Towards a Model for Automated Fault Localization in VHDL Designs: Exploring Counterexample-Traces Using a Model-Based Diagnosis Approach

Bernhard Peischl and Franz Wotawa[1]
Technische Universität Graz
Institute for Software Technology (IST)
Inffeldgasse 16b/2, A-8010 Graz, Austria
{peischl,wotawa}@ist.tu-graz.ac.at

Abstract. In this paper we discuss the exploration of a model checker's counterexample trace using model-based debugging techniques. We show that a diagnosis model obtained from a single counterexample run in an event-driven simulation is not appropriate for localizing a failures real cause in general. Notably, modeling VHDL's event and process semantics as originally defined hampers the integration of today's model checkers with our event-centered diagnosis approach considerably. Therefore, we propose a static but still event-centered and a data-driven approach for debugging hardware description languages. Both models do not exhibit the restrictions of the event-driven simulation approach with respect to integration of model checking tools.

1 Introduction

Detecting, localizing, and fixing faults is a crucial issue in today's fast paced and error prone development process. Detecting and repairing misbehavior in an early stage of the development cycle reduces costs and development time considerably.

Among the most popular techniques in detecting misbehavior of designs are model checking and testing. Today's testing procedures have reached a high state of maturity [2], and model checking [12] particularly helps in revealing the violation of properties even in unusual situations where bugs are difficult to detect. Both approaches, model checking and testing, have in common that they exploit a partial specification (a test case or a property) for detecting a possible fault.

Symbolic model checking automatically verifies whether a design satisfies a given property. The more traditional method of testing requires a test bench, that is, predefined input sequences and the associated intended outputs. In contrast to testing, in model checking properties are stated formally as assertions on the system. Adherence of the design to the specified properties is checked for all possible input combinations, ensuring full coverage of the design. If a program is not correct with respect to its specification, a counterexample is provided. Model checking is particularly useful to detect a misbehavior that is difficult to find because it occurs in unusual situations, which maybe neglected when constructing test benches.

For example, in model checking the tools usually give back a counterexample once a checked property is violated. This counterexample provides a concrete run of the program that leads to a situation where the property is no longer valid. When using this program execution (the executed statements are henceforth referred to as counterexample trace) the fault can be detected but this requires heavy user interaction and designers spend a lot of effort for the task of fault localization and fault correction. Therefore, tool support not only for detecting a fault but also for localizing the real cause of a failure deserves uttermost importance. Recently, the growing interest in not only detecting a fault, but also in localizing its cause automatically, has led to various approaches [1, 7].

Theses fault localization approaches exploit a model checker's abstraction in order to compute fault candidates and therefore must use properties for fault localization. Our approach is focused on model-based diagnosis [15, 5] and, although not specifically tailored towards model checking, we can integrate this approach. The model checker and our software debugger use different abstractions, which, as pointed out in this article, hampers smooth integration of both approaches depending on the used abstractions.

Intuition tells us, that at least one of the executed statements in a counterexample trace contains enough information for automated fault diagnosis. Notably, we point out that this intuition, under the presence of restricted specification (as usual when dealing with properties), turns out to have no firm computational grounds. The reason for this roots in exactly modeling VHDL's process and event semantics. We propose two different approaches to overcome this problem.

The first reflects the original, event-centered semantics of VHDL in a static fashion but its applicability is restricted to small to medium sized designs. The second, however, is an abstraction reflecting the data-flow in the design, and we expect this model to perform well even for bigger designs.

2 A VHDL Program Example

Suppose that we want to design a circuit according to the following specification. The circuit receives a stream of bits, one per clock cycle on input d_{in} and at every clock cycle it indicates, on the three outputs z_2, z_1, and z_0, the difference between the number of ones and the number of zeroes in the last three bits received. The difference is considered to be positive if the number of ones in the last three bits exceeds the number of zeroes and is negative otherwise. At reset, the circuit is assumed to have received an arbitrarily long stream of zeroes, so that the output is -3.

Proceeding informally, we may decide to built our circuit around a 3-bit shift register. This shift register holds the last three bits received, so that a combinatorial circuit can determine the number of ones and zeroes and produce the appropriate output encoded in form of the two's complement. Letting y_0, y_1, and y_2 be the three bits stored in the shift register, the truth table of the combinational circuit is

[1] Authors are listed in alphabetical order. The Austrian Science Fund (FWF) supports this work under project grant P15163-INF.

outlined in Table 2.

$y_2y_1y_0$	$z_2z_1z_0$		$y_2y_1y_0$	$z_2z_1z_0$
0 0 0	1 0 1		1 1 0	0 0 1
0 0 1	1 1 1		1 1 1	0 1 1
0 1 1	0 0 1		1 0 1	0 0 1
0 1 0	1 1 1		1 0 0	1 1 1

Table 1. The truth table for the combinational circuit.

Our initial problem's definition gives rise to a design whose structure is outlined in Figure 1. Although we know the intended behavior of the combinational logic block, we have to provide a running implementation for this block. As mentioned above, today this is done by using a hardware description language (HDLs). Figure 2 specifies the behavior of our solution in VHDL. Using this formal description we can do simulation, automatic verification, synthesis to a gate level representation and, as shown later on, we can also use the VHDL source code for fault localization. If we are interested in ver-

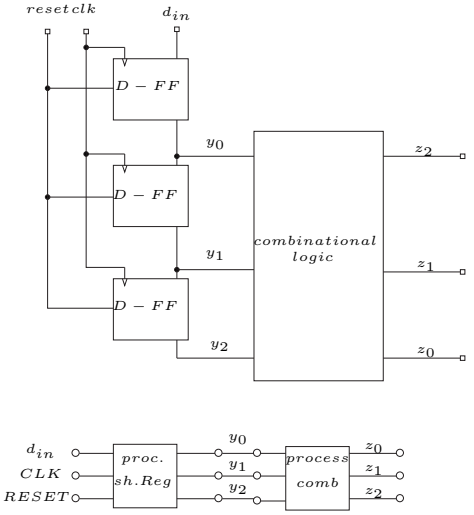

Figure 1. A structural view on the example circuit.

ifying that the implementation is correct, we need a non-ambiguous description of the intended behavior that can be deduced automatically from the VHDL code. One such description of our example program is the state transition graph of Figure 3. In compiling this diagram, we have assumed a synchronous circuit, that is, the values of the signals are updated whenever a clock ticks (i.e. the clock changes from 0 to 1). In Figure 3 the values of the state of the shift register $y_2y_1y_0$ is given in the circles and the input is given at the arcs. Alongside to the states, the corresponding output is given. From this diagram we can deduce a simple property of our circuit:

The least significant bit z_0 always is one. **(Property 1)**

The difference in the number of ones and zeroes in the last three bits of the input stream is always an odd number n. Encoding this number as the two's complement, that is, $n = -z_2(2^2) + z_1(2^1) + z_0(2^0)$, requires z_0 to be one.

A second property that a designer might be aware of can informally be stated as follows:

Assuming the input d_{in} to be one for a sufficient number of clock cycles (i.e., more than 2), the output of our design shall be $z_2z_1z_0 = 011$. **(Property 2)**

```
1  entity FSM is
2    port (
3       reset : in  bit;
4       clk   : in  bit;
5       d_in  : in  bit;
6       z0    : out bit;
7       z1    : out bit;
8       z2    : out bit);
9  end FSM;
10
11 architecture BEHAV of FSM is
12   signal y(3 downto 0): bit_vector;
13 begin
14   comb: process (y)
15
16     variable a, b, c, d, h, g, e : bit;
17   begin
18     b := y(1) nand y(2);
19     c := y(1) nor y(2);
20     a := not(y0) or c;
21     d := b or c;
22     h := a nand d;
23     e := not(c);
24     g := y(0) nor e;
25     z0 <= e or c;
26     z1 <= (b or not(y (0))) nand (h or g);
27     z2 <= not h;
28   end process comb;
29
30   shiftReg : process ( clk, reset )
31   begin
32     if reset = '1' then
33       y <= '000';
34     else
35       if (CLK'event and CLK = '1') then
36         y(2) <= d_in;
37         y(1) <= y(2);
38         y(0) <= y(1);
39       end if;
40     end if;
41   end process shiftReg;
42 end BEHAV;
```

Figure 2. The source code of our correct example circuit.

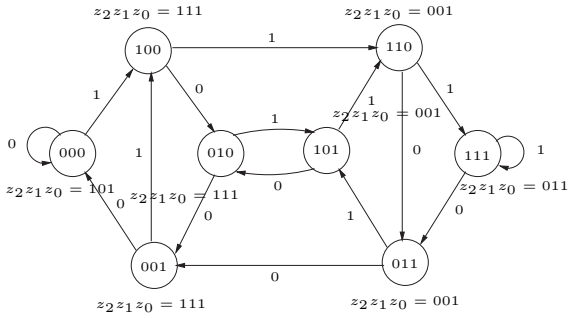

Figure 3. The state transition graph of our correct design.

This can easily be seen from the state transition graph in Figure 3. Moreover, we introduce a bug in line 26 of our program by substituting the nand operator by an and operator.

```
14      comb: process (y)
15
16    variable a, b, c, d, h, g, e : bit;
17    begin
18      b := y(1) nand y(2);
19      c := y(1) nor y(2);
20      a := not(y0) or c;
21      d := b or c;
22      h := a nand d;
23      e := not(c);
24      g := y(0) nor e;
25      z0 <= e or c;
26      z1 <= (b or not(y (0))) and (h or g);
27      z2 <= not h;
28    end process comb;
```

Figure 4 outlines the state-transition diagram of the modified program. We can use the above stated or similar properties to verify our

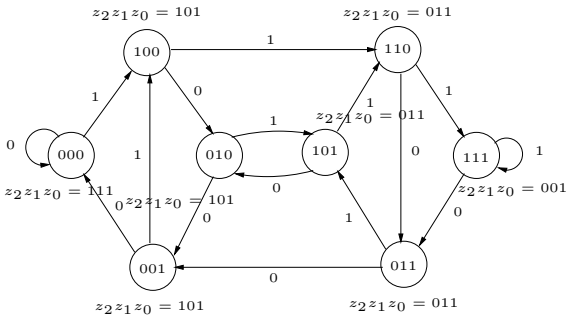

Figure 4. The state transition graph of the modified example.

design. If the properties are fulfilled we don't need to fix a bug. If the properties are violated, however, there is a bug in the VHDL program. For example, the faulty design fulfills the Property 1 but violates Property 2. The sequence of states $(000) \rightarrow (100) \rightarrow (110) \rightarrow (111)$ is a counterexample for Property 2.

Note that this counterexample gives no hint about the real cause of the misbehavior because both processes execute to compute the signals' values. However, the model we are going to introduce in the next section together with this counterexample can be used to automatically compute potential causes for the misbehavior.

3 Modeling for Fault Localization

The basic idea of model-based diagnosis (MBD) is to use knowledge of the correct behavior of the components of a system together with knowledge of the system's structure to locate the cause of malfunctioning. The behavior and the structure of the system are the model and the components of the system are the parts that behave either correct or abnormal.

In the domain of software debugging of hardware designs the model primarily reflects the syntax of the program and the semantics of the language. Considering VHDL designs, depending on the required granularity, components can either be statements, expressions, or processes and the model has to capture the underlying semantics of these artefacts at an appropriate level of abstraction.

In the past, models for different levels of abstraction have been developed. The so called functional-dependency model [6] represents concurrent statements as components and completely abstracts from individual values, referring to variables or signals merely as being correct or incorrect with regard to a given expected behavior. This representation allows for debugging VHDL designs up to 10MB of source code and is able to locate the faulty process, that is, the process in which at least one faulty statement occurs. In a more detailed model [18], expressions, sequential assignments, and concurrent-signal assignments represent diagnosis components. The behavior of those components is given by the semantics of the original statements or expressions allowing for fault localization at the statement and expression level. For all these models we can apply the same standard diagnosis engine to compute possible fault locations, i.e., the one that is based on Reiter's hitting set algorithm [15].

These models are successfully applied to combinational designs, however, recently we developed a novel model that also allows for debugging sequential designs. This is a prerequisite for integration of model checking and diagnosis since a counterexample usually comprises a sequence of inputs rather than a single input at a given point in time. The key to source-level debugging of sequential designs is the model for VHDL's process and event semantics, therefore we briefly introduce these language artefacts.

A design usually consists of several processes, a process itself is composed of statements, such as signal assignments or conditionals. Each process contains a so called sensitivity list enumerating those signals the process is sensitive to. Whenever at least one signal from the sensitivity list of a process changes its value, the process' statements execute and the resulting changes to its output signals propagate to other processes, possibly causing their execution in turn. Thus, processes communicate by means of signals. The parallel execution of the processes of different entities, resulting in the simulated behavior of the designed hardware unit, is performed by executing the VHDL program and recording the signal changes over time. Thus every VHDL program executes as a series of simulation cycles as illustrated in Figure 5. As outlined above, the different processes communicate by means of signals. [8] gives an overview of the VHDL language features and a definition of syntax and semantics and [13] provides an introduction into designing circuits with VHDL.

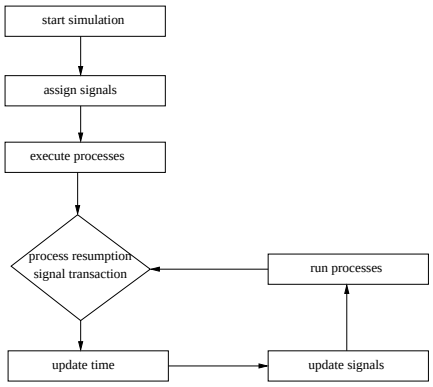

Figure 5. Simplified VHDL simulation cycle.

In the next step, we present our model for the components representing the process and event semantics, which is the key to source-level debugging of HDLs. According to the VHDL semantics, a process p executes if at least one of the signals occurring in its sensitivity list changes its value immediately before. If this is the case, the new values of $p's$ target signals (signals appearing on the left-hand side of an assignment statement in p) are computed taking into account the semantics of p's particular statements.

Formally we represented this computation by components associated with $p's$ sequential statements. Hence, the value of the signals are given by the values of the sub-block connected to $p's$ input; $in_y(p)$ represents the signal value after executing the sequential statement y whereas $def_y(p)$ denotes $p's$ corresponding unmodified

inputs. $s'EVENT$ denotes the corresponding event signal of signal s appearing in p's sensitivity list. If an output signal s changes, the corresponding event signal $s'EVENT$ is set to true. As illus-

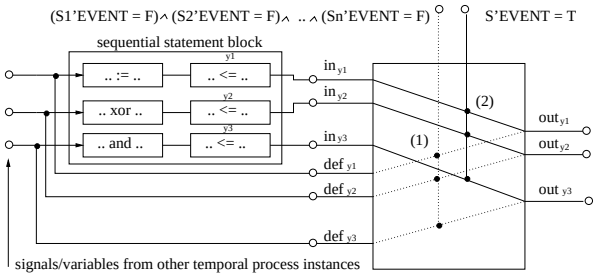

Figure 6. The model for the process component.

trated in Figure 6, if (1) none of the signals within the sensitivity list change their values, then the original input values before executing the sequential statement block propagate to the output of the process component; (2) otherwise the modified signal values propagate to the output. Thus, we have to built model fragments for paths (1) and (2) and add them to the model $m(p)$ for process p:

(1) for all input signals y from process p and all signals s appearing on p's sensitivity-list
add $(s'EVENT(p) = T) \rightarrow (out_y(p) = in_y(p))$ to $m(p)$

(2) let s_i denote element i in the sensitivity-list, $i = 1..n$; for all input signals y from process p
add $((s'_1EVENT(p) = F) \wedge (s'_2EVENT(p) = F) \wedge ... \wedge (s'_nEVENT(p) = F)) \rightarrow (out_y(p) = def_y(p))$ to $m(p)$

When equipping the model given above with $\neg AB$ predicates (stating whether component behaves abnormal or not), not only statements, but also whole processes may represent diagnoses. [18] outlines the modeling approach for other synthesize-able VHDL language artefacts.

4 Integrating Model Checking and Model-based Debugging

Given a specific counterexample we perform a simulation run and record the execution trace. According to this execution trace we built up a diagnosis model employing the above stated logical sentences representing a process' semantics. Note, that though a certain process triggers according to the simulation, we also create the logical sentences for path 2 in order to allow for best possible backward reasoning. Intuition tells us, that at least one of the executed statements must account for detected misbehavior and therefore the error trace cuts down the possible search space for faulty candidates considerably. However, as we show in the following, for models relying purely on VHDL's language semantics, this turns out to have no firm computational grounds.

The program outlined in Figure 7 consists of two processes $inv1$ (which is sensitive to signal X) and $inv2$ (which triggers on signal A).

Considering the correct design, we can immediately deduce a simple property:

If input X is '0' then output B is '0'. **(Property 3)**

For example, the setting $A = 0$, $B = 0$, and $C = 0$ violates Property 3. Considering this input, the execution trace of the correct design is as follows. According to the VHDL semantics processes $inv1$ and $inv2$ execute at simulation start, which yields both, A and

```
1       inv1 : process ( X)
2     begin
3       A <= not not X;   -- should be A <= not X;
4     end process inv1;
5       inv2 : process ( A)
6     begin
7       B <= not A;
8     end process inv1;
```

Figure 7. Line 3 contains the fault (doubled not operator).

B to reach value '1'. Although $B's$ value changes, no further process triggers since no process is sensitive to B. In contrast, the value change of A causes execution of $inv2$ which in turn brings $B's$ value to '0' again. Figure 8 outlines the execution trace of the correct design.

In the faulty design, since doubling the not operator causes A to remain at value '0', no further process executions take place. This, in turn, prevents a data flow from signal A to B, as it happens in the correct design. However, since we only are aware of a partial specification, i.e., we know that B must be '0' instead of '1',we need a continuous path from X to B containing the real fault. As Figure 9 outlines, this is not the case. In terms of model-based diagnosis solely $inv2$ is a minimal conflict, and the real cause of misbehavior, namely $inv1$, is no element of the conflict. This small example

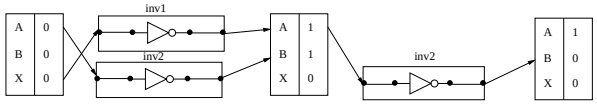

Figure 8. The execution trace for the correct design.

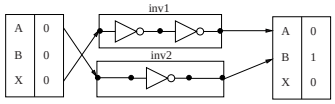

Figure 9. The execution trace for the faulty design.

shows that debugging models purely relying on a single error trace from an event-based simulation cannot localize a failure's real cause under the presence of a restricted specification. Consequently, we cannot build-up a diagnosis model for debugging VHDL as a byproduct of an event-driven simulation run for a given counterexample.

In order to overcome the stated problem we outline two approaches. The first approach directly represents VHDL's event and process semantics, but instead of focusing on a single execution trace, it covers all possible paths statically. Although this model is only applicable for small to medium-sized designs, a case study indicated promising results [14]. Second, we propose a data-driven abstraction resembling the synthesis process. However, this approach does not reflect VHDL's execution semantics directly rather it rearranges the program's statements in a way that reflects the data flow and thus the gate arrangements in the synthesized circuit.

Our first approach assumes that the process activation graph is acyclic. Considering this assumption and a VHDL program that comprises n processes, each process can at most be activated by $n - 1$ other processes. Taking into account the initial activation for every

process, there are at most $n + n \times (n-1) = n^2$ process activations per simulation cycle. For small designs, we can create a temporal instance of a process for every possible activation. However, this model consumes a lot of memory and thus is only applicable for small to medium-sized designs or parts of a bigger design. In Section 5 we outline the obtained results for this event-driven model for our running example.

A data-driven modeling approach overcomes both, the problem of combinational explosion of the static approach above and that caused by VHDL's execution semantics. Like the event-driven static approach this model is applicable to sequential circuits but reflects the data flow from primary inputs (and latch outputs) to primary outputs (and latch inputs) rather then VHDL's execution semantics.

We deduce a sequence of statements representing the data flow of the synthesized circuit by analyzing static dependencies between signals. For example, considering the design in Figure 1, we have to compute signals y_0, y_1, and y_2 before we are able to compute the values for z_0, z_1, and z_2, since, for example, z_0 is dependent on y_1 and y_2. This relationship reflects a data-flow driven ordering of statements that is suitable for building up a diagnosis model. Figure 1 also lines out the conceptual relationship between the structure of the circuit (reflecting the syntactic properties of the design) and the order of statement execution.

In order to deduce the appropriate ordering of statements we use functional dependencies [9]. A functional dependency is defined as:

Definition 4.1 (Functional Dependency) *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 $t' \geq t$. We say that out depends on I and write $(out, \{I\})$.*

For example, consider the statement `O <= not Q` . In this statement signal Q influences signal O since Q determines O's value, thus the functional dependency relation is given by $(O, \{Q\})$. Moreover, we define a functional dependency graph representing functional dependencies.

Definition 4.2 (Functional Dependency Graph) *The Functional Dependency (FD) graph $G = (V, E)$ is a directed graph where signals and ports represent vertices and edges represent dependencies. A (output) port is an ordered pair consisting of a process and an output signal. An output signal is a signal appearing on the left-hand side of a signal-assignment statement. Note that processes may have several output ports. There is an edge from*

(i) every port to the signal depended on that port.
(ii) every signal to every port depending on that signal.

Note that there are only edges from port nodes to signal nodes and from signal nodes to port nodes. There are no edges connecting nodes of the same type.

The process of levelization allows for deducing an appropriate order of statement execution and thus it induces a topological ordering relation on ports, therefore we give the following definition:

Definition 4.3 (Logic Levelization) *A logic levelization of a FD graph is an ordering relation on port nodes. The level of a port is the maximal number of port nodes that are passed on any path between a primary input and that port.*

Figure 10 outlines the FD graph for our running example. In the figure blank circles denote signals and full circles denote ports. In addition ports are leveled according to Definition 4.3. Once all port nodes are levelized, we sort them in ascending order by level number. Since the obtained sequence of process executions directly reflects the data flow in the synthesized circuit, there is always a path from the input to the erroneous output that contains the failure's real cause.

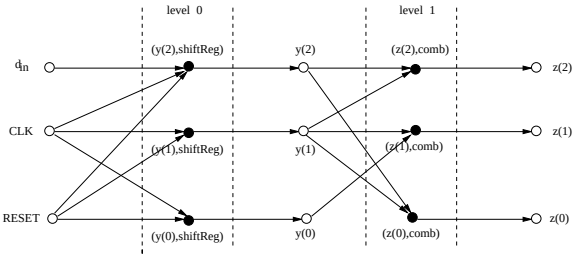

Figure 10. The functional-dependency graph of our design.

This approach can be generalized in order to handle possible loops in the FD graph. For example, [17] outlines an algorithm for levelizing circuits at the gate-level and also discusses how to get rid of possible loops.

We intend use the data-driven approach for larger sized designs since we expect that it scales very well with the circuit's size. For our running example we outline the results obtained from the static, event-based modeling approach.

5 Localizing the Faulty Statements

We now show the outcome of the fault localization process for our running example. When testing Property 2, a model-checker may provide the following counterexample:

$$CLK = [010101], RESET = [000000], d_{in} = [111111], z_2 z_1 z_0 = 001$$

By using the counterexample with our event-driven diagnosis model we applied Reiter's diagnosis algorithm and obtained 27 single-fault diagnoses. In summary, after mapping the temporal instances back to the corresponding components the computed fault locations with respect to this specific counterexample correspond to the statements 14, 18, 26, 30, 32, 35, 36, 37 and 38 and include the introduced bug. The statements 14, 30, 32 and 35 correspond to a process or conditional statements. From the remaining statements 18, 26, 37 and 38 the statements 18 and 26 correspond to functional faults, i.e. wrong operators in the code. Furthermore, in addition to the bug in line 26, we altered line 27 in the following way:

```
24          .....
25          z0 <= e or c;
26          z1 <= (b or not(y (0))) and (h or g);
27          z2 <= not(not (h));
28      end process comb;
```

Now our program contains two bugs and after assuming the input d_{in} to be one for at least 3 clock cycles the output corresponds to $z_2 z_1 z_0 = 101$ and thus violates Property 2. Therefore, our counterexample is suitable for locating both causes of misbehavior. Considering this example we obtained 508 diagnoses, thereof 448 dual-fault diagnoses and 60 single-fault diagnoses. After mapping back the temporal instances to the corresponding source code locations we obtained 185 fault locations , 172 thereof correspond to dual-fault diagnoses including the introduced bugs, and the remaining 13 correspond to single-fault diagnoses. Recognizing the second `not` operator in line 27 to be faulty and therefore searching for those dual-fault diagnoses that contain exactly this operator reduces the number of possible locations that can explain the misbehavior with respect to the given counterexample to 6. Likewise, if we recognize that the other part of the real cause is faulty, that is the `and` operator in line 26, and filter out all the diagnoses containing this operator, in total 12 faulty components can explain the misbehavior. These results indicate that the static, event-driven approach can localize the statements that are responsible for a specific counterexample.

6 Related Work

Zeller's Delta Debugging is a technique for minimizing error trails that works by conducting a modified binary search between a failing and a succeeding run of a program. The technique is extended to an approach to automatic software debugging that includes the modification of parts of the program's state to isolate cause effect chains [19].

Ball, Naik, and Rajamani [1] find successful paths to the control location at which an error is discovered in order to find the real cause of the failure. Once a cause is discovered, a restricted model in which the system is prevented from executing the causal transitions to discover whether other causes for the same error are possible.

Groce and Visser [7] attempt to extract information from a single counterexample produced by model checking in order to facilitate the understanding of malfunctioning systems. The key to their approach is to find multiple variations on a single counterexample.

Algorithmic program debugging as originally introduced by Shapiro [16] is an interactive process where the debugging system acquires knowledge about the expected behavior of the debugged program and applies these knowledge for fault localization. The knowledge is collected by the system through a number of questions and thus requires heavy user interaction. In [10] the authors discuss the automatic diagnosis of VLSI circuit designs using algorithmic debugging.

Chung and colleagues [3] introduce a different approach to fault localization. Although, they use the logical description of hardware designs directly for fault localization, there work is different to ours in several respects: Chung and colleagues introduce specialized algorithms. The hardware designs has to be described at the gate level which is below the register-transfer-level. The algorithms can only find single faults in the design whereas MBD is not restricted to single fault diagnoses.

Zanella and Lamperti [11] present a diagnosis technique subsuming two complementary approaches to diagnosis of discrete-event systems. The proposed technique copes with a generalized class of discrete-event systems and deals with synchronous as well as asynchronous behavior. At the current state of research it is open whether these techniques can be incorporated in our approach.

Cordier and Largouët [4] demonstrate how to exploit model checking techniques for diagnosing dynamic systems represented by discrete event models. Rather than focusing on exploring the counterexample trace this work shows how model checking techniques can perform a diagnostic task.

7 Conclusion

Model-based diagnosis can be used effectively for locating faults in hardware designs. The idea behind the application of MBD to software debugging is to compile the program, i.e., the VHDL design, into a logical description, and use this description together with the specification of the program directly for diagnosis. Although, several models of VHDL programs have been published so far [6, 18] none of them really captures the full semantics of VHDL.

In this article we show that an event-driven, simulation-based debugging approach, that exactly captures VHDL's process and event semantics, hampers the integration of model checking and model-based debugging techniques considerably.

The event-driven model we described in the paper fully captures the semantics by unfolding the process executions over time. The used sub-models for the processes are still extracted from the source code but the rest of the model, i.e., the connections of this models over time are created at run-time. Intuition tells us, that this is not a limitation of the approach since the process comprising the erroneous statement is executed when simulating the VHDL design. This must be the case in order to produce an observable contradiction to a given specification. As shown in this paper, for representing the exact semantics of VHDL in the diagnosis model, this intuition turns out to have no firm computational grounds.

Consequently, we rely on a static but still event-driven model to represent VHDL's original semantics precisely. Our running example indicates the applicability of this diagnosis model for small to medium-sized designs. The proposed data-driven model overcomes both, the problem of state explosion and those of VHDL's event and process semantics, and shows that the coupling of model-checkers and MBD techniques for automated source level debugging is at least in reach on a conceptual level.

Usually, an abstraction like the proposed data-driven model, contains spurious paths to possible faulty statements. At the current state of research, it is unclear whether, in comparison to the data-driven model, the event-driven approach to source-level debugging of VHDL designs offers benefits in terms of quality of the obtained diagnoses. Hence, this area is subject of further research.

References

[1] T. Ball, M. Naik, and S.K. Rajamani, 'From symptom to cause: localizing errors in counterexample traces', in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium of programming languages (POPL)*, pp. 97–105. ACM Press, (2003).

[2] Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1990.

[3] Pi-Yu Chung, Yi-Min Wang, and Ibrahim N Hajj, 'Logic design error diagnosis and correction', *IEEE Transactions on VLSI Systems*, **2**(3), (1994).

[4] Marie-Odile Cordier and Christine Largouët, 'Using model-checking techniques for diagnosing discrete-event systems', in *Proceedings of the 12th International Workshop on Principles of Diagnisis (DX-01)*, pp. 39–46, (March 2001).

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

[6] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa, 'Model-based diagnosis of hardware designs', *Artificial Intelligence*, **111**(2), 3–39, (July 1999).

[7] A. Groce and W. Visser, 'What went wrong: Explaining counterexamples', in *10th International SPIN Workshop on Model Checking of Software*, (5 2003).

[8] IEEE, *IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987*, 1988.

[9] Daniel Jackson, 'Aspect: Detecting Bugs with Abstract Dependences', *ACM Transactions on Software Engineering and Methodology*, **4**(2), 109–145, (April 1995).

[10] Kuchcinski Krzysztof, Drabent Wlodzimierz, and Maluszynski Jan, 'Automated diagnosis of VLSI digital circuits using algorithmic debugging', in *Proceeding of the first International Workshop on Automated and Algorihmic Debugging*, ed., Peter Fritzon, pp. 350–367, (1993).

[11] Gianfranco Lamperti and Marina Zanella, 'Continous diagnosis of discrete-event systems', in *Proceedings of the 14th International Workshop on Principles of Diagnosis*, Washington DC, USA, (June 2003).

[12] Kenneth L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993. ISBN 0-7923-9380-5.

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

[14] Bernhard Peischl and Franz Wotawa, 'Modeling state in software debugging of vhdl-rtl designs - a model based diagnosis approach', in *Proceedings to the 5th International Workshop on Automated and Algorithmic Debugging (AADEBUG 2003)*, Ghent, Belgium, (September 2003).

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

[16] Ehud Shapiro, *Algorithmic Program Debugging*, MIT Press, Cambridge, Massachusetts, 1983.

[17] L.-T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio, 'Ssim: a software levelized compiled-code simulator', in *24th ACM/IEEE conference proceedings on Design automation conference*, pp. 2–8. ACM Press, (1987).

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

[19] Andreas Zeller, 'Isolating cause effect chains from computer programs', *ACM SIGSOFT Software Engineering Notes*, **27**(6), 1–10, (11 2002).