logical contradiction and together with the components assumed to be correct we’ve identified a conflict.

In similar fashion we encode RTL language artifacts like the if statements in Figure 4 in terms of Horn clause like rules: If the cond input evaluates to true, then the value at the components’ output out is equal to the input then, otherwise the output is equal to the else input. Logical facts represent correct behavior, for example, the literal \((\text{out}(\text{and}) = T) \rightarrow T\) represents that output of statement and must be true. Due to this our approach is very flexible in incorporating whatever correct behavior information a designer might be aware of. Furthermore, we associate a source-code location with every component, thus whenever the diagnosis computation procedure returns a specific statement, expression, or operator we directly can highlight the corresponding source code.

Notably, this representation allows for efficient forward and backward reasoning using a standard model-based diagnosis engine [6]. We developed similar models for other VHDL language artifacts like signal-assignment statements, all types of logical operators, and various types of conditional statements [34]. These value level sub-models have proven eligible in our previous modeling approach relying in VHDL’s
We arbitrarily introduced an error by substituting a randomly selected statement by another statement (e.g. substituting an \texttt{and} by an \texttt{or} operator). Then we created a circuit allowing for circuit equivalence checking with the model checker VIS [29]. This allows for obtaining an input sequence where the faulty and the correct circuit differ in at least one primary output’s value. We repeated this procedure until we obtained an input sequence with a length shorter than 6. This input sequence together with the faulty source is the input for our debugging tool.

For our empirical evaluation we recorded the number of diagnosis components building up our debugging model, the running-time and the number of candidates at the expression level. Furthermore, we checked whether the faulty statement as well as the signal-assignment statement assigning the faulty statement’s output to a signal are among the computed fault candidates. Regarding all our experiments this is always the case, Table 1 outlines the obtained results.

To compute the diagnosis candidates we employed Reiter’s approach [30]. For computing the conflict sets we used a version of the LTUR (Linear Time Unit Resolution) [21] algorithm which is optimized towards dealing with consecutive assumption sets [22]. We performed the evaluation on a Pentium 4/1.8 Ghz laptop computer using the VisualWorks Non-Commercial programming environment in version 5i.4. We didn’t devote a lot of effort in optimizing the diagnosis computation, thus there is potential to improve the running-times. Moreover, we computed the amount of source code a designer has to look on when trying to locate the fault.

The first column of Table 1 presents this amount in percentage of the whole number of expressions and statements in the VHDL-RTL code, column 2 lists the number of obtained fault candidates, column 3 shows the running time, column 4 outlines the length of the input sequence in terms of cycles, column 5 lists the total number of diagnosis components and the columns denoted with specification list the number of primary input and primary outputs. For each circuit we’ve repeated the abovementioned procedure for 3 times, thus the table lists three rows for every circuit. We provided the input values for every cycle, but provided the expected output values solely for the final cycle. We provided the input values for every cycle, but provided the expected output values solely for the final cycle. The last column outlines the number of D-type flip-flops in the circuit. The obtained results vary depending on the structural properties of the design and the discrim-
ination capabilities of the specific test case in both, the obtained number of candidates and the running-time. [22] presents a detailed empirical evaluation of the running-time of our diagnosis computation procedure.

In general MBD suffers from poor scalability with respect to the circuit's size and its structural complexity. In spite of this, our novel modeling approach reduces this problem to a reasonable amount so that we can cover larger VHDL designs. Concerning the outlined empirical results on average a designer can narrow the scope of faulty source-code artifacts down to 7 percent of the original program fully automatically. This means that we can exclude 93 percent of the statements and expressions of a given VHDL program fully automatically in a reasonable amount of time by employing solely a single test case. A designer thus can focus on localizing the faulty source-code artifact in the remaining 7 percent. However, different test cases revealing faulty behavior offer different discrimination capabilities, thus the quality of the diagnoses varies depending on the given design and test-case. De Kleer and Williams [7] introduce an algorithm that computes a so called probing strategy – that is, choosing the test-cases in a way that guarantees an optimal yield.

In practice, a single source of misbehavior is more probable than misbehavior caused by multiple erroneous sites. Thus we search for multiple-fault diagnosis only if there is no single-fault diagnosis removing a given contradiction. However, in MBD computing multiple-fault diagnosis is known as a computationally rather expensive task, particularly this holds for larger real world designs where the computational overhead gets prohibitively high.

To our best knowledge, an efficient computation of dual-fault diagnosis is only possible with simulation-based approaches like the gate-level approach introduced in [10]. Although MBD provides a well-founded theory for dealing with multiple-fault diagnosis [30, 7], our unoptimized algorithm for computing multiple fault diagnosis allows solely for computing dual-fault diagnoses for sequential designs up to a few clock cycles. However, even in these cases, compared to the computation of all single-fault diagnoses, the computational overhead increases dramatically. To illustrate this, we introduced 2 faults into the RTL level representation of the s382 circuit and computed all single and dual-fault diagnoses. After approximately one and a half hour of computation time we got 0 single-fault diagnoses (since introduced misbehavior is not correctable by modifying a single statement) and 106 dual-fault diagnoses including the real sites of misbehavior.

6 Discussion

Localizing faulty statements is inherently difficult partly because many diagnosis procedures treat designs as a black box controllable at the primary inputs and observable at the primary outputs. As pointed out by [31] due to this the solution search space grows exponentially with the number of faults and fault models. Motivated by the fact that a lot of diagnosis procedures conform to this observation – that is, we only can employ specification artifacts relating primary inputs to primary outputs, further improvements must focus on exploiting any kind of specification, regardless whether they relate primary outputs or other signals (in a possibly acyclic fashion) to the given inputs.

Our MBD approach directly supports this requirement and thus is very flexible in terms of incorporating correct behavior. Whereas the work in [27, 26] provides similar flexibility a well-founded and intuitively understandable theory supports our approach. The work presented in [27, 26] also focuses on VHDL and employs a hierarchical approach using stuck-at fault modes at the first level and an arbitrary failure model at the second level. Similar to our approach their technique also reasons from first principles, but to our best knowledge, the authors do not provide experimental results. Therefore it is impossible to evaluate whether their rather application specific, two-level approach outperforms ours in terms of running-time and the number of fault candidates.

Our empirical results differ from other published results in [11, 31] mainly in two points. First, we perform evaluation on non-optimized register transfer level representation and second, we compute possible fault locations at the expression level rather than on the level of gates.

The authors of [11] present experimental results for the ISCAS’89 benchmark suite. A reasonable comparison between their approach and ours is almost impossible since their approach employs numerous test cases.
<table>
<thead>
<tr>
<th>circ. name</th>
<th>diagnosis</th>
<th>running time [sec]</th>
<th>cyc. nr.</th>
<th>comp nr.</th>
<th>spec. nr.</th>
<th>inp.</th>
<th>outp.</th>
<th>FF nr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>s208</td>
<td>25.3</td>
<td>1.20</td>
<td>2</td>
<td>900</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18.2</td>
<td>1.06</td>
<td>2</td>
<td>900</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.4</td>
<td>1.28</td>
<td>2</td>
<td>898</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s344</td>
<td>7.7</td>
<td>21.07</td>
<td>3</td>
<td>2,031</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.8</td>
<td>10.15</td>
<td>3</td>
<td>2,028</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.1</td>
<td>36.72</td>
<td>4</td>
<td>2,704</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s349</td>
<td>6.1</td>
<td>22.39</td>
<td>3</td>
<td>2,058</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15.2</td>
<td>156.67</td>
<td>5</td>
<td>3,425</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18.4</td>
<td>146.10</td>
<td>5</td>
<td>3,430</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s382</td>
<td>4.5</td>
<td>1.05</td>
<td>2</td>
<td>1,562</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td>1.14</td>
<td>2</td>
<td>1,564</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.1</td>
<td>19.60</td>
<td>3</td>
<td>2,343</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s386</td>
<td>17.6</td>
<td>21.37</td>
<td>3</td>
<td>2,250</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16.9</td>
<td>21.08</td>
<td>4</td>
<td>3,000</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>4.75</td>
<td>3</td>
<td>2,250</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s444</td>
<td>4.1</td>
<td>3.08</td>
<td>2</td>
<td>1,804</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.1</td>
<td>21.70</td>
<td>3</td>
<td>2,706</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td>14.12</td>
<td>3</td>
<td>2,706</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s510</td>
<td>2.7</td>
<td>8.04</td>
<td>2</td>
<td>2,022</td>
<td>19</td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.0</td>
<td>49.58</td>
<td>3</td>
<td>3,030</td>
<td>19</td>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>41.7</td>
<td>395.97</td>
<td>4</td>
<td>4,044</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s526n</td>
<td>3.4</td>
<td>3.80</td>
<td>2</td>
<td>2,094</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td>2.16</td>
<td>2</td>
<td>2,090</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td>3.63</td>
<td>2</td>
<td>2,090</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s641</td>
<td>6.0</td>
<td>13.22</td>
<td>2</td>
<td>2,790</td>
<td>35</td>
<td>24</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.6</td>
<td>2.56</td>
<td>2</td>
<td>2,790</td>
<td>35</td>
<td>24</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.6</td>
<td>22.22</td>
<td>3</td>
<td>4,185</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s713</td>
<td>3.3</td>
<td>6.47</td>
<td>2</td>
<td>3,010</td>
<td>35</td>
<td>23</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.1</td>
<td>5.32</td>
<td>2</td>
<td>3,008</td>
<td>35</td>
<td>23</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>3.65</td>
<td>2</td>
<td>3,008</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s820</td>
<td>1.7</td>
<td>8.85</td>
<td>2</td>
<td>3,367</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.2</td>
<td>620.98</td>
<td>4</td>
<td>6,720</td>
<td>18</td>
<td>19</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13.8</td>
<td>759.74</td>
<td>4</td>
<td>6,720</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s832</td>
<td>6.0</td>
<td>22.53</td>
<td>2</td>
<td>3,392</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20.0</td>
<td>3,825.23</td>
<td>5</td>
<td>8,480</td>
<td>18</td>
<td>19</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.5</td>
<td>156.05</td>
<td>3</td>
<td>5,085</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s953</td>
<td>1.1</td>
<td>26.96</td>
<td>3</td>
<td>5,571</td>
<td>16</td>
<td>19</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.6</td>
<td>70.92</td>
<td>2</td>
<td>3,712</td>
<td>16</td>
<td>19</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.1</td>
<td>148.92</td>
<td>2</td>
<td>3,712</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1196</td>
<td>2.8</td>
<td>83.23</td>
<td>2</td>
<td>4,732</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>117.08</td>
<td>3</td>
<td>7,101</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>28.08</td>
<td>2</td>
<td>4,734</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1238</td>
<td>2.6</td>
<td>64.02</td>
<td>2</td>
<td>4,766</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>42.23</td>
<td>2</td>
<td>4,766</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.7</td>
<td>166.04</td>
<td>3</td>
<td>7,146</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1488</td>
<td>4.5</td>
<td>1,494.09</td>
<td>3</td>
<td>8,676</td>
<td>8</td>
<td>19</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>20.69</td>
<td>2</td>
<td>5,784</td>
<td>8</td>
<td>19</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.7</td>
<td>433.65</td>
<td>3</td>
<td>8,676</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1494</td>
<td>3.6</td>
<td>293.61</td>
<td>2</td>
<td>5,780</td>
<td>8</td>
<td>19</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>18.39</td>
<td>2</td>
<td>5,780</td>
<td>8</td>
<td>19</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>1,020.10</td>
<td>3</td>
<td>8,670</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| median    | 3.60      | 21.70               | 2        | 3,020    | 10       | 14   | 15    |
| average   | 7.08      | 74.69               | 2.71     | 3,844.9  | 13.5     | 13.24| 14.0  |

Table 1. Empirical results obtained from the circuits in the ISCAS’89 benchmark suite.
and the diagnosis procedure’s outcome in terms of the number of obtained diagnosis as well as running time inherently depends on the introduced fault, the length of the input sequence revealing the fault on the output, the optimization or decomposition techniques applied before diagnosis computation, and the used fault-modes. As with our approach, the real cause of misbehavior always is among the returned set of possible fault locations.

The satisfiability-based approach presented in [31] also employs optimized versions of the benchmarks and leverages recent achievements in boolean satisfiability techniques for computing diagnoses. Notably, the authors point out that a non-optimized version makes diagnosis harder because of redundancies in the circuit. In source level debugging we inherently deal with unoptimized representations because any optimization might removes redundancies and thus possibly fault candidates. Moreover, the authors of [31] claim that fault-mode-free diagnosis is a desirable characteristic since fault effects may have non-deterministic behavior.

However, model-free diagnosis is particular of interest when incorporating abstract domains like integers or arrays into our approach. For example, consider the multiplication of two integer numbers $A$ and $B$ assigned to a variable $C$. Moreover, we assume that a designer undeliberately writes $C <= A + B$ in place of the correct statement $C <= A \times B$. If ‘$A’$ is 2 and ‘$B’$ is 3 our test case says that ‘$C’$ must be 6 after statement execution. However, due to our fault, we obtain 5 instead of 6 thus something is wrong with our design.

Obviously, the set of components ‘$<=$’, ‘$+$’ is a minimal sub-model that causes contradiction, since solely assuming the ‘$+$’ operator and the signal-assignment statement to operate correctly causes contradiction. Assuming that the ‘$+$’ operator’s output value equals 6 apparently removes the given contradiction. The same holds for the signal-assignment statement, thus the ‘$+$’ operator as well as the signal-assignment statement are possible fault locations. However, in contrast to a boolean signal, where the value domain is restricted to 0 and 1, in this case we must consider a whole set of discrete output values, which might yield to combinatorial explosion. Hence, incorporating abstract domains in software debugging requires a diagnosis approach that doesn’t rely on fault modes, which further motivates the proposed approach.

Our approach furthermore exploits the design’s structural properties for identifying latched signals therefore we cannot directly apply it for debugging arbitrary behavioral design descriptions. In [20] the authors present a diagnosis model for the loop-statement in the context of debugging Java programs. However, this modeling approach, although feasible in principle, is rather expensive in terms of computational effort. In dealing with behavioral descriptions it thus is an open issue whether to incorporate such modeling approaches or employ a model representing a more abstract semantics. For example, a yet more abstract view is behavioral semantics [15] which merges sequences of states into so called super transitions resulting in temporal abstraction of the hardware behavior. Unlike to cycle semantics which we employed in our novel modeling approach, behavioral semantics does not specify a certain clock cycle a specific operation occurs on, but that it is performed sometime within a super transition. However, to our best knowledge there are no publications employing these semantics for software debugging of HDLs.

7 Conclusion

In this article we show how to apply the well-proven technique of model-based diagnosis to automated source-level debugging of VHDL designs. We present a novel modeling approach relying on functional dependencies and outline experimental results obtained from the well-known ISCAS’89 benchmark suite. Notably, the empirical results indicate that our approach is applicable to even larger designs and capable of localizing the faulty source-code artifact. Moreover, the approach is very flexible in incorporating knowledge about the design’s correct behavior, relies on fault-mode-free diagnosis procedure and thus supports debugging under presence of abstract domains such as integers or arrays.

8 Acknowledgments

The Austrian Science Fund (FWF) supports this work partially under project grant P15163-INF.
References


