Reveal: A Formal Verification Tool for Verilog Designs

Zaher S. Andraus, Mark H. Liffiton, Karem A. Sakallah
{zandrawi, liffiton, karem}@umich.edu

Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122

Abstract. We describe the Reveal formal functional verification system and its application to four representative hardware test cases. Reveal employs counterexample-guided abstraction refinement, or CEGAR, and is suitable for verifying the complex control logic of designs with wide datapaths. Reveal performs automatic datapath abstraction yielding an approximation of the original design with a much smaller state space. This approximation is subsequently used to verify the correctness of control logic interactions. If the approximation proves to be too coarse, it is automatically refined based on the spurious counterexample it generates. Such refinement can be viewed as a form of on-demand "learning" similar in spirit to conflict-based learning in modern Boolean satisfiability solvers. The abstraction/refinement process is iterated until the design is shown to be correct or an actual design error is reported. The Reveal system allows some user control over the abstraction and refinement steps. This paper examines the effect on Reveal's performance of the various available options for abstraction and refinement. Based on our initial experience with this system, we believe that automating the verification for a useful class of hardware designs is now quite feasible.

1 Introduction

The paradigm of iterative abstraction and refinement has gained momentum in recent years as a particularly effective approach for the scalable verification of complex hardware and software systems. Dubbed *counterexample-guided abstraction refinement* (CEGAR), its power stems from the elimination (i.e., abstraction) of details that are irrelevant to the property being checked and from analyzing any spurious counterexamples to pinpoint and add just those details that are needed to refine the abstraction, i.e., to make it more precise. Originally pioneered by Kurshan [13], it has since been adopted by several researchers as a powerful means for coping with verification complexity. In particular, the use of abstraction-based verification has been thoroughly studied in the context of model checking by Clarke *et al.* [6] and Cousot and Cousot [7] for over two decades. Later methods by Clarke *et al.* [5], Das *et al.* [8] and Jain *et al.* [12] have successfully demonstrated the automation of abstraction and refinement in the context of model checking for safety properties.

Whereas such a verification paradigm is appealing at a conceptual level, its success in practice hinges on effective automation of the abstraction and refinement steps, as well as the various checking steps requiring sophisticated reasoning. In this paper, we describe how these issues are addressed by Reveal, an automatic CEGAR-based verification system. Reveal is used to formally verify complex hardware designs, including pipelined microprocessors whose RTL descriptions have tens of thousands of HDL source lines, thousands of signals, and hundreds of thousands of state bits.

Below, we will describe Reveal's CEGAR flow and analyze its behavior and performance by way of four representative test cases. For each test case, we compare a number of methods to model and check the desired properties on the abstract design, including the use of a *Satisfiability Modulo Theories* (SMT) solver [9]; we study trade-offs between various refinement options; we highlight the types of lemmas generated in the refinement stage and analyze the idiosyncrasies leading to them; we show how genuine bugs were discovered using Reveal; we provide experimental evidence that demonstrates the importance of datapath abstraction for the scalability of formal verification; and, finally, we compare the performance of Reveal against a number of existing automatic tools that perform formal verification of hardware, such as VCEGAR [12], BAT [15], UCLID [3], and VIS [10].

The rest of the paper is organized in four sections. Section 2 reviews Reveal's CEGAR framework, and Sections 3 and 4 describe our benchmark test cases and how they were verified using the Reveal system. Finally Section 5 summarizes the paper's conclusions.

2 The Reveal Verification System

Figure 1 depicts the flowchart of the reveal system. Reveal performs checks of safety properties on hardware designs described in the Verilog hardware description language (HDL). A typical usage scenario involves providing two Verilog descriptions of the same hardware design, such as a high-level specification and a detailed implementation, and checking them for functional equivalence. Reveal adopts the CEGAR-based approach of Andraus *et al.* [1], mainly involving:

Abstraction. The goal of abstraction is to obtain a compact representation of the design for which formal property checking is more likely to terminate (i.e., to scale both in time and space) than if applied directly on the original design. Reveal performs datapath and memory abstraction by replacing datapath units with *uninterpreted functions* and *predicates*, while leaving the control logic unabstracted. This allows reasoning about the complex control of the design, while avoiding the complexity introduced by datapath elements. Previous work (e.g. [4]) showed that it is possible to prove many useful (equivalence) properties if the abstract model is expressed in the logic of Equality with Uninterpreted Functions (EUF), a quantifier-free fragment of first order logic. Scalability can be further improved by abstracting to CLU, which extends EUF with counting arithmetic and lambda expressions for memories [3].

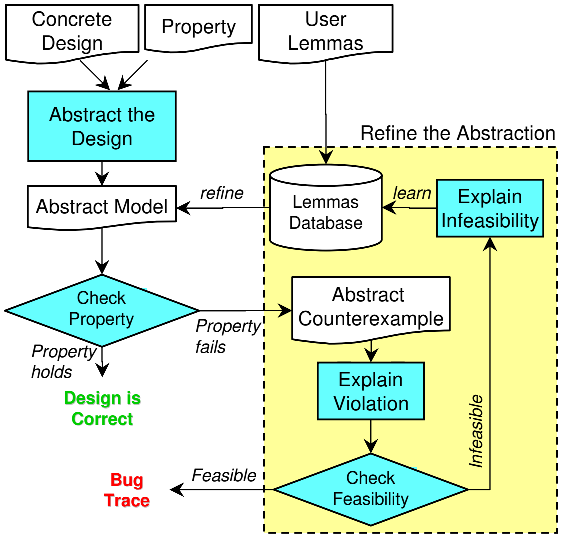


Fig. 1: The Reveal Flow

Table 1: Benchmark Statistics

Name	Verilog Lines	Verilog Signals	State Bits
Sorter	79	30	35 to 1027
DLX	$2.4\text{x}10^3$	399	$1.0\text{x}10^{11}$
Risc16F84	$1.2\text{x}10^3$	169	$1.0\text{x}10^5$
X86	$1.3\text{x}10^4$	$1.0\text{x}10^3$	$5.8\text{x}10^3$

Property Checking. Formal reasoning in the EUF or CLU logics determines if the abstracted design satisfies the specified property. Early EUF/CLU solvers convert the formula to an equi-satisfiable propositional formula and use an off-the-shelf Boolean solver to check for satisfiability. In contrast, Satisfiability Module Theories (SMT) solvers (e.g. YICES [9]) operate on these formulas directly by integrating specialized theory solvers within a backtrack propositional solver. SMT solvers, thus, are able to take advantage of the high-level semantics of the non-propositional constraints, while at the same time benefiting from the powerful reasoning capabilities of modern propositional SAT solvers. Reveal uses the YICES solver [9][17] in the property checking step, allowing the integration of the *empty theory* (consistency of term equality), *UF theory*, and *integer theories* (for counting).

Refinement. An abstract counterexample, demonstrating that the property is violated by the abstract model, has to be checked for feasibility on the concrete model. If feasible, a concrete counterexample trace is generated. If not, the counterexample is spurious, and is *refuted* in the abstract model by adding a blocking clause, similar to learning in SAT, and the process iterates. Reveal avoids the naïve refutation of one counterexample at a time, which usually leads to very slow convergence, rendering the approach impractical. Instead, one or more succinct explanations are used in each iteration to explain infeasibility and refine the abstraction for the next round of checking. These explanations, referred to as *lemmas*, are universal facts extracted from the concrete model to refute current or future spurious counterexamples and can thus be stored in a lemma database and re-used across invocations of Reveal on the same (family of) design(s). Preliminary results of this scheme [2] show that the convergence of the refinement loop is contingent upon the way these lemmas are derived. Reveal employs a reasoning engine that combines YICES with CAMUS [14] during feasibility and refinement. The CAMUS tool can derive one, multiple, or all minimally unsatisfiable subsets (MUSes for short) of constraints from a set of infeasible constraints. Finding MUSes allows for trimming the infeasibility explanations, effectively enlarging their footprint in the abstract solution space. Finding multiple MUSes means learning multiple lemmas in each refinement iteration and yielding a significant speedup in the convergence of the refinement loop. Finally, scalability is further improved by finding MUSes with the use of the *bit-vector theory* in YICES. Earlier methods (e.g. [1]) use a bit-blasting approach in which the abstract counterexample is encoded with propositional constraints and passed to a propositional solver. In contrast, reasoning at the word-level with the bit-vector theory in YICES allows much more efficient derivation of MUSes (i.e., lemmas).

3 Case Studies

We performed our experiments on the four designs which we briefly describe in this section. Table 1 summarizes the design statistics. Interested readers can find more details in [2].

Sorter Case Study. The Sorter design implements two versions of an algorithm that sorts four bit-vectors. The computation delay in both versions is 3 cycles. The property we verified is the equality between corresponding outputs in the two versions. All the bit-vectors in the two units, including the inputs and the outputs, have bit-width W, which we vary from 2 to 64 to see the effect of the datapath width on the scalability of each tool. Reveal's performance on the Sorter is presented in Figure 2, and will be analyzed in Section 4.

DLX Case Study. DLX is a 32-bit RISC microprocessor [11]. Its salient features include a 32-bit address space with separate instruction and data memories, a 32-word register file with two read ports and one write port, and 38 op-codes for arithmetic, logical, and control operations. Our case study involved proving the equivalence of two versions of DLX [18]. The first version, which we will refer to as *DLXSpec*, is a single-cycle implementation of the instruction set architecture (ISA) and serves as the architectural specification of the microprocessor. The second version, labeled *DLXImpl*, is a standard 5-stage pipeline.

RISC16F84 Case Study. This design is an implementation of the Risc16F84 microcontroller [20]. It has a 213x14-bit instruction memory, a 29x8-bit data memory, 34 op-codes, and a 4-stage pipeline. Similarly to the DLX case, we denote the implementation and specification by *OCImpl* and *OCSpec* respectively. *OCImpl* processes one instruction every four cycles, while *OCSpec* needs one cycle to process each instruction.

X86 Case Study. The X86 design is an open source RTL Verilog model developed at IIT Madras that implements Intel's IA-32 ISA [19]. The design's *Decoder* module is responsible for fetching an instruction prefix from memory, finding the total length of the instruction, and fetching and decoding the rest of the instruction. We verified the property that the Decoder activates the corresponding decode unit (Integer versus Floating Point) when the instruction is confined to a set of 6 integer and floating point op-codes.

4 Results and Analysis

We verified a number of buggy and bug-free variations of each of the aforementioned designs. The buggy versions were obtained by injecting errors in the RTL description. These variations are described in Table 2. Columns labeled T, I, and L, describe, respectively, total run-time (seconds), number of iterations, and total number of refinement lemmas (when applicable). 'TO' stands for "Time Out" (600s). Finally, the smallest run-time is highlighted in bold in each row; there can be multiple in each row when the difference is insignificant.

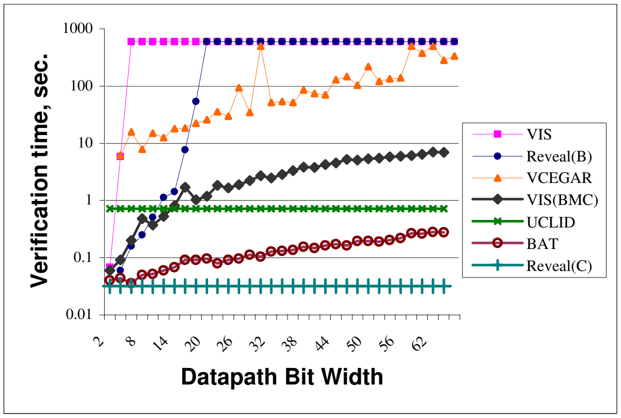
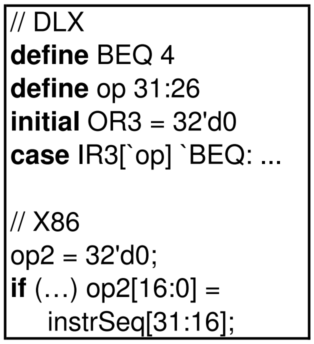


Fig. 2: Runtime Graphs for Sorter



```
// DLX
define BEQ 4
define op 31:26
initial OR3 = 32'd0
case IR3[`op] `BEQ: ...

// X86
op2 = 32'd0;
if (...) op2[16:0] =
    instrSeq[31:16];
```

Fig. 3: Verilog Code Fragment from DLX and X86

Reveal's modes were classified by a one-, two-, or three-letter code that indicates the abstraction and refinement options used. Abstraction options are labeled B (bit-level, i.e., no abstraction), C (CLU abstraction), and E (EUF abstraction). Refinement options are labeled V (refinement via refuting the abstract violation) and L (refinement with lemmas). For lemma refinement, S denotes refinement with a single lemma per iteration, while M denotes refinement with multiple lemmas. For example, the label CLM means CLU abstraction and refinement with multiple lemmas, whereas EV means EUF abstraction and refinement with the negation of the abstract violation.

Our empirical case study compares the performance of Reveal against the verification systems UCLID, BAT, VCEGAR, and VIS on a 2.2 GHz AMD Opteron processor with 8GB of RAM running Linux. UCLID [3] allows modeling of the datapath with abstract terms, and memories with Lambda expressions. BAT [15] models memories with set and get functions for reads and writes, respectively, but models the datapath with finite-length bit-vectors. VCEGAR [12] performs word-level predicate abstraction on the Verilog input but does not abstract memory arrays. Finally, VIS [10] uses, by default, bit-level reachability analysis to verify invariants. It can also be used in two special modes: one that performs bounded model checking of safety properties (denoted herein by VIS(BMC)), and another that performs invariant checking with a CEGAR algorithm based on hiding registers [16] (denoted by VIS(AR)).

#	Test Case/Version	CV		ELS		CLS		ELM			CLM			B
		T	I	T	I	T	I	T	I	L	T	I	L	T
D1	Bug-free DLX	TO	>1507	1.92	9	1.8	8	0.6	4	8	1.0	6	12	TO
D2	Pipeline "Stall" s-a-1	0.11	1	0.15	1	0.12	1	0.11	1	0	0.1	1	0	0.21
D3	Incorrect 'jump' address	3.16	45	2.22	11	1.16	5	1.13	3	5	1.1	4	8	6.7
R1	Bug-free RISC16F84	TO	>1767	TO	>1204	TO	>1085	257	93	185	148	68	170	209
R2	Floating "carry-in" signal	0.79	8	56	20	TO	>1881	72	44	13	40	33	39	15.2
R3	"aluout_zero_node" s-a-1	115	654	50	123	121	311	2.6	5	15	27.3	40	73	11.6
X1	Bug-free X86	TO	>388	TO	>1158	TO	>945	36.5	40	104	60.4	19	96	TO
X2	en_{Int} and en_{FP} swapped	TO	>461	TO	>1062	TO	>1046	30.5	78	161	103	24	86	TO
X3	Wrong FSM transitions	1.98	2	1.95	2	1.96	2	2.0	2	6	2.1	1	0	2.72
X4	CMP activates the FP unit	TO	>308	TO	>847	TO	>1252	23	12	41	58.7	7	43	TO

Fig. 4: Verification results for the DLX, RISC16F84, and X86 variations

4.1 Datapath Abstraction

The merits of datapath abstraction are evident in all verification runs. In particular, the performance of Reveal(C) and UCLID on the Sorter example is oblivious to the datapath bit-width W (Figure 2). In both cases, the abstract model is unaltered when the datapath bit width is changed; thus, the time needed to verify the abstract model is constant. Furthermore, the only interaction between the datapath and the control logic in this design involves bit-vector inequalities, allowing the CLU logic to prove the property without any refinement. The performance of the remaining tools degrades as W increases:

- VCEGAR takes 6.1 seconds to prove the property for W=2 as it incrementally discovers between 33 and 40 predicates within 58 to 130 iterations. Additionally, run-time grows exponentially with W. We suspect that the reason behind this is the expense of simulating the abstract counterexample on the concrete design in each refinement iteration, as well as the repeated generation of the abstract model each time a new predicate is added.

- The run-times of Reveal(B), VIS, and VIS(BMC) degrade rapidly as the bit width is increased. The run-times of VIS(AR) are similar to those of VIS and were removed from the graph to avoid clutter.
- BAT's performance degrades with increasing *W*, but BAT's reduction of the verification formulas to CNF appears to play an important role in keeping the run-time low.

Note that Reveal(B) (in Table 2) has the worst performance, though surprisingly it is able to terminate on a number of buggy versions. This is attributed to the ability of the bit-vector solver in YICES to efficiently find a satisfying assignment and thus its ability to find abstract counterexamples. However, the rest of the cases confirm that proving that a property *holds* is intractable without abstraction.

Finally, comparing Reveal(C) and Reveal(E) sheds some light on the difference between abstraction to EUF or CLU. In particular, Reveal(C) converges faster than Reveal(E) in terms of refinement iterations in the X86 and RISC16F84 cases. This is attributed to the heavy use of counters in these designs. Still, Reveal(E) outperforms Reveal(C) in most cases since the latter uses an integer solver which impacts overall performance.

4.2 Refinement Convergence

The performance of the various options in Reveal demonstrate the role of automatic refinement. In particular, Table 2 shows that the use of lemmas for refinement (modes ELS, CLS, ELM, and CLM) is far superior to refuting one counterexample at a time (mode CV). Also, using multiple lemmas in each refinement iteration (modes CLM and ELM) outperforms refinement with a single lemma at a time (modes ELS and CLS). The R2 case shows an interesting outlier; Reveal(CV) is significantly faster than any version that refines with lemmas. This is due to the heuristic nature of the satisfiability search for finding a bug. Any search, regardless of the refinement used, could "get lucky" and find a bug early, though only rarely.

To further assess the effect of lemmas on the convergence of the algorithm, we ran Reveal(C) on a version that combines the three bugs present in X2, X3, and X4. This was an iterative session, in which Reveal was re-invoked after correcting each reported bug. We tested Reveal in two modes: a mode in which learned lemmas are discarded after each run and a mode in which learned lemmas are saved and used across runs. The total run-time for the first mode was 232 seconds, whereas the run-time in the second mode was 166 seconds, a 40% improvement in speed. This confirmed our conjecture that lemmas discovered in one verification run can be profitably used in subsequent runs. The verification of real-life designs involves tens to hundreds of invocations of the tool, thus a significantly larger speed-up could be seen in practice.

4.3 Refinement Lemmas

We traced the source of refinement lemmas back to the original Verilog code involving control/datapath interactions. For example, most of the lemmas in the DLX example were related to the pipeline registers and control logic in *DLXImpl*, such as the lemma $(IR3 = 32'd0) \rightarrow (IR3\,[31:26]) \neq 6'd4$, which states that it's not possible to extract a non-zero field from a zero bit-vector. The source of the lemma is in Figure 3 and it involves IR3; the initial abstraction lost the fact that IR3[31:26] can not be equal to 4, and it found a spurious counterexample that executed the BEQ instruction. Another example is the set of lemmas in the RISC16F84, most of which are due to the *variable opcode width* feature, wherein the opcode field can be k-bits wide for any $k \in K = \{2, ..., 7, 14\}$. For instance, the opcode of the *goto* instruction is IR[13:11]=3'b101, while the opcode for *addlw* is IR[13:9]=5'b11111. The encoding guarantees that only one opcode is active at any given time. This information is lost when abstracting the bit-vector extraction operation. This results in the occurrence of lemmas of the form $(IR\,[13:k_1] = v_1) \rightarrow (IR\,[13:k_2] \neq v_2)$ for values $v_1 \neq v_2$ and distinct indices $k_1, k_2 \in K$.

It is worth mentioning that our experience with this flow shows that refinement lemmas can be very simple, or very complex, depending on the design and the property. In either case, the automatic discovery and (on-demand) refinement of only those relevant ones is an important enabler for the scalability of this approach.

4.4 Discovering Genuine Bugs

In addition to discovering artificially introduced bugs (e.g. those described in Table 2), Reveal was able to discover a number of *genuine* bugs. In particular, the RISC16F84 design includes the Verilog expression {1'b0,aluinp2_reg,c_in} in *OCImpl*, which uses a *floating* signal c_in as the carry-in bit to an 8-bit adder. In contrast, *OCSpec* performs addition without any carry-in bit. Reveal thus produces a counterexample showing the deviation with c_in assigned to 1. The unit designer acknowledged this problem and asserted that the simulation carried out for this design assumed c_in=0. An additional coding problem was discovered in X86; the RTL description includes the code given in Figure 3, which extracts a 16-bit displacement value from the instruction stream and assigns it to a 17-bit register. Most synthesis tools will zero-extend the RHS expression to make the sizes consistent, in which case the resulting model is still correct. Nonetheless, such an error may indicate additional problems in other units of the design.

4.5 Performance of VIS, VCEGAR, and UCLID

VIS, VCEGAR, and UCLID were not able to successfully terminate on the DLX, RISC16F84, or X86 designs. In some cases, the tool times out or exceeds available memory, and in others, an internal error causes unexpected termination. Details of these experiments can be found in [2].

5 Conclusions

We examined the performance of Reveal, a CEGAR-based formal verification system for safety properties in general, and equivalence in particular. Reveal is particularly suited for the verification of designs with wide datapaths and complex control logic. Datapath abstraction allows Reveal to focus on the control interactions making it possible to scale up to much larger designs than is possible if verification is carried out at the bit level. Additionally, Reveal's demand-based lemma generation capability eliminates one of the obstacles that had complicated the deployment of formal equivalence tools in the past. From a practical perspective, hands-free operation and support of Verilog allow Reveal to be directly used by designers. These capabilities were demonstrated by efficiently proving the existence of bugs, or proving the lack thereof, in four Verilog examples that emulate real-life designs both in terms of size and complexity.

6 Acknowledgements

This work was funded in part by the DARPA/MARCO Gigascale Systems Research Center, and in part by the National Science Foundation under ITR grant No. 0205288.

References

1. Z. Andraus, M. Liffiton, and K. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 19–24, 2006.
2. Z. Andraus, M. Liffiton, and K. Sakallah. CEGAR-based formal hardware verification: a case study. Technical Report CSE-TR-531-07, University of Michigan, 2007.
3. R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with Lambda expressions and uninterpreted functions. In *Proc. of Int'l Conference on Computer-Aided Verification,* LNCS 2404, pages 78–92, 2002.
4. J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of Int'l Conference on Computer-Aided Verification,* LNCS 818, pages 68–80, 1994.
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of Int'l Conference on Computer-Aided Verification*, pages 154–169, 2000.
6. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Sixth Annual ACM SI PLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 2006.

8. S. Das and D. Dill. Successive approximation of abstract transition relations. In *IEEE Symposium on Logic in Computer Science*, pages 51–58. 2001.
9. B. Dutertre and L. de Moura. A fast linear arithmetic solver for DPLL(T). In *Proc. of Int'l Conference on Computer-Aided Verification,* LNCS 4144, pages 81–94, 2006.
10. R. Brayton et al. VIS: a system for verfication and synthesis. In *Proc. of Int'l Conference on Computer-Aided Verification,* LNCS 1102, pages 428–432, 1996.
11. J. Hensessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 2^{nd} edition, 1996.
12. H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word-level predicate abstraction and refinement for verifying rtl Verilog. In *Proc. of Design Automation Conference*, pages 445–450, 2005.
13. R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoritic Approach.* Princeton University Press, 1999.
14. M. Liffiton and K. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.
15. P. Manolios, S. Srinivasan, and D. Vroon. Automatic memory reductions for rtl model verification. In *Proc. of Int'l Conference on Computer-Aided Design*, pages 786–793, 2006.
16. F. Wang, B. Li, H. Jin, G. Hachtel, and F. Somenzi. Improving Ariadne's Bundle by following multiple threads in abstraction refinement. In *Proc. of Int'l Conference on Computer-Aided Design*, pages 408–415, 2003.
17. http://yices.csl.sri.com/.
18. http://www.eecs.umich.edu/vips/stresstest.html.
19. http://vlsi.cs.iitm.ernet.in/x86_proj/x86Homepage.html.
20. http://www.opencores.org.