Automated Debugging from Pre-Silicon to Post-Silicon

Mehdi Dehbashi*
*Institute of Computer Science, University of Bremen
28359 Bremen, Germany
Email: dehbashi@informatik.uni-bremen.de

Görschwin Fey*†
†Institute of Space Systems, German Aerospace Center
28359 Bremen, Germany
Email: goerschwin.fey@dlr.de

***Abstract*—Due to the increasing design size and complexity of modern *Integrated Circuits* (IC) and the decreasing time-to-market, debugging is one of the major bottlenecks in the IC development cycle. This paper presents a generalized approach to automate debugging which can be used in different scenarios from design debugging to post-silicon debugging. The approach is based on model-based diagnosis. Diagnostic traces are proposed as an enhancement reducing debugging time and increasing diagnosis accuracy. The experimental results show the effectiveness of the approach in post-silicon debugging.**

I. Introduction

The cost of VLSI systems verification and debugging has significantly increased in the recent years as design size and complexity has increased. Also due to time-to-market constraints, 100% verification coverage at the design level is an elusive task. Thus, in addition to electrical bugs, design bugs may appear in the final IC product. In this case, automated debugging approaches handling different kinds of bugs can effectively help to reduce the development time of IC products.

Verification aims at deciding if there is an error in a system with respect to a specification. The system is an implementation of a design in pre-silicon verification, and an IC in post-silicon verification. If there is an error, the erroneous behavior is returned as a counterexample. Having a counterexample, debugging is responsible to localize and rectify the root cause of the erroneous behavior. This process often remains as a manual task and increases the time of the development cycle significantly. Automated debugging identifies the potential sources of the observed errors corresponding to the counterexamples. Each potential source of the errors is a *fault candidate* which can fix all erroneous behavior of the available counterexamples.

Different approaches have been proposed for automating pre-silicon and post-silicon debugging. Automated approaches in pre-silicon debugging rely on simulation [1], *Binary Decision Diagrams* (BDD) [2], and *Boolean Satisfiability* (SAT) [3]. In [4], SAT-based debugging is used to debug different abstraction levels of the system description. The post-silicon debugging requires a larger effort. The main challenge of post-silicon debugging is the limited observation of internal signals. To overcome this problem, various on-chip solutions for internal signal observation have been proposed such as scan chains [5] [6] and trace buffers [7] [8] [9]. The techniques based on the trace buffers are widely accepted in the industry [9] [10]. Even by using the trace buffers, getting an execution trace of the on-chip signals related to the time of bug activation is a challenging problem. To address this problem different methods for different kinds of bugs have been proposed. In [10], the approach re-runs the chip with new trigger conditions to "backspace" the content of the trace buffer until the traces related to the activation time of the design bug can be extracted. In [11] and [12], some quick error detection mechanisms for the electrical bugs are used to efficiently store the erroneous behavior related to the bug activation time in the trace buffer.

This work has been funded in part by the German Research Foundation (DFG, grant no. FE 797/6-1).

The works in [13] and [14] use randomly generated test patterns to obtain more counterexamples for automating debugging and to apply automatic correction. Automatic correction increases the computational cost and is not guaranteed to fix an error in the desired way. Using random counterexamples may decrease the diagnosis accuracy, and may increase the iterations between verification and debugging. In [15], a pre-silicon debugging flow is proposed for testbench-based verification environments. The approach uses diagnostic traces to obtain more effective counterexamples and to increase the diagnosis accuracy.

Here we propose a generalized framework to automate debugging that tightly integrates model-based diagnosis using Boolean Satisfiability [16] and diagnostic trace generation [15]. The main contributions of this paper are:

- a unified view on pre- and post-silicion debugging automation and
- a detailed discussion of the post-silicon debugging scenario

Our approach relies on model-based diagnosis as an underlying step. Diagnostic traces [15] close the loop between verification and model-based diagnosis. Diagnostic traces differentiate the fault candidates and increase the diagnosis accuracy. The debugging flow can be applied to electrical bugs as well as design bugs which slip into the IC from different levels of the system description. An instantiation of the generalized automated debugging flow is applied to post-silicon debugging of design bugs.

The remainder of this paper is organized as follows. Section II introduces preliminary information on hardware structures for post-silicon debugging and model-based diagnosis. Then, our generalized approach to automate debugging is presented in Section III. Section IV describes our automated debugging for post-silicon to diagnose different kinds of bugs. A concrete instantiation of our approach for design bugs is presented in Section V. Section VI presents experimental results on benchmark circuits. The last section concludes the work.

II. Preliminaries

A. Hardware Structures for Post-Silicon Debugging

The hardware structures for post-silicon debugging are divided into two main categories: *Design-For-Test* (DFT) Structures and *Design-For-Debug* (DFD) structures. Scan chains are commonly used as a DFT structure in manufacturing test. This hardware can be reused for post-silicon debugging [17].

During the test mode, the state of all the scan registers can be extracted by performing a scan dump. Unless two-state elements are used for each register, which leads to an excessive area overhead, the test environment needs to be restarted after each scan dump [18]. The scan registers with two-state elements are used in [19] for online detection of design bugs.

An overview of DFD structures is given in [7]. Trace buffers are commonly used as a DFD structure in industry. A trace buffer is based on an on-chip memory which records internal signals. The trace buffer includes control logic which is responsible to trigger on-line monitoring of circuit behavior. Once the trigger condition is asserted by control logic, the trace buffer can start/stop recording the selected signals values [18]. The trace buffer size in practice is typically 1K $\times$ 8 bits to 8K $\times$ 32 bits [20].

B. Model-based diagnosis (MBD)

Model-based diagnosis (MBD) is a precise approach which is frequently used to localize bugs in hardware and software [21] [22]. In MBD, a system model is provided in terms of components and their interconnections [23]. The component models describe how each component behaves. Then a domain-independent reasoning engine calculates the diagnosis from the model and system observations. SAT-based reasoning engines have been shown as a robust and efficient approach to diagnose and localize the bugs, called SAT-based debugging [3].

In SAT-based debugging, a circuit is divided into components. Depending on which elements are chosen as components, the granularity of the debugging result differs. Typical choices are gates or expressions, but also hierarchical or structural information are taken into account [24] [25]. SAT-based debugging searches for all possible fault candidates in the circuit. Given an implementation of a circuit and a set of counterexamples, one copy of the circuit is created for each counterexample. Then, the inputs and outputs are constrained to the input stimuli and to the correct output response of the corresponding counterexample. Also the circuit is enhanced with correction logic by adding a multiplexer at the output of each component. The original output function F_c of component C is replaced by F'_c. The select line S_c of the added multiplexer controls F'_c such that if S_c is activated $F'_c = R_c$ where R_c is an unconstrained variable and a value for correcting the erroneous behavior may be injected, otherwise $F'_c = F_c$. The select line is also called *abnormal predicate*. The number k of active abnormal predicates is controlled by a fault cardinality constraint.

To reduce the space requirement, instead of running the SAT solver on the all counterexamples, SAT solver can be run consecutively on the counterexamples [3]. In this way, the memory consumption is independent of the number of counterexamples.

Debugging for sequential circuits is done by unrolling the circuit for some time steps equal to the length of the counterexample [3]. The correction logic is added as in the combinational case and usually the same abnormal predicate is used for the same gate in all time steps and for all counterexamples.

III. Generalized Automated Debugging Procedure

Here we present a generalized *Automated Debugging* (AD) procedure which uses MBD and diagnostic traces for automation and accurate localization of potential root causes of an error. AD can be reused in different contexts for various debugging situations. [1] Diagnostic traces help automated debugging by distinguishing fault candidates.

The diagnosis in a system starts with system observations violating normal system behavior determined by the system specification. This discrepancy between a system and its specification is called counterexample. Having an initial counterexample, MBD tries firstly to localize fault candidates, i.e., components capable of rectifying the erroneous behavior. But usually the initial number of fault candidates is large. Thus, diagnostic traces are introduced to discriminate the fault candidates and to help debugging to accurately localize the root cause of the error. More erroneous behavior is discovered by diagnostic traces and this new behavior should be used to iterate MBD for excluding fault candidates that cannot fix the new erroneous behavior.

The inputs of the AD function include a system specification as a reference (Ref), a system model as an object (Obj), and one or more initial counterexamples (CEs) which show the initial discrepancy between the system and its specification. The output of AD is the set FCs of fault candidates which are the potential sources of the observed error corresponding to the available counterexamples. Each fault candidate is a set of components of the system which can fix all erroneous behavior of the counterexamples under consideration:

$$FCs = AD(Ref,\, Obj,\, CEs) \tag{1}$$

The AD function includes three subfunctions. The first subfunction of AD applies MBD. MBD finds the initial set FCs of fault candidates as potential sources of the observed errors according to the initial counterexamples:

$$FCs = MBD(Obj,\, CEs) \tag{2}$$

Diagnosis accuracy is a function of fault candidates, e.g., a small number indicates good accuracy. In general, the quantity and quality of fault candidates determine the *Diagnosis Accuracy* value. If the diagnosis accuracy determined by fault candidates is not sufficient, then the second step of AD starts. The second step is *Diagnostic Trace Generation* (DTG). Similar to diagnostic test patterns [27] [28], diagnostic traces distinguish the behavior of fault candidates [15]. DTG works on sequential circuits and does not require a precise fault model. Section V-B discusses more details. Diagnostic traces may help the next debugging session to exclude fault candidates which cannot fix all erroneous behavior:

$$DTs = DTG(Obj,\, FCs) \tag{3}$$

Afterwards, diagnostic traces are validated with respect to the system specification to guarantee that the diagnostic traces really create a discrepancy between the system and its specification. A diagnostic trace which creates a discrepancy is a counterexample. This step is called *Diagnostic Trace Validation* (DTV):

$$CEs = DTV(Ref,\, DTs) \tag{4}$$

[1]Actually, a concept similar to diagnostic traces is used not only for debugging in computer science but also in other sciences (e.g. medical science, psychology, ...) to accurately discover disorders in clinical cases (as fault candidates) out of a community (as a system) [26].

```
1  function $AD(Ref, Obj, CEs)$
2  do
3  {
4      $FCs = MBD(Obj, CEs)$
5      $NewCEs = \emptyset$
6      if $DiagnosisAccuracy(FCs) < Threshold$ then
7      {
8          while $NewCEs == \emptyset$ and $!DTsLimitation$ do
9          {
10             $DTs = DTG(Obj,\ FCs)$
11             $NewCEs = DTV(Ref,\ DTs)$
12         }
13         $CEs = CEs \cup NewCEs$
14     }
15 } while $NewCEs! = \emptyset$
16 end function
```

Fig. 1. Generalized Automated Debugging Procedure

Finally, the new counterexamples are used by iterating the process to decrease the number of fault candidates. Figure 1 shows the automated debugging function as a pseudo code. AD is controlled by a parameter for the threshold of the *Diagnosis Accuracy* (line 6), e.g., the number of fault candidates can be used as a value for the threshold parameter. The number of diagnostic traces is controlled by the parameter *DTsLimitation* (line 8).

To apply the generalized approach to pre-silicon debugging, Obj is a hardware design at the RTL or the gate level. Ref can be a formal specification or a testbench. Initial counterexamples CEs are given by design verification tools. In this case, AD is invoked to search the set FCs of fault candidates such that the appropriate diagnosis accuracy is achieved [15]:

$$FCs = AD(Spec,\ Design,\ CEs) \tag{5}$$

When Ref is a formal specification, the functions DTG and DTV can be integrated in a unified instance to generate new counterexamples [29]. However, a formal specification is often not given for complex designs. In the following, we explain how AD is utilized in post-silicon debugging.

IV. Automated Post-Silicon Debugging

This section describes how the general automated debugging procedure is used for post-silicon debugging of design bugs and electrical bugs. In the IC design hierarchy, there can be multiple references for a chip as a system. By using different references, different kinds of bugs are distinguished. As shown in Figure 2, for post-silicon debugging usually there are three main descriptions of the hardware system: specification, design, and chip.

The specification as a golden model can be a formal specification or a high level simulation model or a testbench. The specification is used for creating the expected correct output of a trace in the debugging process. A design is a circuit which is represented at RTL by *Hardware Description Languages* (HDL). Then, the gate level and the transistor level designs are created respectively by logic synthesis and the place-and-route processes for chip manufacturing.

After the chip is manufactured, post-silicon validation is started by running a test program, such as an end-user application or functional tests, or applying the test vectors. An error may be observed by hardware or software assertions. In this case, signal traces are stored in trace buffers. The content of the trace buffer is used to extract the system state, its inputs, and the corresponding outputs. Then, the specification and the design are used to check the extracted traces. There are

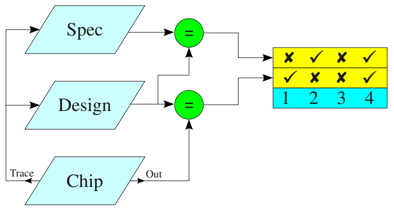

Fig. 2. The start of debugging after an error detection on a chip. $\times$ shows the inconsistency between the corresponding responses in each level.

some shared observation points among specification, design, and chip which can be assumed as a partial state equivalence for debugging. Our general idea is based on inconsistencies detected between different system descriptions which can be used to distinguish different bugs. As Figure 2 shows, after an error occurrence, the traces and their corresponding responses are extracted from the trace buffer. Checking the extracted traces from the trace buffer in the specification and in the design leads to four cases. For each case, the general AD function is configured in different ways to efficiently diagnose bugs. In the following, we discuss each case.

A. Case 1: Design Bug

In this case (Figure 2, case 1), the extracted traces applied to the design create responses which are consistent with the extracted responses from the trace buffer. Applying the traces to the specification and the design creates different responses. This shows there is a bug in the design which has escaped pre-silicon verification and has slipped into the chip. Actually, the extracted traces show a special sequence of the system traces which was not verified in the pre-silicon verification. This is revealed after running the chip for a longer time and in communication with the peripherals. In this case, erroneous behavior obtained from trace buffer data (chip) or design, and expected behavior obtained from specification constitute a counterexample. The AD function localizes the bug on the design:

$$FCs = AD(Spec,\ Design,\ CEs) \tag{6}$$

In this case, diagnostic traces are validated by the specification.

B. Case 2: Electrical Bug

The extracted responses from the trace buffer are not reproduced in the design while the responses of the design and the specification are consistent (Figure 2, case 2). As the silicon is assumed to implement the RTL correctly, in this case, there is an electrical bug which can be reproduced neither in the design nor the specification. The erroneous behavior obtained from trace buffer data (chip) and the expected behavior produced by the design or the specification constitute one or more counterexamples. Then, the AD function operates on the design (as RTL or gate level) to localize the root cause of electrical bugs:

$$FCs = AD(Chip,\ Design,\ CEs) \tag{7}$$

For electrical bugs, fault candidates are found on the design as this allows to access the internal structure of the circuit. Also there is a mapping between the design and IC components. Diagnostic traces generated by DTG are checked by being applied to the chip. For electrical bugs, the difficulty

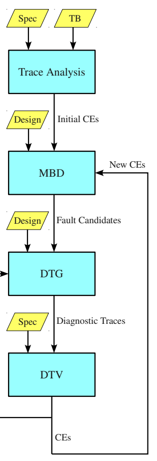

Fig. 3. Automated post-silicon debugging of design bugs

is to generate diagnostic traces which can reactivate the bug. Diagnostic traces can be generated by considering the suspected electrical bug type (e.g. drive strength, coupling, antenna effects, ...) and the layout information. Diagnostic traces can be applied instead of randomly generated traces [14] to improve the debugging performance. A diagnostic trace which creates an inconsistency between the chip and the design is a counterexample. There are different methods for applying the diagnostic traces to a chip. Diagnostic traces can be applied to a chip by hardware structures like scan-chains or wrappers allowing what-if analysis [7]. In microprocessor-based systems, *Software-Based Self-Testing* (SBST) methods are effectively used to apply test patterns [30]. In [31], a set of instructions, called *Access-Control Extensions* (ACE), are defined and used to access and to control the microprocessor's internal state. ACE instructions are used to run directed tests on the hardware.

C. Case 3: Electrical Bug and Design Bug

This case is a rare case in practice. When the extracted traces create different behaviors in the design in comparison to the extracted behavior of the chip, an electrical bug has occurred. In the case (Figure 2, case 3), also the extracted traces create different behavior in the design and the specification which shows a design bug. Here debugging the design bug and the electrical bug can be performed independently and in parallel. Calling the functions defined by Equations (6) and (7) can discover the root causes of the design bug and the electrical bug concurrently:

$$\begin{cases} FCs_D = AD(Spec,\, Design,\, CEs_D) \\ FCs_E = AD(Chip,\, Design,\, CEs_E) \end{cases}$$

D. Case 4

After the error detection on a chip, the extracted traces from the trace buffer create no inconsistency in any level. In this case (Figure 2, case 4), the erroneous behavior related to the bug activation time may not be stored in the trace buffer and is overwritten. To overcome this problem some approaches have been proposed in the cases of electrical bugs and the design bugs. In [10], to get and to backspace an execution trace of on-chip signals for many cycles leading up to the activation time of a design bug, the chip is re-run with new trigger conditions. In each run, the state of the trace buffer is dumped out. This procedure is repeated automatically until the traces related to the activation time of the design bug can be extracted. In [11] and [12], quick error detection and localization mechanisms are used to efficiently debug the electrical bugs.

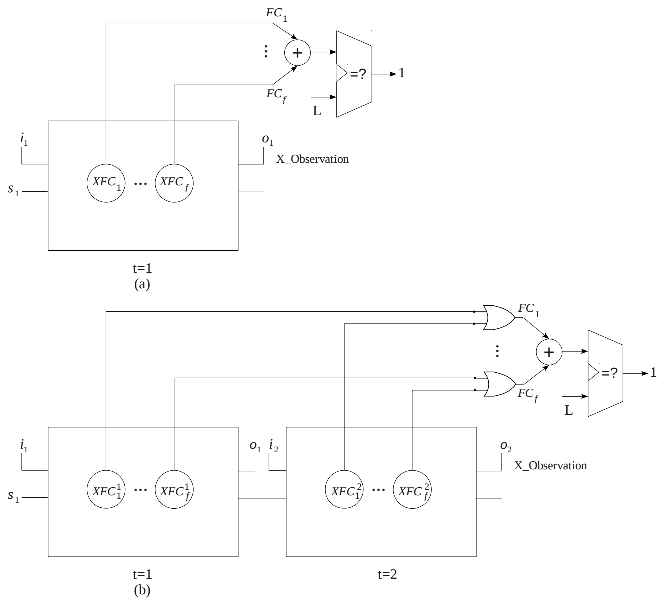

Fig. 4. Diagnostic Trace Generation method for sequential circuits with single faults: (a) first time step. (b) second time step

Another situation which may result in case 4 is a software bug or a bug related to hardware/software integration. The bugs in this state may be distinguished by running N-version programs (like software fault tolerant systems) and analyzing their behavior on software and hardware assertions.

V. Automated Post-Silicon Debugging of Design Bugs

In this section, we focus on case 1 when a design bug occurs in the system. Exemplarily we present AD for post-silicon debugging. Section V-A describes our automated flow for post-silicon debugging which integrates data analysis of the trace buffer, model-based diagnosis, and diagnostic trace generation to be used for increasing the accuracy of debugging. In Section V-B, we present a heuristic method for diagnostic trace generation which increases the diagnosis accuracy of automated debugging.

A. Automated Flow for Post-Silicon Debugging

Figure 3 shows the overall approach which consists of four steps. These steps are data analysis of the trace buffer, model-based diagnosis, diagnostic trace generation, and diagnostic trace validation. In the first step, trace buffer data which is obtained after running a test program on the chip should be analyzed and compared with the expected correct outputs obtained from the specification. As we consider only design bugs, the design and the chip have same behavior. If there is an inconsistency between trace data and golden data, this inconsistency or erroneous behavior represents a counterexample. By having the initial counterexample, the generalized AD (Equation (6)) is invoked to execute MBD, DTG, and DTV (Equations (2)-(4)). Here we use SAT-based debugging as an effective approach to MBD. DTV is performed by applying the diagnostic traces to a simulation model or a formal model of the specification. DTG is explained in the following.

TABLE I
DIAGNOSIS ACCURACY

Method		Initial Result		**Heuristic Method**			**Random Method**		
Circuit	#C	#FC	#CE	#FC	#CE	#Trace	#FC	#CE	#Trace
or1200_alu	436	5	1	3	5	41	3	7	153
or1200_ctrl	1865	7	6	7	13	12	7	6	2000
or1200_genpc	732	20	3	**12**	8	296	20	3	2000
or1200_if	463	5	3	**1**	9	6	5	3	2000
or1200_lsu	793	3	2	**2**	9	10	3	7	2000
or1200_operandmuxes	376	11	3	**7**	8	62	11	3	2000
or1200_wbmux	278	7	10	7	15	13	7	10	2000

B. Diagnostic Trace

A diagnostic trace is an input stimulus which tries to activate a fault candidate (or a set of fault candidates) and propagate its behavior to the outputs. The work in [15] presents a heuristic technique which generates diagnostic traces by only having fault candidates and a faulty circuit. The technique considers the X-value as a token for the behavior of one fault candidate or a set of fault candidates. Firstly the token X is injected at a fault candidate in the faulty circuit. Then, the algorithm searches for the diagnostic traces in a way that the token X is propagated from inputs, crosses only the considered fault candidates and arrives at outputs. This procedure is iterated for all fault candidates. After this step, there are some diagnostic traces which can discriminate the behavior of all fault candidates. If the set of fault candidates and the number of fault candidates having the value X are denoted by FCs and L respectively, the algorithm firstly tries to find the diagnostic traces which satisfy the following formula when $L = 1$:

$$\sum_{i=1}^{|FCs|} (FC_i == X) = L \qquad (8)$$

If there is no diagnostic trace with one fault candidate having X which can create a counterexample, the algorithm tries to find the diagnostic traces with more fault candidates having X until at least one counterexample is found. The algorithm continues until L is equal to the number of fault candidates. The diagnostic traces generated by this heuristic do not necessarily find counterexamples decreasing the number of fault candidates. But the experimental results show that in most of the experiments they are effective to decrease the number of fault candidates.

For sequential circuits, firstly the circuit is unrolled for some time steps. In this case, each $FC_i \in FCs$ has one component in each time step: $FC_i = \{FC_i^1, FC_i^2, \ldots, FC_i^s\}$, where s is the number of time steps. The erroneous behavior of a fault candidate can be propagated to outputs by each component or by a combination of components. This behavior is modeled by inserting one OR-gate for each fault candidate to control the components of the fault candidate. To clarify this method, Figure 4 shows a sequential circuit with a single fault where the number of fault candidates is represented by $|FCs| = f$. Figure 4(a) considers one time step which is similar to a combinational circuit. Figure 4(b) shows two time steps where the circuit is unrolled two times. For each fault candidate, there is one OR-gate. The inputs of the OR-gates correspond to the variables of fault candidates specifying the X value. The output of the OR-gates are added and constrained to L.

VI. EXPERIMENTAL RESULTS

This section presents the effect of automated post-silicon debugging of design bugs (as described in Section V) on diagnosis accuracy, time, and memory. The hardware structure is written at RTL with Verilog hardware description language. The experiments are evaluated on the modules of the OpenRISC CPU from OpenCores [32]. A matrix multiplication program is used as a test program to be run by OpenRISC in ModelSim environment. The experiments are executed for each module independently. For each experiment, a random single functional bug (wrong assignment, incorrect case statements, etc) is inserted into the RTL code. The trace data is recorded in the trace buffer of the corresponding module such that for a time window with 8 cycles we have initial states at the first step of the window, inputs, and output results at the end of window. The size of the trace buffer is different for different modules, but the maximum size of the trace buffer is assumed to be 8K $\times$ 32 bits.

For specification, here we consider each bug-free Verilog module as a black box module with access only to module inputs, module outputs, and some internal registers (state bits) which would typically be available in a high level specification, too. For each time window, the recorded initial states and inputs are applied to the specification. Then, output results are compared to the output results of the corresponding window in the trace buffer to detect inconsistencies and to constitute initial counterexamples. After having initial counterexamples, the buggy RTL design is unrolled for 8 time steps for MBD.

The experiments are carried out on a Quad-Core AMD Phenom(tm) II X4 965 Processor (3.4 GHz, 8 GB main memory) running Linux. MiniSAT is used as underlying SAT solver [33]. The techniques described in the paper are implemented using C++. Run time is measured in CPU seconds, and the memory consumption is measured in MB. In the experiments, we compare the heuristic method for diagnostic trace generation to a method based on random trace generation.

In these experiments the methods are limited to a maximum of five iterations between the debugging and the verification procedures. The number of generated traces is limited to 2000 traces.

Table I presents the experimental results with respect to the diagnosis accuracy. The first and second columns show the module name of OpenRISC and the total number of components ($\#C$) which are used for SAT-based debugging as mentioned in Section II-B. The third and fourth columns present the debugging result in the first session when debugging tries to find the potential number of fault candidates ($\#FC$) with the initial counterexamples ($\#CE$). The diagnosis accuracy is considered to be the inverse of $\#FC$. The columns 5-7 show the result when the heuristic method is used for generating the diagnostic traces, while the columns 8-10 are related to the random trace generation. The best results are marked bold in Table I.

For $or1200\_alu$, the heuristic method obtains four new counterexamples after generating 41 diagnostic traces. Thus by a total number of five counterexamples (one initial counterexample and four new counterexamples) the diagnosis ac-

TABLE II
TIME AND MEMORY

Method		Heuristic Method				Random Method			
Parameter		Time			Memory	Time			Memory
Circuit	#C	Deb. (s)	Ver. (s)	Total (s)	(MB)	Deb. (s)	Ver. (s)	Total (s)	(MB)
or1200_alu	436	0.44	11.41	**11.85**	13	0.47	16.98	17.45	13
or1200_ctrl	1865	22.81	106.81	**129.62**	166	20.55	234.45	255	166
or1200_genpc	732	17.33	214.3	231.63	144	15.9	123.38	**139.28**	144
or1200_if	463	4.25	7.76	**12.01**	72	4	81.71	85.71	72
or1200_lsu	793	7.86	27.42	**35.28**	145	8.21	144.9	153.11	145
or1200_operandmuxes	376	49.25	80.01	129.26	41	45.65	74.86	**120.51**	41
or1200_wbmux	278	42.83	7.16	**49.99**	54	42.4	62.78	105.18	54

curacy increases, i.e., the total number of fault candidates decreases. Also the random method has the same accuracy for $or1200\_alu$. For $or1200\_ctrl$, $or1200\_genpc$, $or1200\_if$, $or1200\_operandmuxes$, and $or1200\_wbmux$, the random method cannot obtain any counterexample, while the heuristic method obtains the new counterexamples with a small number of diagnostic traces. For $or1200\_lsu$, the random method generates some counterexamples which do not have potential to reduce the number of fault candidates, while the counterexamples obtained from diagnostic traces reduce the number of fault candidates. Totally, in four experiments out of seven experiments, the heuristic method achieves a better diagnosis accuracy than the random method.

Table II shows the required run time ($Time$), and the maximum memory consumption ($Memory$) for each method. The time is related to debugging time and verification time. The best total times are marked bold. The heuristic method spends less verification time than the random method in most of the experiments. Because usually after generating a small number of diagnostic traces, some new counterexamples are identified.

Overall, the experimental results show that debugging automation by using diagnostic traces increases the diagnosis accuracy and decreases the debug time.

VII. CONCLUSION

This paper presented an approach for automating debugging which can be used in different debugging scenarios from pre-silicon to post-silicon. The approach integrates model-based diagnosis and diagnostic trace generation. The experimental results on post-silicon debugging showed that automated debugging by using diagnostic traces increases diagnosis accuracy and decreases debug time.

REFERENCES

[1] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation," *IEEE Trans. on CAD*, vol. 18, no. 12, pp. 1803–1816, 1999.

[2] P.-Y. Chung and I. N. Hajj, "Diagnosis and correction of multiple logic design errors in digital circuits," *IEEE Trans. on VLSI Systems*, vol. 5, no. 2, pp. 233–237, 1997.

[3] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[4] A. Sülflow and R. Drechsler, "Automatic fault localization for programmable logic controllers," in *Formal Methods for Automation and Safety in Railway and Automotive Systems*, 2010, pp. 247–256.

[5] A. Hopkins and K. McDonald-Maier, "Debug support for complex systems-on-chip: a review," *Proc. of Computers and Digital Techniques*, vol. 153, no. 4, pp. 197–207, 2006.

[6] B. Vermeulen, T. Waayers, and S. Bakker, "IEEE 1149.1-compliant access architecture for multiple core debug on digital system chips," in *Int'l Test Conf.*, 2002, pp. 55–63.

[7] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Design Automation Conf.*, 2006, pp. 7–12.

[8] J.-S. Yang and N. A. Touba, "Expanding trace buffer observation window for in-system silicon debug through selective capture," in *VLSI Test Symp.*, 2008, pp. 345–351.

[9] Y. Lee, T. Matsumoto, and M. Fujita, "On-chip dynamic signal sequence slicing for efficient post-silicon debugging," in *ASP Design Automation Conf.*, 2011, pp. 719–724.

[10] F. M. de Paula, A. Nahir, Z. Nevo, A. Orni, and A. J. Hu, "TAB-BackSpace: Unlimited-length trace buffers with zero additional on-chip overhead," in *Design Automation Conf.*, 2011, pp. 411–416.

[11] Y. L. Ted Hong, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "QED: Quick error detection tests for effective post-silicon validation," in *Int'l Test Conf.*, 2010, pp. 1–10.

[12] S.-B. Park, T. Hong, and S. Mitra, "Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA)," *IEEE Trans. on CAD*, vol. 28, no. 10, pp. 1545–1558, 2009.

[13] K. Chang, I. Markov, and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," in *ASP Design Automation Conf.*, 2007, pp. 944–949.

[14] K.-H. Chang, I. L. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," in *Int'l Conf. on CAD*, 2007, pp. 91–98.

[15] M. Dehbashi, A. Sülflow, and G. Fey, "Automated design debugging in a testbench-based verification environment," in *EUROMICRO Symp. on Digital System Design*, 2011, pp. 479–486.

[16] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, pp. 57–95, 1987.

[17] B. Vermeulen, T. Waayers, and S. Goel, "Core-based scan architecture for silicon debug," in *Int'l Test Conf.*, 2002, pp. 638–647.

[18] Y.-S. Yang, N. Nicolici, and A. G. Veneris, "Automated data analysis solutions to silicon debug," in *Design, Automation and Test in Europe*, 2009, pp. 982–987.

[19] K. Constantinides, O. Mutlu, and T. M. Austin, "Online design bug detection: RTL analysis, flexible mechanisms, and evaluation," in *International Symposium on Microarchitecture (MICRO)*, 2008, pp. 282–293.

[20] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Design, Automation and Test in Europe*, 2008, pp. 1298–1303.

[21] G. Friedrich, M. Stumptner, and F. Wotawa, "Model-based diagnosis of hardware designs," *Artificial Intelligence*, vol. 111, no. 1–2, pp. 3–39, 1999.

[22] W. Mayer and M. Stumptner, "Model-based debugging - state of the art and future challenges," ser. Electronic Notes in Theoretical Computer Science, vol. 174, no. 4, 2007, pp. 61–82.

[23] J. de Kleer and J. Kurien, "Fundamentals of model-based diagnosis," in *IFAC Symposium on Fault Detection, Supervision, and Safety of Technical Processes (Safeprocess)*, 2003, pp. 25–36.

[24] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.

[25] M. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.

[26] J. A. Knottnerus and J. W. Muris, "Assessment of the accuracy of diagnostic tests: the cross-sectional study," *Journal of Clinical Epidemiology*, vol. 56, no. 11, pp. 1118–1128, 2003.

[27] F. Zheng, K.-T. Cheng, X. Yan, J. Moondanos, and Z. Hanna, "An efficient diagnostic test pattern generation framework using boolean satisfiability," in *ASP Design Automation Conf.*, 2007, pp. 288–294.

[28] T. Grüning, U. Mahlstedt, and H. Koopmeiners, "DIATEST: A fast diagnostic test pattern generator for combinational circuits," in *Int'l Conf. on CAD*, 1991, pp. 194–197.

[29] A. Sülflow, G. Fey, and R. Drechsler, "Using QBF to increase accuracy of SAT-based debugging," in *IEEE International Symposium on Circuits and Systems*, 2010, pp. 641–644.

[30] M. Psarakis, D. Gizopoulos, E. Sánchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[31] K. Constantinides, O. Mutlu, T. M. Austin, and V. Bertacco, "A flexible software-based framework for online detection of hardware defects," *IEEE Trans. Computers*, vol. 58, no. 8, pp. 1063–1079, 2009.

[32] *OpenCores*, http://www.opencores.org.

[33] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT 2003*, ser. LNCS, vol. 2919, 2004, pp. 502–518.