Enhancing Bug Hunting Using High-Level Symbolic Simulation

Hong-Zu Chou†, I-Hui Lin†, Ching-Sung Yang†, Kai-Hui Chang‡, and Sy-Yen Kuo†
†Electrical Engineering Department, National Taiwan University, Taipei, Taiwan.
‡Avery Design Systems, Inc., Andover, MA, USA.
sykuo@cc.ee.ntu.edu.tw

ABSTRACT

The miniaturization of transistors in recent technology nodes requires tremendous back-end tuning and optimizations, making bug fixing at later design stages more expensive. Therefore, it is imperative to find design bugs as early as possible. The first defense against bugs is block-level testing performed by designers, and constrained-random simulation is the prevalent method. However, this method may miss corner-case scenarios. In this paper we propose an innovative methodology that reuses existing constrained-random testbenches for formal bug hunting. To support the methodology, we present several techniques to enhance RTL symbolic simulation, and integrate state-of-the-art word-level and Boolean-level verification techniques into a common framework called BugHunter. From case studies DLX, Alpha and FIR, BugHunter found more bugs than constrained-random simulation using fewer cycles, including four new bugs in the verified design previously unknown to the designer. The results demonstrate that the proposed techniques provide a flexible, scalable and robust solution for bug hunting.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*

General Terms

Verification

Keywords

BugHunter, symbolic simulation, design for verification

1. INTRODUCTION

Verification techniques for hardware design have been extensively investigated over the past decades [1,9,12]. Among the verification methods that exist today, logic simulation is the most commonly used. It verifies the functionality of the design under test through user-provided or constrained-random patterns [15]. Although simulation-based methods are scalable and inexpensive to implement, it is difficult to measure how well a design has been tested unless all possible input patterns can be exhaustively applied. To overcome this limitation, formal verification techniques have been developed [9]. Formal verification tools exploit mathematical methods to check whether the design meets the requirements of the specification for all possible inputs. However, formal verification is not scalable enough to handle the increasing complexity of modern designs. In addition, formal methods rely on assertions to express design properties using temporal logic, which could not be easily produced by designers. In practice, designers often write direct or constrained-random testbenches to test their blocks, and rely on verification engineers for rigorous testing. This approach leaves many bugs that could have been found in early design phases to a much later stage, prolonging the verification cycle.

Recently, symbolic simulation has attracted attention due to its efficiency over traditional simulation-based methods [12]. Unlike logic simulation which only simulates the given scalar inputs at a time, symbolic simulation treats primary inputs and outputs in Boolean expressions as symbols which can evaluate all possible values simultaneously to provide formal proof capabilities. Traditionally, Binary Decision Diagrams (BDDs) [2] have been widely used to represent the Boolean expressions due to their flexibility in Boolean manipulations [4,5]. Seger and Bryant invented symbolic trajectory evaluation (STE) [14] which is much less sensitive to state explosion due to precise constraining and checking of system state during sequences of operation. Kolbl et al. [10,11] introduced symbolic RTL simulation; this approach is the first to enable RTL Verilog construct with delay support. Rieschl [13] proposed a symbolic interface for debugging register-transfer simulator.

In this paper, we propose a formal verification paradigm that designers can use in their block-level testing. Our first contribution is a new verification methodology that reuses existing constrained-random testbenches for formal verification via high-level[1] symbolic simulation. This methodology can help designers thoroughly verify their designs without much additional effort. Since a powerful symbolic simulator is critical to the effectiveness of our methodology, we propose an enhancement that utilizes on-line simulation bounding to improve the scalability and efficiency of symbolic simulation. In addition, we integrate state-of-the-art Boolean-level verification techniques into our unified framework, called BugHunter, for RTL verification. Due to the improved power and flexibility of BugHunter over traditional formal tools, we also propose a heterogeneous design verification methodology that provides a novel way to verify the design under test when golden models are not available. Furthermore, we point out how "inconclusive results" can be derived and used to improve the verifiability of the design [7]. We believe the concepts and techniques presented in this paper can improve verification at early design stages, thus reducing bugs in the final products and improving overall electronic design quality.

The rest of this paper is organized as follows. In Section 2, techniques to enhance high-level symbolic simulation are presented. We propose new verification methodologies to leverage our enhanced symbolic techniques in Section 3. Empirical results are shown in Section 4, and Section 5 concludes this paper.

2. ENHANCING HIGH-LEVEL SYMBOLIC VERIFICATION

The efficiency of high-level symbolic simulation greatly affects the quality of BugHunter. In this section we propose several innovations to improve existing symbolic simulation methods.

2.1 Integrating High-Level Symbolic Simulation and Boolean-Level Verification

To leverage the benefits of both high-level symbolic simulation and recent improvements in Boolean-level verification methods, we propose a novel framework for RTL symbolic verification as shown in Fig. 1. This framework can be divided into four phases:

[1]Behavior or Register Transfer Level (RTL)

GLSVLSI'09, May 10–12, 2009, Boston, Massachusetts, USA.

high-level symbolic simulation, word-level optimization, Boolean-level conversion and Boolean-level verification. In the first phase, we perform high-level symbolic simulation to generate word-level logic expressions. Next, we perform word-level optimizations (described later) to simplify these expressions. In the third phase, we decompose word-level expressions to Boolean level and generate a netlist. Finally, we feed the netlist to Boolean-level verification tools, such as VIS [16] or ABC [17], to solve the problem instance.

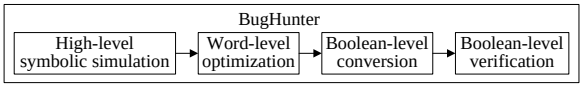

Figure 1: Integrated framework for bug hunting using symbolic simulation and Boolean-level verification (BugHunter)

By decoupling Boolean-level verification from word-level simulation and optimization, we can achieve even better performance by running multiple Boolean-level engines simultaneously. Since different engines may be suitable for different designs and verification targets, we utilize the following Boolean-level verification methods to support various verification goals. 1) BDD/SAT based method [16]: this method is suitable for small designs whose BDDs can be built easily. 2) AIG/SAT based method [17]: this method first converts the design into AIG and then uses SAT to solve the verification problem. 3) AIG/SAT/synthesis-for-verification approach [17]: this method performs several synthesis-for-verification optimizations, such as node merging and local AIG rewriting, before calling SAT. This approach is suitable for verification problems with abundant structural similarities, such as sequential equivalence checking between a golden model and its optimized variant.

Our experimental results show that this framework can leverage the power of both word-level and Boolean-level verification techniques. As the FIR example shows (see Section 4), problems that are difficult for Boolean-level verification tools can be solved easily using our framework due to our word-level optimizations. On the other hand, our use of both word-level and Boolean-level engines allows us to handle both control and datapath circuits.

In our implementation we also utilized high-level structural hashing and on-line simplification. Structural hashing can dramatically reduce the size of generated Boolean expressions. In some cases, a problem instance can be proved or disproved using symbolic simulation alone due to structural hashing. In addition to traditional on-line simplification methods that rely on Boolean manipulations such as simplifying "1 OR A" to "1", we also perform algebraic simplifications like converting "A*2" to "A$\ll$1". We found that when used correctly, these techniques can improve simulation efficiency significantly.

2.2 Branch and Bound of Simulation Traces

Symbolic simulation needs to consider all possible execution paths of the design code, which may produce a large number of simulation traces, all under different conditions. For example, in an "if-then-else" statement, symbolic simulation will need to execute both branches under different conditions. Although branch merging techniques exist [10, 11], they can only alleviate this problem but cannot totally solve it.

To address this problem, we use Boolean-level verification tools to check the conditions when branching is necessary and bound the simulation traces that cannot possibly happen. Take an "if-then-else" construct for example, if the condition can be proven to be constant 0 or 1, then only one branch needs to be evaluated, thus reducing the complexity of symbolic simulation. Currently, we use this technique to check two types of branching situations: 1) conditional statements like "if-then-else" or "while" loops; and 2) edge detection constructs like "@(posedge signal)". Empirically, we found that this method can greatly reduce the Boolean expressions generated by symbolic simulation, thus reducing memory use. However, this simplification may sometimes increase runtime due to the overhead to call Boolean-level solvers, showing a trade-off between runtime and memory usage. In our experiments, we only check type 1 for the best performance.

3. NEW RTL SYMBOLIC VERIFICATION PARADIGMS

Our BugHunter framework enables many new RTL symbolic verification paradigms that can improve bug hunting at the block level. In this section we describe these new paradigms in detail.

3.1 Reusing Constrained-Random Testbench for Symbolic Verification

In a constrained-random testbench, random numbers are assigned to certain variables in order to explore more design space. Usually, constraints are also used to make sure the inputs to the design are legal. In our work we reuse the same testbench for symbolic verification by injecting a new symbol to a variable whenever a random value is assigned to a variable in logic simulation. In this way, we can explore all possible states that can be reached by the testbench simultaneously to achieve full proof.

3.2 Heterogeneous Design Verification Using Common Testbench

One challenge in constrained-random simulation is how to determine the correctness of results. Usually, designers use a golden model, typically a high-level behavior model or a previously-known correct RTL model, to generate outputs for comparison. However, this approach cannot be used if no such model exists. Alternatively, self-consistency can be used to verify a pipelined design running in a relaxed mode [8]. Although effective, this method cannot discover bugs that are unrelated to the pipeline, such as a bug in the multiplier. To solve this problem, we propose the use of heterogeneous designs to verify each other under the same testbench. To illustrate the idea, we describe a flow to verify a DLX design using an Alpha processor in Fig. 2. In the flow, we first write a testbench that generates common high-level transactions, and then convert them to corresponding instructions for each processor. These instructions are executed by DLX and Alpha separately, and the results are used to verify each other. In this methodology, we do not compare the architectural states between the two processors because their discrepancy may still be large. Instead, we compare a few architecture-independent results such as values in general registers or program counters.

To better explain how our methodology works, we use the verification of the addition instruction as an example. The instruction converter generates ADD for DLX and ADDQ for Alpha, and then asks both processors to write the results to their first general registers. The comparator then compares the results in both registers. If there is any difference, then there is a bug. The same procedure can be applied to many other instructions as long as there are mappings between the MIPS and Alpha instruction sets. By running more than one instruction symbolically using both processors and comparing their final register-file values or program counters, more comprehensive verification can be achieved. Although we may not be able to verify every aspect of the design under test in this way, this methodology still provides a simple way to detect design bugs. With this approach we found two additional bugs in the "correct" DLX design using the Alpha processor, and these bugs have been confirmed by the designer.

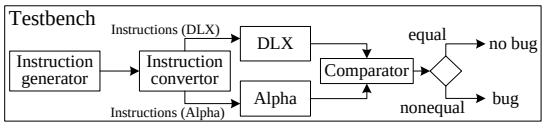

Figure 2: Verifying DLX using Alpha.

Note that this approach will generate difficult problem instances for traditional Boolean-level verification techniques because the heterogeneity of circuits makes structural hashing and signature analysis difficult. However, with word-level simplification and structure hashing, this approach can be applied successfully.

3.3 Improving Verifiability by Identifying Hard-to-Verify Code at Design Stage

Besides the methodologies mentioned above, we can also exploit symbolic simulation to find "inconclusive results" proposed by Ho et al. [7]. Inconclusive results indicate design properties that are intrinsically difficult to verify, such as those buried deeply in a design and require very specific conditions to trigger. By identifying such properties early in the design phase, the design may be modified so that it can be verified more easily. In order to use existing tools for this purpose, they have to write assertions to encode all the conditions, which is a manual and error-prone process.

This design-for-verification tuning method can be supported well by BugHunter, and it works as follows. Most back-end verification tools that we use have a "time-out" mechanism. For example, in ABC we can specify the number of conflicts that are allowed during SAT solving. As described in Section 2.2, we use those verification tools to determine whether or not branching should take place. By reducing the "time-out" threshold, the conditions that timed-out during the bounding process, as well as the properties under those conditions, are those which may be intrinsically more difficult to verify in the future. Since this testing can be performed before the design and testbench are finished, BugHunter can be used to improve the verifiability of the design at early development stages, thus making the design more formal-friendly.

4. EXPERIMENTAL RESULTS

We integrated all techniques mentioned above into a commercial symbolic simulator called Insight [18]. Several RTL/behavior-level testbenches are developed to verify well-known designs as our case studies: a DLX processor, an Alpha processor and FIR. We integrated several Boolean-level verification tools, such as ABC and VIS, into BugHunter. We execute all Boolean-level tools in parallel and use the results from the one that returns first [2]. The experiments are conducted on a Dell PowerEdge 2900 (2 GHz Quad-Core Xeon processor with 9 GByte main memory).

4.1 Case Study: DLX Processor

DLX is a 32-bit RISC microprocessor with 5-stage pipeline designed by Hennessy and Patterson [6]. It is a simplified version of the MIPS architecture and provides a good reference point for verification. The Bug UnderGround project from University of Michigan [19] provides a DLX implementation with 40 bugs, and it is used in this case study[3]. We first developed a testbench to perform sequential equivalence checking (SEC) of DLX between the correct model and the buggy model. To evaluate the performance of the proposed approach, we performed symbolic simulation and constrained-random simulation to compare the execution time, memory consumption and the number of cycles needed to catch the bugs. The circuit is initialized to a random state.

We found that several bugs can never be triggered (i.e., 20, 22, 29, 31, 33 and 34). For testing purpose only, we changed the buggy design to trigger bugs 20, 22, and 31 based on the designer's description. As shown in Table 1, our symbolic method can catch 37 bugs while constrained-random simulation can catch 36 bugs. Although constrained-random simulation can catch several bugs using small numbers of cycles, large numbers of cycles are needed for corner-case or data-dependent bugs. For example, bug 27 can be triggered when an ADD (addition) instruction is followed by a SW (store word) instruction, and the target register of the SW instruction is the same as the source register of the instruction following SW. To trigger this condition, constrained-random simulation needed more than 5 millions cycles. On the other hand, our symbolic method can hit all the bugs using a fixed number of cycles due to its formal nature. The maximum execution time of BugHunter is 613 seconds and the memory usage is less than 287 Mbytes. Moreover, for bugs 29, 33 and 34, BugHunter proved the equivalency between the correct model and the buggy models in less than 700 seconds up to 12 cycles, and this cannot be done by constrained-random simulation. The results demonstrate that our symbolic method can dramatically improve the verification quality compared with constrained-random simulation using the same testbench. A comparison with intelligent test generators such as StressTest [15] shows that the performance of BugHunter is still better: the number of cycles generated by StressTest is much larger; moreover, they need additional monitor for status checking and parameter turning, while BugHunter only requires existing constrained-random testbenches and is easier to use.

When a golden model does not exist, BugHunter can be used for self-consistency checking. To evaluate this feature, we made a simple experiment that feeds pipelined and non-pipelined instructions to verify DLX Bug22. The result shows that even though the structural similarity between two designs is weak, our BugHunter can still catch the bug in 107.997 seconds using 758 M memory.

In summary, BugHunter provides the following benefits compared with constrained-random simulations. First, it can verify designs using much smaller numbers of cycles. This feature not only exhibits good performance gain, but also reduces the complexity of debugging. In addition, our method can catch all bugs and achieve a much higher coverage than constrained-random simulation. The most important feature of our symbolic method is that existing constrained-random testbenches can be reused for formal verification. Therefore, it can be easily integrated into currently prevalent simulation-based verification flows.

4.2 Case Study: Alpha Processor

In this section, we verified an Alpha design using BugHunter. Alpha is a processor which includes 64-bit registers, instructions and datapaths. The BUG project from Michigan also includes an Alpha implementation with 10 bugs, and it is used in this case study. Similar to the previous case study, we prepared a testbench to check the equivalence of the correct model and the buggy model. The results are presented in Table 2. We observe that both symbolic and constrained-random simulation can hit all 10 bugs. However, symbolic simulation uses significantly smaller numbers of cycles in most cases, making debugging much easier. On the other hand, Alpha processor is more complex than DLX, but the execution time and memory consumption does not increase significantly. This result suggests that our symbolic bug hunting method is a practical solution for block-level design verification.

4.3 Case Study: Verifying DLX with Alpha

We developed a testbench to check the functionalities of DLX using an Alpha processor. The testbench includes an instruction converter for transforming DLX and Alpha instructions. Since there are some architectural differences between DLX and Alpha implementations (e.g., forwarding logic), we did not convert every instruction included in DLX. For instance, jump and branch operations are not converted. In addition, the testbench tests DLX operations using non-pipeline mode by injecting four NOP (no operation) instructions after every randomly-generated instruction.

BugHunter found that the DLX correct model includes 4 bugs: LUI, SLT, JAL and all shift operations. Our heterogeneous design-verification method caught the last 2 bugs of DLX, as shown in TABLE 3 (the other two bugs were caught by performing sequential equivalence checking between a behavior Instruction-Set Simulator (ISS) and the correct DLX model, and the results are not shown due to space limitations).

Table 3: Results of Verifying DLX Using Alpha

	Heterogeneous Design Verification		
Bug	Run Time	Memory	Cycles
SLT	235.292	1103	12
LUI	44.687	838	12

Golden model plays an important role in verification. In case study A, the DLX correct design is used as the golden model to verify other design variants. Since the golden model is incorrect,

[2]Consistently, ABC with iprove [17] has better performance than other SAT solvers.

[3]BugHunter found four additional bugs in the DLX correct model that are confirmed by the designer. These bugs are removed from our DLX correct model.

Table 1: Verification results for DLX (SEC)

	Symbolic			Constrained-Random				Symbolic			Constrained-Random		
Bug#	Run time	Memory	Cycles	Run time	Memory	Cycles	Bug#	Run time	Memory	Cycles	Run time	Memory	Cycles
1	31.815	252	12	0.109	8.3	36	21	29.085	258	12	0.156	8.3	141
2	32.192	286	12	0.103	8.3	17	22	420.037	258	12	470.542	8.3	1092935
3	32.319	283	12	0.110	8.3	39	23	29.180	265	12	3.770	8.3	8565
4	32.228	279	12	0.247	8.3	361	24	381.419	254	12	214.059	8.3	499267
5	30.672	266	12	0.183	8.3	211	25	303.175	276	12	1267.371	8.3	2936454
6	32.415	271	12	4.152	8.3	9497	26	36.398	281	12	73.444	8.3	169386
7	31.070	272	12	0.930	8.3	1937	27	380.307	249	12	2402.056	8.3	5576017
8	32.596	284	12	0.161	8.3	147	28	32.160	279	12	10.439	8.3	24101
9	32.585	274	12	0.105	8.3	16	(29)	672.133	273	12	>2 hr.	8.3	>20M
10	32.243	281	12	0.105	8.3	16	30	31.615	271	12	6.857	8.3	15814
11	29.161	253	12	0.111	8.3	31	31	32.516	285	12	46.720	8.3	109082
12	32.584	282	12	0.102	8.3	18	32	29.822	259	12	1.214	8.3	2618
13	32.850	266	12	0.153	8.3	129	(33)	679.139	252	12	>2hr.	8.3	>20M
14	32.675	279	12	0.103	8.3	16	(34)	678.060	249	12	>2hr.	8.3	>20M
15	27.792	206	12	0.275	8.3	415	35	32.206	256	12	3.996	8.3	9122
16	32.944	285	12	0.101	8.3	17	36	613.277	266	12	>2hr.	8.3	>20M
17	32.724	267	12	0.101	8.3	20	37	391.901	260	12	54.262	8.3	126579
18	32.413	278	12	0.110	8.3	16	38	36.555	285	12	63.207	8.3	147803
19	27.271	248	12	0.562	8.3	1102	39	29.774	259	12	5.943	8.3	13569
20	28.648	261	12	0.271	8.3	404	40	32.720	287	12	0.103	8.3	20

Table 2: Verification results for Alpha (SEC)

	Symbolic			Constrained-Random				Symbolic			Constrained-Random		
Bug#	Run time	Memory	Cycles	Run time	Memory	Cycles	Bug#	Run time	Memory	Cycles	Run time	Memory	Cycles
1	44.091	391	12	0.275	8.9	727	6	54.265	418	12	152.047	8.9	710647
2	60.176	419	12	1670.588	8.9	7698703	7	57.563	583	12	3.391	8.9	14649
3	47.928	391	12	0.138	8.9	203	8	54.058	549	12	1.533	8.9	6839
4	53.111	549	12	0.256	8.9	773	9	52.445	420	12	0.115	8.9	97
5	57.062	437	12	3.837	8.9	17847	10	50.823	417	12	0.426	8.9	1539

the 4 new bugs in the DLX design can never be found by traditional equivalence checking or self-consistency checking. However, half of the bugs can be found by our heterogeneous design verification methodology, showing the power of this new methodology. Note that in practice it is often easy to find heterogeneous designs that have similar functionalities as the design under test, especially for processors and arithmetic cores. Writing testbenches is also simple because BugHunter supports symbolic simulation of behavior-level code. These characteristics ensure that this methodology can be applied to solve many practical design verification problems.

4.4 Case Study: FIR

FIR is a testcase used in Brinkmann's paper [3] that cannot be handled by the reported Boolean-based method. In this case study, we use our tool to prove that the equations are always equivalent. Table 4 shows the performance of our bug hunting method. From the table, we found that our method can solve the problem in 20 seconds even up to 64 bits. Although Brinkmann's technique has very short run time (< 0.01 second), it cannot be applied to generic designs with mixed word-level datapath and Boolean-level control logic. On the other hand, we can solve the problem using very short time and is much more flexible.

Table 4: Runtime of testcase FIR

Bit length	4	8	16	32	64
Run time	0.25	8.24	10.83	13.82	14.15

5. CONCLUSIONS

In this paper, we proposed several methods to enhance block-level bug hunting using high-level symbolic simulation. In addition, we integrated these features into a BugHunter framework and applied our methods to verify DLX, Alpha and FIR. Our empirical results demonstrate that BugHunter can reuse existing constrained-random testbenches and effectively catch all the bugs. Moreover, it can verify design correctness using heterogeneous designs, such as verifying DLX with Alpha shown in our case study, as well as improve the verifiability of the design under development. Thus, our BugHunter provides a robust, scalable, and flexible solution for design verification.

Acknowledgments

This research was supported by the National Science Council, Taiwan under Grant NSC 97-224-E-002-216-MY3; Excellent Research Projects of National Taiwan University, 95R0062-AE00-05; and Small Business Innovation Project of Ministry of Economic Affairs, Taiwan, 1Z960401.

6. REFERENCES

[1] J. Bhadra, M. Abadir, L.-C. Wang, and S. Ray, "A Survey of Hybrid Techniques for Functional Verification," *IEEE Design and Test of Computers*, vol. 24(2), pp. 112-122, 2007.
[2] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, pp. 677–691, Aug. 1986.
[3] R. Brinkmann and R. Drechsler, "RTL-Datapath Verification using Integer Linear Programming," *Proc. of IEEE ASPDAC*, pp. 741-746, 2002.
[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98(2), pp. 142-170, 1992.
[5] J. R. Burch, E. M. Clarke and D. E. Long, "Representing circuits more efficiently in symbolic model checking," *Proc. of IEEE/ACM DAC*, pp.403-407, 1991.
[6] J. L. Hennessy, and D. J. Patterson, "Computer Architecture: A Quantitative Approach," 2nd edition, Morgan Kaufman, 1996.
[7] C.-R. Ho, M. Theobald, M. M. Deneroff, R. O. Dror, J. Gagliardo and D. E. Shaw, "Early Formal Verification of Conditional Coverage Points to Identify Intrinsically Hard-to-Verify Logic," *Proc. of IEEE/ACM DAC*, pp. 268-271, Jun. 2008.
[8] R. B. Jones, C.-J. H. Seger, and D. L. Dill, "Self-consistency checking," *Proc. of IEEE FMCAD, LNCS*, vol. 1166, pp. 159–171, Springer-Verlag, 1996.
[9] C. Kern and M. R. Greenstreet, "Formal Verification In Hardware Design: A Survey," *ACM Trans. Des. Auto. Elec. Sys.*, vol. 4(2), pp. 123-193, Apr. 1999.
[10] A. Kolbl, J. Kukula and R. Damiano, "Symbolic RTL simulation," *Proc. of IEEE/ACM DAC*, pp. 47-52, 2001.
[11] A. Kolbl, J. Kukula, K. Antreich and R. Damiano, "Handling Special Constructs in Symbolic Simulation," *Proc. of IEEE/ACM DAC*, pp. 105-110, 2002.
[12] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *Int. J. Software Tools for Technology Transfer*, vol. 7(2), pp. 156-173, Apr. 2005.
[13] M. J. Rieschl, "Symbolic debug interface for register transfer simulator debugger," US Patent 6785884, 2004.
[14] C.- J. H. Seger, and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6(2), pp. 147–190, 1995.
[15] I. Wagner, V. Bertacco and T. Austin, "StressTest: An Automatic Approach to Test Generation Via Activity Monitors," *Proc. of IEEE/ACM DAC*, pp. 783-788, 2005.
[16] The VIS Group, "VIS: A system for Verification and Synthesis," *Proc. of Intl. Conf. on Computer Aided Verification, LNCS*, vol. 1102, pp. 428-432 Springer-Verlag, 1996.
[17] ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/abc.htm
[18] Avery Design Systems Inc., http://www.avery-design.com
[19] Bug UnderGround, http://bug.eecs.umich.edu