From RTL to Silicon: The Case for Automated Debug

Andreas Veneris[1,2], Brian Keng[1], Sean Safarpour[3]

***Abstract*—Computer-aided design tools are continuously improving their scalability and efficiency to mitigate the high cost associated with designing and fabricating modern VLSI systems. A key step in the design process is the root-cause analysis of detected errors. Debugging may take months to close, introduce high cost and uncertainty ultimately jeopardizing the chip release date. This study makes the case for debug automation in each part of the design flow (RTL to silicon) to bridge the gap. Contemporary research, challenges and future directions motivate for the urgent need in automation to relieve the pain from this highly manual task.**

I. Introduction

The enormous demand for larger and more complex VLSI systems dramatically increases the cost of their verification, test and debug. Verification aims to determine if there is an error in the implementation of a design that causes a mismatch with its specifications. If there is an error, debugging follows and aims to determine the root-cause of the detected errors found during verification. It has been reported, these two tasks combined can contribute up to 70% of the total chip design time costing millions of dollars in non-recurring engineering costs [1].

Although bugs can manifest themselves in every part of the design stage, debugging functional errors has recently become one of the largest pain points consuming more than half of the total verification time [2]. To compound this problem, errors frequently escape pre-silicon verification and appear in silicon prototypes during test, exasperating the difficulty of the debugging task. It then comes as no surprise that 60% of ASICs fail not because of timing or power issues, but because of functional errors [3]. And this is not expected to get better; it is projected, the burden of verification/debug is expected to grow 675% by 2015 [4] from current levels.

For one reason, this dramatic increase in the complexity of functional debugging relates to the complexity of modern designs and their accompanying verification environments. Modern designs are typically composed of an interwoven ecosystem of in-house, legacy and acquired IP design blocks. Each block is stitched together to create a complex system whose behavior is increasingly becoming more opaque and thus more difficult to debug. Meanwhile, the modern verification environment has also become a diverse collection of different technologies built to efficiently verify these complex systems. Testbenches, assertions, and verification IP are all standard building blocks in the contemporary flow. The interaction of these components with the design further complicate the debugging process by adding another layer of understanding. Complex interactions demand experts who are familiar with all these components to perform debugging efficiently.

However, even with expert engineers, debugging remains an incredible feat for any one person to undertake. Once a failure is detected, the actual error may no longer just be confined to the design. Errors are now just as likely to occur in one of the heterogeneous verification components such as the testbench or the assertions. Moreover, the design and the verification environment are typically written by different teams of engineers with different sets of expertise. This only compounds the debugging pain by adding additional communication and complexity overhead, demonstrating a desperate need for automation to manage this complexity. In addition, when the bug is discovered in the silicon, entirely new challenges present themselves in this daunting debugging task.

Current state of the art industrial tools primarily aid the engineer in manually navigating this complexity. Tools such as graphical waveform viewers and "what-if" analysis engines are commonplace in an industrial environment. Although they provide good value, their manual nature limits their ability to efficiently scale to larger and more complex systems. Without significant advances in automated industrial scale solutions, debugging will continue to severely add overhead and cost as the size and complexity of the systems increase.

In this paper, we make the case for automated debugging tools in different steps of the design cycle: RTL verification, design emulation and silicon test. We review recent research advances in automated RTL- and silicon-debug methodologies, we discuss the challenges behind these problems and point to future research directions. Throughout this presentation, we emphasize the need for scalable debugging solutions as a means to reduce costs and to handle the inherent complexity behind the modern chip development cycle.

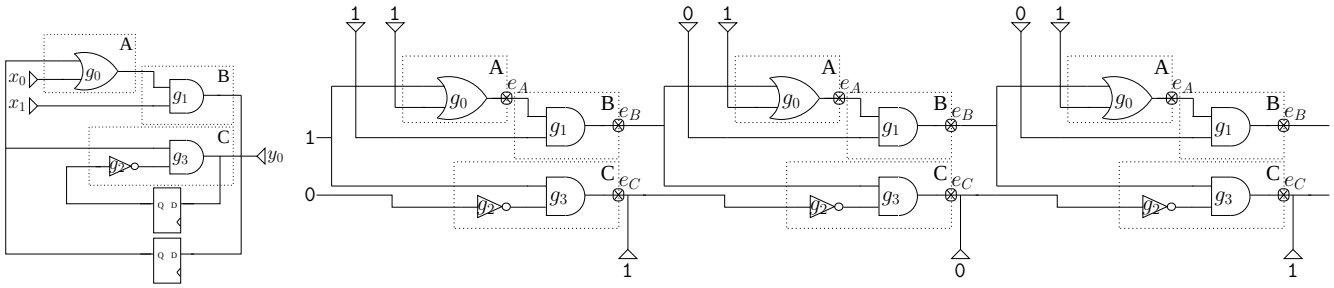
II. Debugging the RTL

Functional debugging begins when a failure occurs during RTL verification. Whether this failure is caught by a simulation testbench or by a formal property checker, the debugging process typically requires three pieces of information to determine its source. The first piece is the buggy design. Next, an error trace (*i.e.*, a counter-example) that contains an initial state with

[1]University of Toronto, ECE Department, Toronto, ON M5S 3G4 ({briank, veneris}@eecg.toronto.edu)

[2]University of Toronto, CS Department, Toronto, ON M5S 3G4

[3]Vennsa Technologies, Inc., Toronto, ON M5V 3B1 (sean@vennsa.com)



(a) Original Circuit

(b) SAT-based Debugging

Fig. 1. Example 1: SAT-based Debugging

a primary input stimulus is required. Finally, a set of correct or expected values at an observation point(s) such as a primary output(s) must be provided. Using this information, debugging identifies the root-cause of the error, also known as the *suspect* location in the RTL, so that a correction can be applied to fix it. There are cases when debugging has to be performed in the absence of an error trace, for example, when the design never reaches a given expected state [5]. These types of errors present additional challenges and are not covered here.

Whether using a manual or an automatic approach, the debugging problem becomes more complex as the design, error trace length and the number of errors grow [6]:

$$solution\ space = (design\ size * trace\ length)^{\#errors} \quad (1)$$

The growing complexity of modern system-on-chip designs increase all three factors in the above equation. Evidently, this severely adds burden to the manual analysis and places an increased emphasis for scalable automated solutions to manage these complexity components. This section provides an overview of recent advances in automated RTL debugging that tackle these aspects of problem complexity.

A. Simulation and BDD-based Techniques

Debugging techniques relying on simulation [7]–[10] and BDDs [7], [11] were among the earliest. These solutions perform well under certain conditions but the size of modern designs poses a challenge in their ability to scale.

The simulation-based methods localize errors by first simulating the design and then tracing back from the erroneous output based on different criteria. Although this process is memory efficient, its run-time and resolution degrades with sequential designs and multiple errors limiting its applicability. BDD-based methods [11] demonstrate an algebraic solution to debugging single and multiple errors. Although effective for single errors, BDDs are limited by memory issues as the design scales.

B. Satisfiability-based Techniques

Recent advances in Boolean satisfiability (SAT) provide significant benefit in automated debugging techniques. In [12], debugging is first encoded as a SAT instance. Other formulations that followed, built upon this initial framework using various related satisfiability technologies [13]–[16]. These techniques have shown to outperform previously proposed simulation and BDD-based techniques by orders of magnitude in certain cases. Moreover, they have shown to be robust and maintain their good resolution when scaling to larger designs and to multiple errors.

The basic formulation proposed in [12] creates a SAT instance where each satisfying assignment corresponds to a potential set of suspects. The formulation is constructed in several steps. First, the design is enhanced with error models for individual gates or groups of gates corresponding to lines of RTL. Each error model has a corresponding suspect variable, e_i, such that if the $e_i = 1$, then its fan-out is disconnected from its fan-in and it becomes free. This can be achieved through a hardware construction using multiplexors, or directly in conjunctive normal form (CNF).

Next, the combinational part of the design is translated to CNF and unrolled using *time-frame expansion* for the length of the error trace. This models the sequential behavior of the design. The initial state, primary inputs and primary outputs are then constrained with the values from the error trace and expected values respectively. The number of errors (N) to search is set to a fixed value by adding a constraint on the suspect variables. When a satisfying assignment is returned, exactly N suspect variables will be activated corresponding to a set of locations that could potentially fix the observed failure. The following example illustrates this technique.

Example 1 *A simple sequential circuit with four gates and two state elements is shown in Figure 1(a). This circuit has primary inputs x_0, x_1 and primary output y_0. It also contains three groupings of gates (A, B, C) that represent locations in the RTL. A SAT-based debugging formulation for Figure 1(a) is shown in Figure 1(b) unrolled for three time-frames. The circuit is enhanced with error models represented by $\otimes$. Notice error models are only added to the portions corresponding to RTL locations. A failure occurs in the third time-frame where the primary output should be* 1 *instead of the implemented* 0*. When setting $N = 1$, we find the only satisfying assignments are $e_B = 1$ or $e_C = 1$. This corresponds to suspects for groupings B and C.*

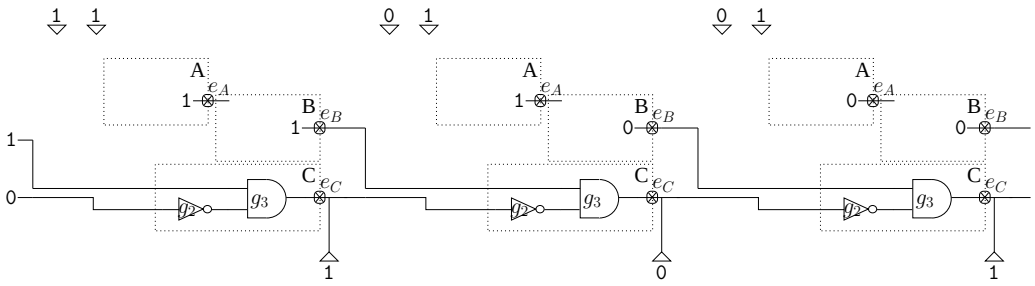


Fig. 2. Example 2: Abstraction and Refinement

C. Managing Complexity

By leveraging the tremendous amount of research in formal verification, recent advances [6], [17], [18] have been able to further contain much of the parameters behind the inherent complexity (Equation 1) in design debugging. The most promising techniques manage the design size and the error trace length. The following subsection describe two approaches that aim to tackle these complexity metrics.

One technique is based upon *abstraction and refinement*. Research in [17] adapts this concept in an effort to reduce the complexity by reducing the design size. It works by removing components in the design to create a simpler abstract one. The abstracted design is then solved for all suspects. Based on suspects returned, an iterative refinement step occurs in which some of the removed components are reintroduced back into the design. This process continues until either the bug is found or a stopping condition is reached.

The iterative and incremental nature of this algorithm ensures that only relevant portions of the circuit need be modeled reducing a major portion of the debugging complexity. It also provides a generic solution that is not tied to any explicit debugging formulation. Experimental results show orders of magnitude reductions in run-time and memory demonstrating the efficacy of this proposed technique. The next example illustrates the main idea.

Example 2 *The abstraction and refinement algorithm is used on the same debugging problem from Example 1 as shown in Figure 2. Here we see the debugging problem after one refinement iteration where grouping C is found as a suspect and has been reintroduced while groupings A and B have been abstracted. For each component that has been abstracted, its outputs have been constrained to their simulated values. Notice that the unused transitive fan-in of these components can also be removed, greatly simplifying the debugging problem. For $N = 1$, both B and C are returned as suspects, the same result as Example 1, without the need to model every component in the circuit.*

Another technique tackles complexity by reducing the number of cycles in the error trace to be analyzed. *Bounded Model Debugging* [6] works by analyzing subsections, or windows, of the error trace. In this debugging-independent formulation, the algorithm iteratively models a growing suffix window of the error trace. This approach uses the fact that errors are likely to be closer to their failure point to focus the analysis on the most likely portions of the trace. In the worst case, the basic formulation requires that the whole trace to be analyzed. A later extension, introduces SAT-based interpolants to partition the error trace into non-overlapping windows. Each window of the error trace is analyzed separately to reduce memory and run-time at the cost of a minor loss in suspect resolution.

Long error traces with thousand of clock cycles are a reality in practical industrial problems. The ability of Bounded Model Debugging to iteratively analyze long traces greatly enhances the practical applicability of automated debugging. Experimental results show the benefit of this approach for industrial designs and real-life error traces as it is able solve these instances where previous methods either memory-out or time-out.

III. Debugging Emulation and Test

Although pre-silicon verification aims to detect all functional errors, the size and complexity of modern designs rarely results in a bug-free silicon prototype. This problem is becoming more prevalent as time-to-market constraints reduce pre-silicon verification time and modern system-on-chip designs increase in size and complexity. As such, functional bugs may escape pre-silicon verification only to be found during emulation or during in-system silicon test. Similarly, design starts that target Field Programmable Gate Arrays (FPGAs) are also experiencing errors during validation [19].

The difference in the challenges for emulation and silicon debug when compared to pre-silicon verification are manifold. First, observability of signals during emulation and test, is severely limited as compared to RTL debug. In RTL debug, any signal at any time may be observed. In contrast, when the design is prototyped in silicon, only a limited amount of signals can be observed within a limited clock cycle window. This greatly increases the time and cost of debugging. Next, the size of traces that occur can be orders of magnitude larger than traces found during RTL verification. This limits the amount of manual analysis possible and restricts the types of automated computational techniques that can be used. Finally, bugs detected during silicon test can be either deterministic or non-deterministic. A deterministic bug is reproducible with the same test-vector. Non-deterministic bugs may not reproducible with the same test-vector due to events such as interrupts or refresh of dynamic memories [20].

A. Design for Silicon Debug Techniques

The two main Design-for-Debug (DfD) techniques to increase observability during test are *scan chains* and *trace buffers*. Scan chains are inserted during Design-for-Test and they are later reused in silicon debug [21]. They work by connecting all scanned registers in a large shift register. During test-mode, the values in the registers may be shifted out to be observed. However, unless two state elements are used for each register, which dramatically increases the area, the environment must be reset after each scan dump. This leads to greatly improved observability of the chip but the need to reset the test environment after a scan dump results in a significant loss of debug efficiency.

Trace buffers [22], [23] increase observability by recording internal signals using an on-chip memory. They contain control logic (e.g., embedded assertions [24]) that trigger the online monitoring of circuit signals. Once the trigger condition is activated, the logic values of selected signals are recorded onto the on-chip memory. The data can then be output via a low-bandwidth interface such as boundary scan. Typical sizes of trace buffers range from 8k to 256k limiting their ability to scan all signals for an extended period of time.

B. Techniques for Debugging Silicon Prototypes

Techniques to improve silicon debug efficiency have primarily aimed at improving the silicon debug work flow. A typical debug session involves running a test vector on the chip, observing selected signals for a limited window of time, and analyzing the data to determine the root-cause of the failure. Due to limited observability, even with DfD techniques, multiple iterations of this flow are typically necessary.

The *BackSpace* framework presented in [25] aims to provide the ability to trace back thousands of cycles from the crash state to determine the original error excitation state. It works by augmenting the design with hardware to record small signatures of key states. When a crash occurs, it uses formal techniques combined with the signatures to trace back to a set of possible states from which it could have come from. Although limited in the amount of cycles that it can trace back, BackSpace additionally adds programmable break point circuitry. This allows calculated predecessor states to be set as break points allowing for significantly increased efficiency in computing back traces.

Another technique [26] provides a solution to automating the analysis portion of the flow. It uses the captured scan dump combined with trace buffered data to determine the erroneous module as well as its excitation time. Additionally, it provides suggestions as to which signals and times to capture during the next debug session. Through judicious use of SAT-based techniques it is able to improve the silicon debug work flow by reducing the iterations and manual root-cause analysis required.

Although these solutions present promising results in providing value for silicon debug, admittedly much more research is required to make them practical. The ability to scale to practical sized industrial designs is the primary challenge for silicon debug automation. This only demonstrates the need for automation as these debugging problems grow beyond the capabilities of human analysis alone.

C. Debugging In-System FPGA Designs

As current state-of-the-art FPGAs boast hundreds of thousands of look-up tables (LUTs), thousands of hard-IP blocks, and multi-mega bits of memory, they are experiencing similar growing pains to ASICs. In particular, they are experiencing the same verification gap as ASICs in that the ability to design larger FPGAs outpaces the ability to verify them [19].

The problem of debug becomes troublesome when a bug is found during in-system validation of the FPGA. This could be caused either by extremely long test vectors that are not practically simulatable during RTL verification, or because the chip does not match its specification. The debug work flow is similar to silicon prototypes as one needs re-run the test environment, capture select signals and analyze the resulting data. A significant difference is the ability to reconfigure the chip, adding user defined observation points to many more signals. Despite this reconfigurability, debug remains a pain as reprogramming the chip requires an new synthesis run, dramatically limiting efficiency. This points to a need for solutions to debugging FPGAs. Without this innovation, the full benefit of the increased capacity and capabilities of FPGAs will be severely limited very soon.

IV. Debugging Verification Environments

Traditionally, automated debugging research has focused solely on automating debug within a design due to the high cost of root-cause analysis for design errors. However as verification environments grow more complex, they are also highly likely to contain costly errors.

Errors in the verification environment represent a significant departure in objectives from debugging traditional RTL. First, they are complicated entities comprised of various assertions, testbenches and verification IP. Each component can be at a different level of abstraction (*e.g.* procedural, behavioural, synthesizable) using a different language (*e.g.* SystemC, SystemVerilog, Matlab, Verilog, VHDL) all stitched together in a complex heterogeneous entity. This adds immense complexity to any type of root-cause analysis, manual or automated. Second, many of the components are non-synthesizable. This eliminates the applicability of most traditional design debugging techniques that require a synthesizable model. Finally, as the likelihood of verification environment errors becomes larger, determining if the error is in the design or in the environment becomes an issue. This leads to not only a more costly debugging process but also an increased uncertainty in the quality and robustness of the verified design. These reasons point to a need for innovative solutions to tackle these new challenges.

A. Automating Debugging of Temporal Assertions

The work in [27] presents an automated methodology for debugging errors occurring within SystemVerilog assertions. It takes a different approach compared to traditional design

debugging techniques. Instead of viewing the process as a localization task, it views it as a correction problem. The rationale is that localization is effective for design debugging because there are many simple components in the design. Once an error is localized, correction is typically straight forward. However for temporal assertions, the reverse is true. Most assertions are relatively small combinations of expressions so localization is typically not the most difficult aspect of debugging. Rather, finding the appropriate correction makes the bulk of the work.

To this end, [27] presents a methodology to debug assertions by correcting them. Using the failing assertion and counter-example, a *mutation model* produces a set of closely related properties to the failing assertion that have been verified against the design. These closely related properties aid debugging by acting as a basis for possible corrections to the failing assertion. Mutations also provide significant insight into the design behavior if the generated property does not correct the error. Experiments confirm that the technique always finds closely related alternative properties.

B. Debugging Testbench Environments

The rapid adoption of new testbench technologies has led to a dramatic increase in verification efficiency but has lead to increased complexity in the environments as well. Because new technologies are utilized over a short period of time, this results in a patchwork of new and legacy testbench components. Due to this evolution, there is no standard methodology or standardized interactions between the numerous different types of components and technologies. For example, a typical environment might involve a SystemVerilog testbench, Matlab simulations, and shell scripts to act as intermediaries between the two. If the bug is assumed to be in any of these components, then any analysis tool would require knowledge of each component. This represents significant challenges to automation.

Beyond compatibility and support for the various different types of technologies, automated analysis techniques for the high level testbench languages are limited. Many popular languages, such as Matlab and SystemVerilog, are not practically synthesizable due to the inclusion of programming language constructs such as wide data types and dynamic memory allocation. This severely limits the types of analysis, resulting in a narrow category of solutions that can deal with these problems. With the increased likelihood of errors occurring in testbenches, this problem remains a significant roadblock to reap the full benefits of recent advances in functional verification.

V. Conclusion

The manual and resource-intensive nature of the debugging process is a major pain in the industry today that manifests itself in every design and test aspect. This study outlines the challenges, recent advances and future directions in debug automation from RTL to silicon. After 20 years of intense research, this paper makes a case that debug automation today is feasible, and necessary, to relieve the high cost associated with the manual effort.

References

[1] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publisher, 2000.

[2] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, 2008, pp. 5–10.

[3] J. Jaeger, "Virtually every ASIC ends up an FPGA," *EE Times*, December 2007.

[4] D. McGrath, "De Geus touts new products, says ICs will rebound," *EE Times*, March 2009.

[5] R. K. Ranjan, C. Coelho, and S. Skalberg, "Beyond verification: leveraging formal for debugging," in *Design Automation Conf.*, 2009, pp. 648–651.

[6] B. Keng, S. Safarpour, and A. Veneris, "Bounded Model Debugging," *IEEE Trans. on CAD*, vol. 29, no. 11, pp. 1790 –1803, 2010.

[7] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[8] M. Tomita, T. Yamamoto, F. Sumikawa, and K. Hirano, "Rectification of multiple logic design errors in multiple output circuits," in *Design Automation Conf.*, 1994, pp. 212–217.

[9] V. Boppana and M. Fujita, "Modeling the unknown! toward model-independent fault and error diagnosis," in *Int'l Test Conf.*, 1998, pp. 1094–1101.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] H. Mangassarian, A. Veneris, and M. Benedetti, "Robust QBF encodings for sequential circuits with applications to verification, debug and test," *IEEE Trans. on Comp.*, vol. 99, 2010.

[16] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated Design Debugging With Maximum Satisfiability," *IEEE Trans. on CAD*, vol. 29, no. 11, pp. 1804 –1817, 2010.

[17] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Trans. on CAD*, vol. 28, no. 10, pp. 1597–1608, 2009.

[18] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Great Lakes Symp. VLSI*, 2008.

[19] D. Orecchio, "FPGAs advance, but verification challenges increase," *EE Times*, October 2010.

[20] S. Sarangi, B. Greskamp, and J. Torrellas, "CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging," in *International Conference on Dependable Systems and Networks*, 2006, pp. 301 –312.

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

[22] A. Abramovici and Y.C.Hsu, "A new approach to silicon debug," in *IEEE International Silicon Debug and Diagnosis Workshop*, Nov. 2005.

[23] E. Anis and N. Nicolici, "Low Cost Debug Architecture using Lossy Compression for Silicon Debug," in *Design, Automation and Test in Europe*, 2007, pp. 1 –6.

[24] M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer Publishing Company, Incorporated, 2008.

[25] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, "Backspace: Formal analysis for post-silicon debug," in *Formal Methods in CAD*, 2008, pp. 1–10.

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

[27] B. Keng, S. Safarpour, and A. Veneris, "Automated Debugging of SystemVerilog Assertions (to be presented)," in *Design, Automation and Test in Europe*, 2011.