Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers

Per Bjesse[1], Tim Leonard[2], and Abdel Mokkedem[2]

[1] Chalmers University of Technology, Sweden
bjesse@cs.chalmers.se
[2] Compaq Computer Corporation, USA
{tim.leonard,abdel.mokkedem}@compaq.com

Abstract. We describe the techniques we have used to search for bugs in the memory subsystem of a next-generation Alpha microprocessor. Our approach is based on two model checking methods that use satisfiability (SAT) solvers rather than binary decision diagrams (BDDs).
We show that the first method, bounded model checking, can reduce the verification runtime from days to minutes on real, deep, microprocessor bugs when compared to a state-of-the-art BDD-based model checker. We also present experimental results showing that the second method, a version of symbolic trajectory evaluation that uses SAT-solvers instead of BDDs, can find as deep bugs, with even shorter runtimes. The tradeoff is that we have to spend more time writing specifications.
Finally, we present our experiences with the two SAT-solvers that we have used, and give guidelines for applying a combination of bounded model checking and symbolic trajectory evaluation to industrial strength verification.
The bugs we have found are significantly more complex than those previously found with methods based on SAT-solvers.

1 Introduction

Getting microprocessors right is a hard problem, with harsh punishments for failure. With current design methods, hundreds to thousands of bugs must be found and removed during the design of a new processor, and there are heavy economic incentives to get most of them out before first silicon.

Current designs are so complex that simulation-based methods are no longer adequate. Most companies in the industry, including at least AMD, Compaq, HP, IBM, Intel, Motorola, and Sun, have therefore investigated formal verification. Their choices of methods, tools, and application areas have varied, as has their level of success.

One of the areas we have concentrated on at Compaq is property verification for our microprocessor designs. Among other things, we have investigated the use of symbolic model checking [9] to find *Register Transfer Level* (RTL) bugs in a next-generation Alpha processor. Our goal in this work has been to find bugs, rather than to prove their absence, since there are many bugs to find in a design under development.

G. Berry, H. Comon, and A. Finkel (Eds.): CAV 2001, LNCS 2102, pp. 454–464, 2001.

Our initial experiments with symbolic model checking convinced us that the capacity limits of many model checkers prevent us from finding bugs cost effectively. The best model checker we could find, an experimental version of Cadence SMV [10], needs several hours to days to check simple properties of heavily reduced components. As a consequence, we have also looked at model checking using satisfiability (SAT) solvers [3,2,16]. These methods have shown real promise, especially for finding bugs, when compared to BDD-based model checkers like SMV.

In this paper, we describe how we have applied two SAT-based verification techniques to find real bugs in the memory subsystem of the Alpha chip. The first technique, bounded model checking (BMC) [3], has previously been applied to industrial verification, but not for finding bugs of length anywhere near what we will describe. The second of these techniques, symbolic trajectory evaluation (STE) [12], has previously not been used together with SAT-solvers at all.

We compare the performance of SAT-based bounded model checking to state-of-the-art BDD-based model checking, and present results showing the usefulness of SAT-based STE. Our experiences are very positive: the use of SAT-based methods has reduced the time for finding certain bugs from days to a few minutes. We also compare the performance, when finding bugs in real designs, of the two SAT-solvers we have used: GRASP [15], and Prover Technology's PROVER [14] proof engine. Finally, we present guidelines for applying a combination of BMC and SAT-based STE to microprocessor bug finding.

Related Work. *Bounded model checking* [3] (BMC) was invented by Biere and coworkers as a method for using SAT-solvers to do model checking. BMC has previously been applied to bug finding for Power PC chips [4]. To our knowledge, BMC is the only SAT-based model checking method that has been used in realistic microprocessor verification.

In the Power PC verification, the authors did not model the environment of the designs under analysis. BMC quickly found short counterexamples to the properties being verified, but they were false failures due to illegal input sequences. BMC did well at this compared to BDD-based model checking, but the results said little about whether BMC could find real bugs, which are generally much deeper. We, on the other hand, present the results of searching for, and finding, real, deep bugs. One of our important contributions is therefore that we demonstrate that BMC together with cutting edge SAT-solvers has the capacity to find realistic bugs in industrial designs.

Symbolic trajectory evaluation (STE) is a model checking method invented by Seger and Bryant [12] that consists of an interesting mix of abstract interpretation and symbolic evaluation. STE is in industrial use, primarily for data path and memory verification, at companies including Intel [1] and Motorola. Up to now, STE has always been implemented using BDDs; the use of SAT-solvers to do STE has not been reported previously in the literature. Moreover, we apply symbolic trajectory evaluation to verification at the synchronous gate level—a fairly high level of abstraction for STE, which has previously been used predominantly at the transistor level.

There are other ways of doing SAT-based model checking than the ones that we discuss in this paper. We refer readers interested in these alternative approaches to [2,16,7,13,5].

The paper is organised as follows. In Sections 3 and 4, we give brief introductions to BMC and STE. We then describe the component that we have focused on, the merge buffer, and the process we have used to analyse it. After that, we go on to describe the actual use of the verification tools and the results. Finally, we give guidelines for using a combination of BMC and STE for heavy-duty industrial verification.

2 Preliminaries

In this paper, we will search for counterexamples to properties of synchronous gate-level hardware. Such circuits can be viewed as finite transition systems, where the states are value assignments to a vector $s = (s.0, \ldots, s.n)$ of boolean variables called the system's *state variables* [6]. The transition system for a given circuit can be represented as two propositional logic formulas [2]:

$\mathit{Init}(s)$	Initial states formula
$\mathit{Trans}(s, s')$	Transition relation formula

The first formula, *Init*, is a formula that characterises the initial states by evaluating to true exactly for the assignments to the state variables that are initial states. The second formula, *Trans*, evaluates to true for s and s' precisely when there is a transition from the state assigned to s to the state assigned to s'.

Our analyses take as inputs the formulas *Init* and *Trans* together with a description of a property to check. Such a property might for example be "a store instruction to an IO address is never discarded." The aim of the analyses is then to generate a trace, if one exists, where an IO store is thrown away.

In the case of BMC, we will specifically focus on detecting failures of *safety properties*. Informally, safety properties are properties of the form "in every reachable state of the system, the property P holds."

3 Bounded Model Checking

Bounded model checking tries to find bugs in a system by constructing a formula that is satisfiable precisely if there exists a length N or shorter trace violating a property given by the user. The BMC procedure feeds this formula to an external SAT-solver, and uses the returned assignment (if any) to extract a *failure trace*.

The bound N is given by the user, and will affect both the size of the generated formulas, and the length of the failure trace that can be detected. A negative answer from the SAT-solver for a given N does not mean that the whole system is safe, only that there are no failure traces of length N or shorter. BMC is thus used to find bugs, rather than to prove their absence.

We assume that the safety property we are interested in has been encoded as a propositional logic formula $Prop(s)$ that will evaluate to true exactly for the states fulfilling the property. Given the bound N, and the formulas $Init(s)$, $Trans(s, s')$, and $Prop(s)$, the BMC procedure constructs the following formula, which characterises failure traces of length N or shorter:

$$\begin{aligned} &Init(s_1) \wedge \\ &Trans(s_1, s_2) \wedge \ldots \wedge Trans(s_{N-1}, s_N) \wedge \\ &(\neg Prop(s_1) \vee \ldots \vee \neg Prop(s_N)) \end{aligned}$$

If the SAT-solver returns an assignment to the state variables in $s_1 \ldots s_N$ that makes this formula true, then there exists an initial state s_1 in the system, from which we can reach another state s_k ($k \in \{1 \ldots N\}$) where the property fails. The BMC procedure can thus extract a failure trace from the assignment.

4 Symbolic Trajectory Evaluation

A symbolic trajectory evaluator takes $Trans(s, s')$ as input together with a so called *trajectory assertion* of the form $Ant \Rrightarrow Cons$. The antecedent and consequent of the trajectory assertion, Ant and $Cons$, are lists of equal length, in each of which the ith entry says something about the system's state variables at time i. Informally, a trajectory assertion will be true with respect to a system if a trace of the system that agrees with the antecedent necessarily must agree with the consequent. The objective of symbolic trajectory evaluation is to generate a failure trace for the system that satisfies the antecedent, and violates the consequent.

As an example, assume that we have constructed a circuit whose state variables $s.a$ and $s.b$ should contain the **or** and the **and**, respectively, of the current and previous value of the state variable $s.i$. The following trajectory assertion specifies this property:

$$\begin{gathered} [\texttt{node}\ s.i\ \texttt{is}\ x, \texttt{node}\ s.i\ \texttt{is}\ y] \\ \Rrightarrow \\ [\langle\cdot\rangle, \texttt{node}\ s.a\ \texttt{is}\ x \vee y\ \texttt{and}\ \texttt{node}\ s.b\ \texttt{is}\ x \wedge y] \end{gathered}$$

Here $\langle\cdot\rangle$ means "no requirements on the state variables", so the assertion can be read, "if we have a trace of the system where $s.i$ contains the value x at some time t, and $s.i$ contains the value y at time $t+1$, then at time $t+1$ $s.a$ and $s.b$ contains the logical **or** and the logical **and** of x and y, respectively."

In order to generate a failure trace, the trajectory evaluator first computes a boolean expression ok over the user-introduced variables x and y. This expression has the property that it evaluates to true for the assignments to x and y for which the antecedent guarantees the consequent (and no others). A key element of symbolic trajectory evaluation is that ok is constructed by symbolic reasoning in a four-valued logic. In addition to the two standard values *True* and *False*, the four-valued logic contains the values X (unknown), and $\top$ (overspecified).

The value X is used to model unknown contents of state variables, and the value $\top$ is used to model the contents of state variables that are required to contain two different values at the same time.

When *ok* has been computed, the evaluator uses an external SAT-solver to check whether there exists an assignment to x and y that makes *ok* evaluate to false. If there exists such an assignment, there is a trace of the circuit that is consistent with the antecedent but violates the consequent. The trajectory evaluator then instantiates x and y with the falsifying values, and constructs a failure trace that is given back to the user.

5 The Merge Buffer

Alpha processors, like most state-of-the-art microprocessors, have a very hierarchical structure. A processor is divided into a handful of so called *boxes*, each responsible for dealing with a particular aspect of instruction execution. For example, the IBox handles instruction fetch, and the MBox executes memory-reference instructions. Each box is further divided into a handful of parts that we will call subboxes.

The subbox that is the focus of our attention in this paper is the *merge buffer*, an important component of the MBox for a next-generation Alpha chip. We chose the merge buffer as it is one of the most complex subboxes in the processor. Our hope is that if we can cost-effectively find bugs in this component, then we can use the same methods on most other subboxes.

The function of the merge buffer is to receive requests to write into memory, and to reduce the traffic on the memory bus by merging stores to the same physical address. In order to do the merging correctly, the merge buffer communicates with four other subboxes: (1) the *store queue*, where store instructions are saved until they are written out of the merge buffer; (2) the *load queue*, where load instructions are stored until they have received results from memory; (3) the *CBox*, which deals with the cache coherence protocol; and (4) the *backend tag module*.

The merge buffer is essentially a large buffer with a very complex policy for reading in entries, merging stores, and writing out stores to the memory. It has about 14 400 latches, 400 primary inputs, and 15 pipeline stages. The pipeline has complex feedback that prevents us from retiming away latches.

6 Analysis Cycle

In Figure 1 we show the analysis cycle that we have used to locate bugs in the merge buffer.

We start off with the original RTL description of the circuit. As the full-size merge buffer contains more than ten thousand latches—too much state to be feasible to verify using standard model checking technology—we need to reduce the size of the model. The idea is to remove portions of the state in the circuit in ways that do not alter the circuit behaviour with respect to the properties of interest. The most important reductions are *symmetry reductions* [8], which we

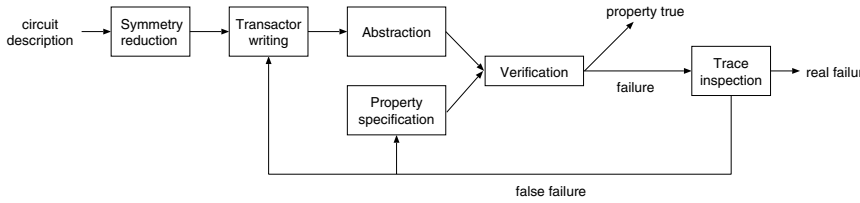

Fig. 1. Our Verification Flow.

use to reduce the number of buffer entries, address bits per entry, data bytes per entry, and bits per data byte.

We do not mind if some of our reductions do not preserve all possible properties of the circuit, as long as we can find problems in the reduced circuit that also are present in the full size circuit. The reason for this is that we are interested in finding bugs, as opposed to proving correctness. We are thus permitted to do ad-hoc reductions that are formally incorrect, but that preserve most of the interesting behaviour of the circuit.

After the reductions, the merge buffer has about 40 primary inputs. When the merge buffer is in use, these inputs will be connected to the four subboxes with which the merge buffer communicates. If we leave them unrestricted, the verification will be done under the assumption that any inputs can occur at any time. However, in order to function correctly, the merge buffer relies on assumptions about the behaviour of its environment. We therefore have to restrict the input to the merge buffer by adding *transactor* state machines that provide a verification environment that rules out input behaviours that could not arise in real use.

We then abstract the resulting circuit in two ways. First, we use an RTL compiler to optimise the circuit by performing transformations like constant propagation and common subexpression elimination. The reduced merge buffer now has about 1800 latches and 10 free primary inputs. We then do a final abstraction step that removes redundant latches, and replaces groups of transparent latches with standard flip-flops (a single transparent latch can not be modelled synchronously, but we can often model clusters of transparent latches). The final model has about 600 state nodes in the cone of most properties.

The end result of the reductions and abstractions is the model that we give to the verification tools. However, before we can do that, we need to write down the property of interest in a format that the tool we want to use accepts. Given the model and the property, the verification tool then either produces a failure trace, or tells us that the property is true (which has little meaning as we have performed ad-hoc reductions).

A lot of design knowledge is needed to decipher a failure trace; a property can fail for more than one reason. First of all, we might have made a specification mistake that causes the tool to diagnose an intended behaviour of the system as a failure. In this case we need to modify the property. Second, the trace might be a trace that the real system could not exhibit, because it has arisen due to the

merge buffer's environment providing input signals that cannot occur in real-life. In this case we need to go back and modify the transactors so that we disallow this behaviour, and re-abstract the resulting model. Third, we might have found a real bug.

7 Verification

In this section, we describe our experiences of applying BDD-based symbolic model checking, BMC, and STE to the merge buffer. The areas of the merge buffer that we target have previously been well explored with simulation-based verification.

7.1 BDD-Based Symbolic Model Checking

SMV was the first BDD-based tool that we evaluated that showed some promise for checking non-trivial merge buffer properties. (We have evaluated several.) However, most of the interesting merge-buffer properties contain about 600 latches in the cone of influence, and BDD-based model checking of state machines containing more than a couple of hundred latches is highly non-trivial. In order to find bugs using SMV, we therefore have to decrease the size of the cone by setting a subset of the 10 free primary inputs to specific values during the run. These values restrict the part of the state space that we explore using the model checker.

In order to get better performance out of SMV, we have ported it to the 64-bit Alpha architecture. This allows us the benefits of performing the model checking runs on a high performance server with 8 GB of main memory. To further improve SMVs capacity, we have also augmented the standard variable reordering heuristics with two special purpose tactics.

In spite of the improvements to SMV, each property still takes several hours to explore on the server. We have found many bugs this way, but it is slow.

7.2 Bounded Model Checking

The first alternative to BDD-based model checking that we have investigated is bounded model checking, as implemented in the SAT-based model checking workbench FixIt [2].

One of the SAT-solvers that we wanted to use together with FixIt, Prover [14], was not available for the Alpha architecture when this work was done. We have therefore done all of our BMC runs on a 32-bit PC. The performance of the BMC analysis is still remarkable. Even though we are not using a high-performance processor with many gigabytes of memory, we can find failures in a fraction of the time needed by SMV. In Table 1 we compare the runtimes of BMC, running on a 450 MHz 32-bit PC, to SMV, running on a 700 MHz 64-bit Alpha.

The first column of BMC runtimes is obtained using Captain Prove, a command-line tool from Prover Technology. Captain Prove uses Prover's

Table 1. Comparison between Bounded Model Checking and SMV.

Failure length	SMV sec	CAPTAIN PROVE BMC sec	GRASP BMC sec
25	62 280	85	25
26	32 940	19	19
34	11 290	586	272
38	18 600	39	101
53	54 360	1 995	[>10000 s]
56	44 640	2 337	[>10000 s]
76	27 130	619	6 150
144	44 550	10 820	[>10000 s]

application programming interface [11] to search for models using *strategies*. A simple such strategy, which we will refer to as the *timed strategy*, looks as follows:

```
sat 1 time 3600.
back level 5 [ sat 1 time 30. ].
```

The timed strategy first does a preprocessing step called *1-saturation* [14] for 3600 seconds. This analysis tries to find information restricting the search space we have to traverse for a model. The 1-saturation is then followed by the actual search, *backtracking*. At every fifth level of the search tree, the SAT-solver is instructed to do 30 seconds of additional 1-saturation.

The use of strategies allows us to control the search for assignments. We use different choices of strategies for different bounds N. When N is less than 40, we use the default strategy of 1-saturation without a time limit followed by normal backtracking. For N larger than 40, we use the timed strategy with different values for the initial 1-saturation. For example, for length 60 traces we normally need 1000 seconds of initial saturation, whereas for traces over 100 cycles long we use 10 000 or 20 000 seconds of initial saturation.

As can be seen from Table 1, BMC using CAPTAIN PROVE detects the failures significantly faster than SMV. In some cases it reduces the runtime for finding a bug from a day to a couple of minutes. The lengths of failures that are detected range from 25 cycles up to well over a hundred cycles.

The second column of BMC runtimes is obtained using GRASP [15], a high-capacity public domain SAT-solver. As can be seen in the table, CAPTAIN PROVE and GRASP both work well for short failures. For longer failures, CAPTAIN PROVE outperforms GRASP. (Please note that the reason for the [>10000 s] table entries is that GRASP automatically terminates after 10 000 seconds; we have not cut it off.)

7.3 SAT-Based Symbolic Trajectory Evaluation

The second alternative to BDD-based model checking that we have investigated is a SAT-based version of symbolic trajectory evaluation that we have implemented in FIXIT.

The advantage of using STE instead of BMC is that we are not forced to give symbolic values to each time-instance of a state variable. Instead we can choose to give concrete values to some state variables, or leave them to contain X. This potentially permits us to do much deeper exploration of the state-space than we can do using BMC, while preserving the short run times.

However, in order to take full advantage of this increased flexibility, we have to spend more time coming up with a good specification that judiciously gives concrete and symbolic values to the right variables.

For example, if we do not give concrete or symbolic values to some of the state variables, they are initialised to contain the unknown value X. This value often propagates, since it may be impossible to draw conclusions about the outputs of a gate with an unknown input. We might also have forgotten to assign a value to a primary input at an important time. When a property fails because of such underspecification, we have to make the specification more detailed by introducing symbolic or concrete values. A given STE specification will thus often have to go through several iterations of revision.

Table 2. Runtimes for detecting failures using symbolic trajectory evaluation.

Failure length	CAPTAIN PROVE sec	GRASP sec
77	7.7	33.3
77	7.7	34.2
112	10.8	51.9
123	11.7	51.9

In Table 2, we present the runtimes needed to find four bugs in the merge buffer using STE. The times to do the actual detections are short, but we had to spend a lot of time developing the specifications. Luckily, the turnaround time for discovering that an assertion is underspecified is a few seconds at most, which means that the specification work is very interactive.

The table shows a clear difference between the performance of STE using GRASP and CAPTAIN PROVE. However, the actual runtimes are very low in both cases. For the purpose of using SAT-based STE to locate bugs in the merge buffer, we can clearly make do with a public domain SAT solver.

8 A Proposal for a Methodology

From the previous section, it is clear that BDD-based model checking, BMC, and STE have very different characteristics. Based on the experiences we have had while locating design errors in the merge buffer, we have the following suggestion for a methodology:

- Start the analysis of a new subbox with bounded model checking.
- Initially test a new property with a small bound, so that the check only takes a few seconds. This will catch low-hanging fruit, and alert us to simple problems with inputs that are not properly constrained.
- Remove false counterexamples by modifying the transactors or the property, as appropriate.
- Start looking for long failures of the property. Choose a small set of bounds, ranging from medium long up to very challenging, and check each of them using the timed Captain Prove strategy. Use longer and longer saturation times.
- Use STE to quickly check that the problem is fixed whenever the designers have corrected a bug found using BMC. Also abstract the failure trace by making some of the inputs or control signals symbolic. This allows quick checking for failures that are similar to the original failure.
- When the BMC checks start taking more than half an hour or so, start working in parallel on using STE to find the bug.
- If neither BMC nor STE seems to find any failures, try SMV or move on to another property.

9 Conclusions

In this paper, we have presented the techniques that we have used to find bugs in a crucial component of a microprocessor in design. Our approach is based on bounded model checking and a SAT-based version of symbolic trajectory evaluation that we have developed.

Our experimental results demonstrate that it is possible for BMC to outperform state-of-the-art BDD-based symbolic model checking by two orders of magnitude, even when we look for bugs in deeply pipelined industrial components. None of the bugs described here has been a false counterexample. As a result, their complexity in terms of the length of minimum failure traces has been significantly larger than previously have been found using SAT-based techniques.

We have had less time to evaluate the use of SAT-based STE, but it seems clear that it is a very attractive bug-finding method. We have used STE to find bugs as deep as the ones we have been able to find using BMC, with negligible runtimes. However, this does not come for free; we have decreased the tool's runtime by spending more time developing specifications.

We have also presented a comparison of the performance of Captain Prove and GRASP for BMC and STE, and suggested a methodology for SAT-based industrial bug finding.

We believe that the approach we have presented here can be cost effective, and that the techniques we have used will become vital instruments in the standard verification toolbox. During the two months when the work that is presented in this paper was done, we improved the SAT-based framework FixIt significantly and removed many bottlenecks that we had not encountered on academic examples. The dramatic decrease in runtimes that we achieved in this short time makes us believe that there is a large potential for further improvement.

Acknowledgements

Many thanks to Gunnar Andersson, Luis Baptista, Arne Borälv, and João Marques Silva, who gave advice on running the SAT-solvers. We would also like to thank Gabriel Bischoff, John Matthews and Mary Sheeran for their useful comments on earlier drafts of this paper. Finally, Per Bjesse thanks Compaq's Alpha Development group for hosting him during the autumn of 2000.

References

1. M. Aagaard, R.B. Jones, T.F. Melham, J.W. O'Leary, and C.-J. H. Seger. A methodology for large-scale hardware verification. In *Formal Methods in Computer Aided Design*, November 2000.
2. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Proc. TACAS '00,* 9^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2000.
3. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS '99,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.
4. A. Biere, E.M. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs. In *Proc.* 11^{th} *Int. Conf. on Computer Aided Verification*, 1999.
5. P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer Aided Design*, November 2000.
6. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, December 1999.
7. A. Gupta, Z. Yang, and P. Ashar. SAT-based image computation with application in reachability analysis for verification. In *Formal Methods in Computer Aided Design*, November 2000.
8. C.N. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.
9. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
10. K.L. McMillan. The SMV language. Technical report, Cadence Berkeley Labs, 1999.
11. Prover Technology AB. *Prover 4.0 Application Programming Reference Manual*, 2000. PPI-01-ARM-1.
12. C.-J. H. Seger and R.E. Bryant. Formal verification by symbolic evaluation of partially ordered trajectories. *Formal Methods in System Design*, 6(2):147–190, March 1995.
13. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, November 2000.
14. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, January 2000.
15. J.P.M. Silva. *Search algorithms for satisfiability problems in combinational switching circuits.* PhD thesis, EECS Department, University of Michigan, May 1995.
16. P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, 2000.