Using Filtering to Improve Value-Level Debugging of Verilog Designs

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

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

Naveed Riaz
Shaheed Zulfikar Ali Bhutto Institute
Islamabad, Pakistan
n.r.ansari@szabist-isb.edu.pk

Abstract— **In this article, we report on novel insights in model-based software debugging of hardware description languages (HDLs). Our debugging model allows one for exploiting failing and passing test cases by incorporating Ackermann constraints. This article reports on an empirical evaluation of the introduced models. The evaluation of our approach on the well-known ISCAS 89 benchmarks concerning single and dual-fault diagnoses clearly indicates that incorporating passing test cases into fault localization improves considerably the accuracy of the obtained diagnosis candidates.**

Keywords – hardware/software debugging, model-based debugging, source-level debugging, fault localisation

I. Introduction

This article reports on the most recent results in software debugging of Verilog designs. It is a major extension to previous research work that primarily reports on fault localization in Very High Speed Integrated Hardware Description Language (VHDL) [1]. Verilog [2], has a formal semantics and thus, it is amendable to research in verification and debugging, e.g., its synthesis semantics is formally specified in Gordon [3].

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 for more than two decades. In the recent past, numerous test cases have been employed for localizing faults, e.g., in terms of employing spectrum-based diagnosis [4, 5, 6, 7, 8].

Spectrum-based techniques, however, allow one for logical reasoning at the level of dependencies and do not consider the semantics of the language in terms of value-level models. Consequently, there is a lack of research dealing with multiple test cases in conjunction with value-level models taking into account language semantics. This is noteworthy as we do have well-founded techniques that allow for considering whole test suites and – as shown in this article – there is solid empirical evidence that taking into account test suites improves the fault localization capabilities considerably.

Over the last 25 years, the Artificial Intelligence community has developed a framework for system diagnosis called model-based diagnosis (MBD). This framework is extremely general and covers a broad range of capabilities, including the isolation of faulty components and the handling of multiple fault locations [9, 10]. Harnessing these techniques in software engineering tools, may help considerably to master the development of complex circuits and software-enabled systems.

Since its well-founded theory, we rely on MBD, and employ the ISCAS 89 benchmark suite [11] to demonstrate the practical applicability of our novel models. Relying on an exhaustive evaluation, our insights clearly indicate that the incorporation of test suites (rather than only single test cases as for example in [12]) considerably contributes to locate accurately the root cause for detected misbehavior. According to our empirical evaluation using the ISCAS 89 benchmarks, with a couple of failing test cases (up to 5), we can exclude almost 94 percent of the statements and expressions of being faulty. By leveraging passing test cases, we can further rule out around half of the remaining 6% of the potentially erroneous code. In this article, we show how to incorporate passing test cases. In contrast to previous articles addressing this issue, we report on our most recent empirical evaluation on the ISCAS 89 benchmarks regarding the proposed filtering algorithm.

The next section gives a brief introduction to simulation, test and debugging of HDLs and afterwards (Section III), we discuss the debugging of sequential circuits. In Section IV, we show how to exploit passing test cases. Section V reports on practical experiences and the evaluation of the approach and Section VI concludes this article.

II. Simulation, Test and Debugging

In designing circuits, a 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 amendable 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 are typically conducted under time pressure and thus, the time for debugging – detecting, localizing, and repairing the misbehavior – is a critical process measure.

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 MBD for fault localization in HDL designs1.

According to a study conducted at IBM Haifa, 50 to 80 percent of the overall development is attributed to verification activities, and localization and correction amounts to 35 percent of the design cycle [13]. 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 Sequenital Verilog Designs

The semantics of Verilog has been analyzed rigorously, and thus provides the necessary theoretical underpinning in language semantics and circuit synthesis. Gordon [3] 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 states at the end of a simulation cycle. For computing these quiescent values, each procedure is evaluated only once per cycle [3]. Procedures are evaluated in an order such that a procedure is not evaluated until all its driving procedures have been evaluated. In other words, the outputs of a procedure are computed only when all its inputs are known (or already 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 [12]. In this way, we build up a chain of procedures and their inputs and outputs, thus allowing 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 other words, synchronous sequential circuits consist of multiple cycles. In electrical engineering, sequential circuits are often viewed as a sequence of connected combinational circuits. This can be done by selecting some connections and splitting them in two separated connections. One is the input and one the output. 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 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 component-connection model. This component-connection model [9, 10] represents our model at level 1 (cycle no. 1). For every component C, we attach a timestamp i during the creation of the model to ensure a unique identification. Thus Ci 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. So we create n instances for each component.

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

$$SD = \bigcup_{i=1..n} SD_i \text{ where } SD_i \text{ is the system descry. for cycle i} \quad (1)$$

$$COMP = \bigcup_{i=1..n} C_i \text{ where } C_i \text{ are the components in cycle i} \quad (2)$$

and

$$OBS = \bigcup_{i=1..n} OBS_i \text{ and } OBS_i \text{ denote the obs. in cycle } i. \quad (3)$$

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:

Diagnosis problem, test suite: Given a test suite comprising the test cases $TC_1, TC_2, ..., TC_k$. 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, the sets 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:

$$SD^* = \bigcup_{j=1..k} \begin{matrix} SD_j \cup \{\neg AB(C_0{}^j) \rightarrow \neg AB(C_1^j) \wedge \\ \neg AB(C_2^j) \wedge ... \quad .... \wedge \neg AB(C_n^j)\} \end{matrix} \quad (4)$$

$$COMP^* = \bigcup_{j=1..k, i=0..n} C_i^j \quad (5)$$

$$OBS^* = \bigcup_{j=1..k, i=1..n} OBS_i^j \quad (6)$$

As passing testcases do not cause a logical contradiction, we do not obtain conflicts from passing testcases considering the diagnosis model for a test suite (SD*, COMP*, OBS*).

IV. Exploiting Passing Testcases

To illustrate the potential of using passing test cases to locate the root cause for detected misbehavior we continue with a simple example.

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

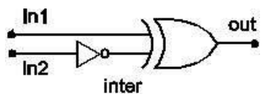

Figure 1: Passing and failing testcases and part of a circuit.

Figure 1 illustrates a part of a circuit an exclusive or and a NOT gate together with a passing and failing test case. We further assume that the circuit is faulty, that is, our test suite has identified misbehavior and we obtain both components (the exclusive OR and the NOT gate) as possible diagnosis candidates.

Suppose we have the test cases given in Figure 1. Considering the first (failing) test case in the first line, and assuming the NOT gate to be abnormal but the exclusive OR gate to be correct, we can deduce that signal inter becomes 0. However, under the same assumption, the passing test case in line 2, forces the value of inter to become 1. We immediately see that the NOT gate is required to map the signal inter to 0 and to 1 for the same input value in2=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 [14] 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 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 does not yield to additional conflicts, we capture the specific information about diagnoses in terms of Ackermann constraints [22, 23]. 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 for exploiting the many test cases that typically do not reveal a fault. The system description with Ackermann constraints SDA is given as follows:

System description with Ackermann constraints: Let TC_p be a set of passing test cases form a test suite TC, let $in(C_i) = \{ i_{Ci}^1, ..., i_{Ci}^m \}$ denote the inputs of component C_i, let $out(C_i) = \{ o_{Ci}^1, ..., o_{Ci}^n \}$ 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,

where, i≠j and i,j denote indices of the passing test cases. As we will show in the next section, Ackermann constraints increase the complexity of the model considerably.

$$SD_A = SD^* \cup CON_A, \quad (7)$$

$$CON_A = \neg AB(C_i) \wedge \forall_{l=1}^m i_{ci}^l = i_{cj}^l \rightarrow \forall_{p=1}^n o_{ci}^p = o_{cj}^p \quad (8)$$

Therefore, we used a post processing technique proposed by the authors of [21]. 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 for repairing the program for the given test case. The function *replace(C)* allows 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 $D \in \Delta$ do
2. For all test cases $TC_i \in TC$ 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
```

Figure 2: Exploiting passing testcases via filtering.

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 conclude that

$$\neg\exists i, j, i \neq j \bullet (\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \wedge (\forall_{p=1}^{n} o_{Di}^{p} \neq o_{Dj}^{p}) \qquad (9)$$

$$\forall i, j, i \neq j \bullet \neg(\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \vee (\forall_{p=1}^{n} o_{Di}^{p} = o_{Dj}^{p}) \qquad (10)$$

$$\forall i, j, i \neq j \bullet (\forall_{l=1}^{m} i_{Di}^{l} = i_{Dj}^{l}) \rightarrow (\forall_{p=1}^{n} o_{Di}^{p} = o_{Dj}^{p}) \qquad (11)$$

Algorithm 1 (Figure 2) thus imposes the Ackermann constraints on the set of single-fault diagnosis candidates. Therefore, for our approach evaluation we therefore took advantage of the filtering algorithm presented previously.

V. PRACTICAL EXPERIENCES AND EVALUATION

With a series of our most recent experiments we pursue the goal to evaluate the discriminating capabilities of several test cases on sequential circuits, the response time (and thus the computational complexity on a technical level) and the effect of the filtering technique.

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 in [15]. Note that, for the efficient computation of diagnoses, we convert the rules capturing the language semantics (discussed in [16]) into a specific Horn-like encoding [17]. As the computation of conflict sets is a time critical issue, the (minimal) conflict sets are computed according to the procedure explained in [17].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 [16]. 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 Diagnosis

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 [17]. From the obtained conflicts, we retrieve diagnoses by computing the minimal hitting sets in increasing order, where for practical purposes, primarily single- and double-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 [18]. Since we use two fault modes (AB(C) and ¬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 size (particularly single-fault diagnoses) are more likely than diagnoses with bigger size. For example, finding all single diagnoses is of order O(|COMP|) assuming the decision procedure can be executed in unit time.

B. Test Suite Generation

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. In some (rare) cases, for example for the circuit s444, we were not able to find five test cases and stopped this process earlier (see Figure 3). 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. Furthermore, we implicitly removed/added negations as we substituted a logical statement by the negated counterpart (e.g., NAND by AND vice versa). These error types are not necessarily complete wrt. functional errors, but as they are believed to be common in the design process, we capture the most com-

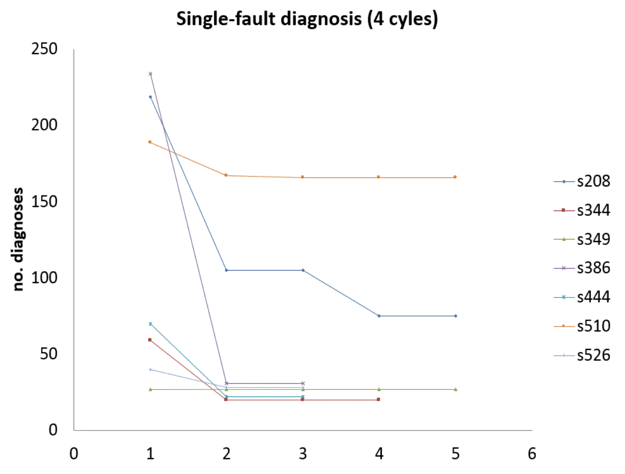

Figure 3: No. of obtained single-fault diagnoses for the ISCAS 89 benchmark (4 cycles).

mon scenarios [20]: (1) Mistakenly replacing one gate by another gate with the same number of inputs and (2) incorrectly adding or removing a gate.

All empirical evaluations are conducted on the Verilog RTL version of the ISCAS 89 benchmark suite [11]. 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. A detailed analysis including the results for the specific circuits can be found in [16]. In the following, we summarize the major results. In this article, we summarize the work presented in [16] and present novel results regarding the incorporation of passing tests alongside with first empirical results.

C. Empirical Evaluation and Discussion

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, the traced variables correspond to the primary inputs vin for every instant of time (vin, valin, t), t=1..n, together with the primary outputs (vout, valout, n) at time n and thus, 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). To evaluate the impact of the temporal unfolding of the circuit, we conducted experiments with four and eight simulation cycles relying on the well-known ISCAS 89 benchmark suite.

First, the figures underpin the findings discussed in previous research papers [19]. The number of single diagnoses being obtained depends from both, the concrete test case being applied and the structural complexity of the program being considered. Second, as Figures 3 and 4 illustrate – even with only a couple of test cases (in our case up to 5) – the number of obtained diagnoses can be reduced significantly when com-

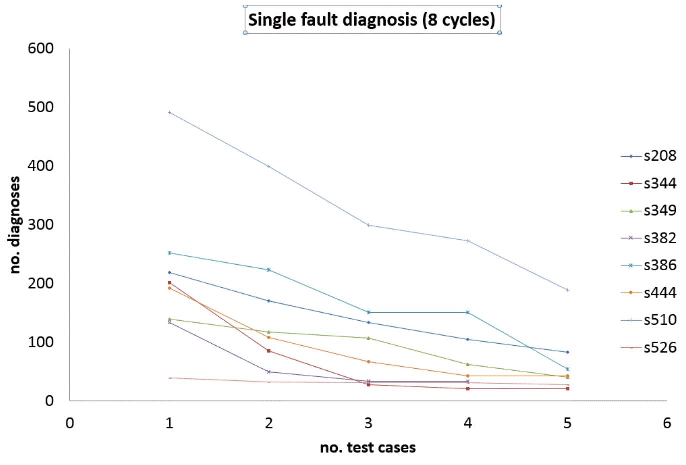

Figure 4: No. of obtained single-fault diagnoses (ISCAS 89, 8 cycles).

pared to the experiment with solely a single test case. 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 specific error being introduced and the (3) specific test cases identifying the introduced faults result in a (at least in relation to the other circuits) computationally expensive problem. On average, we obtained 74(123) single-fault diagnoses and 44(70) faulty lines in the source code when unfolding the circuit for 4(8) instances of time. Remarkably, a designer can exclude over 90 percent of the source code from being faulty (93,6 percent for 4 cycles and 92,5 percent for 8 cycles of unfolding).

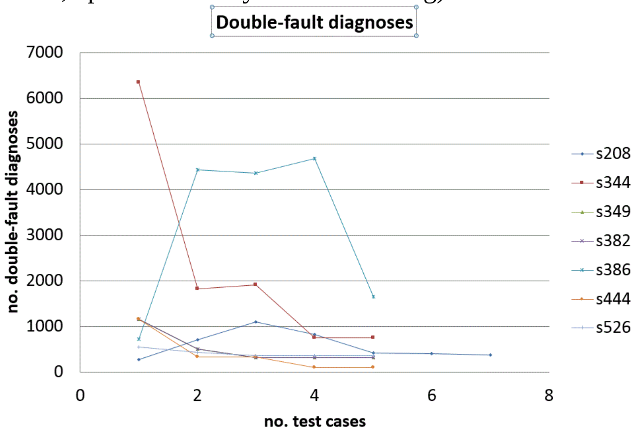

Figure 5: No. of obtained dual-fault diagnosis (ISCAS 89, 4 cycles).

Figure 5 outlines further empirical results. We obtained these results from the ISCAS 89 benchmark suite considering dual-fault diagnoses as well. When considering dual-fault diagnoses, the no. of diagnosis candidates does not necessarily decrease monotonically with the increasing set of test cases.

However, our experiments revealed that for most of the circuits, the obtained number of fault candidates decreases monotonically with an increase in the size of the test suite. Together with the results for single-fault diagnoses, this gives empirical evidence that the additional cost in the running time, pays off in terms of a higher accuracy in the obtained diagnosis candidates. In [15], we present novel algorithms and an analysis on scalability and the corresponding running times.

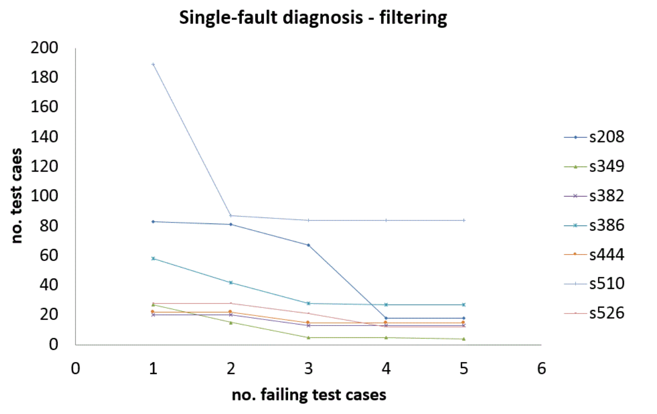

Figure 6: No. single-fault diagnoses when using the filtering algorithm (4 cycles).

Figure 6 summarizes the results on a further series of experiments incorporating the filtering algorithm. To our best knowledge, the filtering approach has never been subject to an empirical evaluation. When compared to Figure 3, one can see that exploitation of passing test cases contributes to accurately isolate the real cause of misbehavior.

VI. Conclusion

In this article, we briefly discuss the simulation-driven design process with hardware description languages (HDLs) and point out the importance of fault localization techniques. Afterwards, we introduce a model extension that allows one for exploiting failing and passing testcases. Failing testcases results in conflicts, and thus it contributes to locate the fault in an accurate manner. To exploit the numerous passing test cases, we introduce Ackermann constraints and establish a relationship to the filtering technique proposed earlier. Our empirical evaluation on the ISCAS 89 benchmark suite demonstrates that the proposed technique is practically feasible and considerably contributes to locate the real cause of misbehavior. According to our experiments using the ISCAS 89 benchmarks, on average, we can exclude almost 94 per cent of the statements and expressions from being faulty making use of up to 5 failing test cases per circuit. By leveraging passing test cases, we are able to rule out around half of the remaining 6 per cent of the potentially erroneous code. These results motivate research on value-level models for debugging HDL designs. Future research should apply the proposed techniques to even bigger circuits (e.g., using more recent benchmarks, etc.) and investigate the relationship between filtering and Ackermann constraints under presence of multiple-fault diagnoses.

References

[1] Z. Navabi, VHDL Analysis and Modeling of Digital Systems, McGraw-Hill, New York, 1993.

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

[3] M. J. C. Gordon, “Relating event and trace semantics of hardware description languages”, The Computer Journal, 45(1), 2002, pp. 27–36.

[4] R. Abreu, P. Zoetewei, van A. J. C. van Gemund, “On the Accuracy of Spectrum-based Fault Localization”, Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007, TAICPART-MUTATION 2007, vol., no., 10-14 Sept. 2007, pp. 89-98.

[5] B. Baudry, F. Fleurey, Y. Le Traon, “Improving test suites for efficient fault localization”, In Proceedings of the 28th international conference on Software engineering (ICSE '06), ACM, New York, NY, USA, 2006, , pp. 82-91.

[6] D. Hao, L. Zhang, T. Xie, H. Mei, and Jia-Su Sun, 2009, “Interactive fault localization using test information”, J. Comput. Sci. Technol. 24, 5, September 2009, pp. 962-974.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan, “Scalable statistical bug isolation”, In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05), ACM, New York, NY, USA, 2005, pp. 15-26.

[8] Y. Yu, James, A. Jones, M. J. Harrold, “An empirical study of the effects of test-suite reduction on fault localization”, In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 2008, pp. 201-210.

[9] R Reiter, A theory of diagnosis from first principles, Artif. Intell. 32, April 1987, pp. 57-95.

[10] J. de Kleer, A. K. Mackworth, R. Reiter, “Characterizing diagnoses”, In Proceedings of the National Conference on Artificial, Intelligence (AAAI), Boston, Aug. 1990, pp. 324–330.

[11] F. Brglez, D. Bryan, K. Kozminski, “Combinational Profiles of Sequential Benchmark Circuits”, IEEE International Symposium on Circuits and Systems, 1989, pp. 1926-1934.

[12] B. Peischl and F. Wotawa, “Automated Source-Level Error Localization in Hardware Designs”, IEEE Design and Test of Computers, January/February, 2006, pp. 8-19.

[13] G. Auerbach, M. Moulinn, B. Jobstmann, R. Bloem, A. Cimatti, M. Roveri, PROSYD: Property-Based System Design, Deliverable 2.1/1, May 2005, PROSYD Technical Report, FP6-IST-507219.

[14] O. Raiman, J. de Kleer, V. Saraswat, and M. Shirley, “Characterizing non-intermittent faults”, In Proceedings AAAI, Anaheim, Morgan Kaufmann, July 1991, pp. 849–854.

[15] B. Peischl, N. Riaz, F. Wotawa, “Advancements in Automated Debugging of Verilog Designs”, Submission to the Applied Artificial Intelligence Journal in preparation.

[16] B. Peischl, N. Riaz, F. Wotawa, “Automated Debugging of Verilog Designs”, International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Sept. 2012, Vol. 22, No. 5, World Scientific, pp. 695-724.

[17] B. Peischl and F. Wotawa, “Computing Diagnosis Efficiently: A Fast Theorem Prover for Propositional Horn Theories”, In Proceedings of the 14th International Workshop on Principles of Diagnosis (DX-03), Washington DC, June 2003, pp. 175-180.

[18] F. Wotawa, Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages, PhD thesis, Technische Universität Wien, 1996.

[19] G. Friedrich, M. Stumptner, F. Wotawa, “Model-based diagnosis of hardware designs”, Artif. Intell. 111(1-2), 1999, pp. 3-39.

[20] D. Nayak, D. M. H. Walker; “Simulation-based design error diagnosis and correction in combinational digital circuits”, VLSI Test Symposium, Proceedings of the 17th IEEE Test Symposium (VIS 99), 1999, pp. 70-79.

[21] F. Wotawa, “Debugging hardware designs using a value-based Model”, Applied Intelligence, 16(1), 2002, pp. 71–92.

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

[23] S. Staber, G. Fey, R. Bloem, and R. Drechsler, “Automatic fault localization for property checking”, In E. Bin, A. Ziv., and S. Ur, editors, Second International Haifa Verification Conference (HVC 2006), Haifa, Israel, October 2006, Springer-Verlag, LNCS 4383, pp. 50-64.