View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 125 (2005) 3–12

www.elsevier.com/locate/entcs

Combining SAT Methods with Non-Clausal Decision Heuristics [1]

Clark Barrett [2]

New York University

Jacob Donham [3]

Abstract

A decision procedure for arbitrary first-order formulas can be viewed as combining a propositional search with a decision procedure for conjunctions of first-order literals, so Boolean SAT methods can be used for the propositional search in order to improve the performance of the overall decision procedure. We show how to combine some Boolean SAT methods with non-clausal heuristics developed for first-order decision procedures. The combination of methods leads to a smaller number of decisions than either method alone.

Keywords: satisfiability modulo theories, Boolean satisfiability, non-clausal, decision heuristics, CVC Lite

1 Introduction

Decision procedures for domain-specific first-order theories and combinations of such theories are useful in applications such as hardware verification, translation validation, extended static checking, and proof-carrying code. These first-order decision procedures are based on core algorithms that decide the satisfiability of a conjunction of literals. In order to decide arbitrary formulas,

[1] This research was supported by a grant from Intel Corporation. The content of this paper does not necessarily reflect the position or the policy of Intel.

[2] Email: barrett@cs.nyu.edu

[3] Email: jake@bitmechanic.com

1571-0661 © 2005 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.09.042

we must layer a propositional satisfiability procedure on top of the first-order procedure.

We can view the overall process as follows: Form a *propositional abstraction* of the formula by replacing each distinct atomic formula with a Boolean variable; find a variable assignment which satisfies the propositional abstraction; convert the assignment to a conjunction of first-order literals by replacing each Boolean variable assigned *true* or *false* with the corresponding atomic formula or its negation, respectively; finally, check that the conjunction of literals is satisfiable using the first-order decision procedure.

For large formulas with significant Boolean structure, the size of the propositional search tree dominates the overall performance, so heuristics and clever search algorithms for SAT are important. We can combine SAT methods with non-clausal heuristics developed for first-order decision procedures to obtain a method which takes fewer decisions to decide a formula than either one by itself. Section 2 reviews existing methods for propositional satisfiability and describes some non-clausal search heuristics. Section 3 describes our implementation combining these methods, and Section 4 gives quantitative results obtained using CVC Lite [2], a proof-producing decision procedure for a combination of theories without quantifiers.

2 Efficient SAT Algorithms

The essence of the standard Davis-Putnam-Logemann-Loveland (DPLL) algorithm for SAT [5,6] is shown in in Figure 1. It explores the space of partial variable assignments depth-first and checks each one to see if it satisfies the formula. The variable ϕ represents the formula under consideration, and Δ represents the partial assignment so far. If ϕ simplifies to $\top$ under Δ then Δ is a satisfying assignment. If ϕ simplifies to $\bot$ then Δ is an unsatisfying assignment. If ϕ simplifies to neither $\top$ nor $\bot$, then the algorithm chooses an unassigned variable (a *splitter*; such a choice is a *decision*), and then calls `checkSat` on the simplified formula along with Δ augmented first with the splitter assigned to $\top$ and then to $\bot$. The algorithm incrementally builds a partial assignment until the assignment satisfies the formula or exhausts the tree of possible assignments.

Many modern SAT solvers like GRASP [11] and Chaff [12] are based on refinements of the basic DPLL algorithm. The two most important refinements are *Boolean constraint propagation* and *conflict clauses*.

Boolean constraint propagation (BCP) takes advantage of the fact that for a formula in conjunctive normal form (CNF), every clause must be satisfied by a satisfying assignment. So if there are n literals in a clause and $n-1$ of them

```
checkSat(Δ, φ)
  φ′ = simplify(Δ, φ);
  if (φ′ ∈ (⊤, ⊥)) return φ′;
  α = findSplitter(φ′);
  if (checkSat(Δ ∪ {α = ⊤}, φ′) == ⊤) return ⊤;
  if (checkSat(Δ ∪ {α = ⊥}, φ′) == ⊤) return ⊤;
  return ⊥;
```

Fig. 1. Propositional DPLL algorithm

evaluate to $\bot$ under the current partial assignment (such clauses are called *unit* clauses), then the nth must evaluate to $\top$ in order to satisfy the clause. By propagating Boolean constraints until there are no more unit clauses the algorithm may deduce the values of many variables and avoid having to split on them.

If we find the formula to be unsatisfiable under a particular assignment Δ, then there is a minimal set $\delta \subseteq \Delta$ which makes the formula unsatisfiable. If the algorithm later generates a Δ' such that $\delta \subseteq \Delta'$ then it can immediately determine that the formula is unsatisfiable under Δ' and save some work. A *conflict clause* asserts that at least one assignment in δ is false. For example, if $\delta = \{a = \top, b = \bot, c = \top\}$ then the conflict clause is $(\neg a \vee b \vee \neg c)$. When ϕ simplifies to $\bot$ under some Δ, we find a minimal conflict set δ by tracing the *implication graph* describing how each variable got its value—whether directly from an assignment or through a chain of propagations from some set of assignments—and add a conflict clause derived from δ to ϕ. Then if a Δ' is generated such that $\delta \subseteq \Delta'$, BCP on the conflict clause cuts off the search tree immediately.

In the first-order version of DPLL, we replace variable assignments with first-order assumptions. Propositionally satisfying assignments are checked by submitting the conjunction of first-order literals induced by a propositional assignment to the first-order decision procedure. If the first-order decision procedure is *online* (like CVC Lite is) then first-order literals can be submitted as the partial assignment is built, rather than when the algorithm finds a propositionally satisfying assignment. The algorithm may then discover much earlier that a partial assignment does not first-order satisfy the formula.

In the first-order version of conflict clauses, the cause of a conflict can be richer than a simple implication graph over propositional variables. CVC Lite is a proof-producing decision procedure; it can generate a proof object giving justification of its conclusion [1]. So when ϕ simplifies to $\bot$ under Δ, CVC Lite produces a proof of that fact, and the assumptions that are used in the proof comprise exactly the subset of Δ that contributes to the conflict.

Other systems which use conflict clauses generated from first order decision procedures include CVC, ICS, and Verifun. CVC [4,13] (the predecessor to

CVC Lite) uses the same strategy of generating conflict clauses based on proof assumptions. The ICS decision procedure [7] does an optimized trial-and-error elimination of irrelevant literals in a clause rather than tracking dependencies on assumptions. Verifun [8] takes an intermediate approach: it cannot produce proofs, but it does track just enough dependency information to enable the production of conflict clauses.

2.1 Non-Clausal Decision Heuristics

A great difficulty of the DPLL algorithm is choosing splitters. The order in which splitters are chosen can have a huge impact on the performance of the algorithm, because a particular choice may prune a large subtree of the decision tree. SAT solvers such as Chaff incorporate decision heuristics which work well on many pure Boolean problems given in CNF. But we can do better by taking advantage of the structure of a non-clausal (i.e. non-CNF) formula to guide the search.

We have implemented the "depth-first-search" and "caching" heuristics that were developed for SVC [3,10] (a predecessor of CVC Lite). In what follows, the formulas under consideration are in non-clausal form. The logic of these formulas includes a Boolean *if-then-else* operator, defined as `if` a `then` b `else` $c \equiv (a \rightarrow b) \wedge (\neg a \rightarrow c)$, and a similar operator for terms. A formula containing the term `if` a `then` t_1 `else` t_2 can be translated to an equisatisfiable formula by replacing the if-then-else with a fresh variable v and conjoining the side conditions $a \rightarrow v = t_1$ and $\neg a \rightarrow v = t_2$. While these operators add no expressive power to the logic, they are very useful in applications.

The depth-first-search (DFS) heuristic chooses as the splitter the top-most, left-most atomic subformula within the formula being checked. The intuition behind this heuristic is that in the best case, when the top-level expression of the formula is an if-then-else with a literal as its condition and the consequent and alternate are of equal size and share no literals, then the heuristic splits on the condition and divides the problem into two sub-problems which are half the size of the original. Of course, in the general case, these criteria are not all satisfied and the sub-problems can be almost as large as the original problem. A refinement is to search the sub-trees of an expression in order of their height in the hope of splitting a larger sub-tree and yielding smaller sub-problems.

The caching heuristic identifies splitters that are *effective* and caches them for use in similar sub-problems. Given a partial assignment Δ, a splitter α is effective if it terminates the recursive case-splitting of `checkSat`; that is, if both `checkSat(`$\Delta \cup \{\alpha = \top\}$`,` ϕ'`)` and `checkSat(`$\Delta \cup \{\alpha = \bot\}$`,` ϕ'`)` reach a conflict without any further case splits. When the heuristic finds an effective splitter

it adds the splitters in the current partial assignment to an LRU cache, and when choosing a splitter it favors those that are in the cache. The intuition here is that we will encounter similar sub-problems for which the splitters in the cache are also effective.

However, we may encounter a sub-problem that contains a splitter that is in the cache, but is not effective for the sub-problem; the sub-problem is not closely related to the sub-problem for which the splitter was originally effective. In particular, when a splitter is added to the cache because it is effective for a small sub-problem, it is unlikely that it will be effective for a much larger sub-problem. Moreover, a poor splitter choice in a large sub-problem is worse, in terms of the amount of extra work it causes, than in a small sub-problem.

To mitigate these effects, the caching heuristic maintains a "trust" metric for each splitter in the cache. A splitter starts out with an initial trust, and each time it is found to be effective its trust is increased. If it is evicted from the cache it loses the trust it has earned. When choosing a splitter for a sub-problem of height h (where the height of a sub-problem is the height of the formula's parse tree), only those splitters in the cache with a trust of at least h are considered.

Finally, the splitters in the cache are ordered according to how recently they were added to the cache. So for a particular sub-problem, the newest splitter that is in the sub-problem and has sufficient trust for the height of the sub-problem is chosen. If no such splitter exists in the cache we fall back to the DFS heuristic.

The caching heuristic is similar in some respects to the heuristics in Chaff (both VSIDS and the heuristic that chooses a splitter from the most recent conflict clause if possible), insofar as they both try to take advantage of the adjacency of similar sub-problems as the decision tree is searched. The differences are that the caching heuristic is somewhat more conservative (it puts splitters in the cache only when it finds an effective splitter, not on every conflict), it falls back to the DFS heuristic when there are no applicable splitters in the cache, it makes no attempt to weed out splitters that do not contribute to the two conflicts of an effective splitter (corresponding to conflict-clause minimization), and it maintains a trust metric to avoid poor splitter choices. While the caching heuristic is somewhat *ad hoc*, it works well in practice.

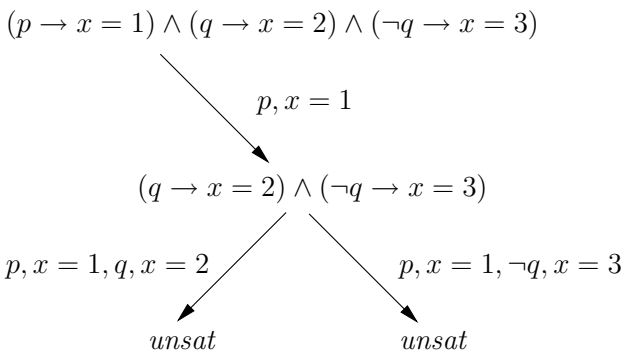

Fig. 2. q is an effective splitter

3 Combining Non-Clausal Heuristics with SAT Methods

The SVC decision procedure uses DPLL without conflict clauses or BCP, and works directly on non-clausal formulas. CVC has a mode where it converts the formula to CNF and calls Chaff for the propositional search; it annotates the CNF variables in order to reconstruct the non-clausal structure of the formula and use the DFS heuristic. CVC Lite implements its own Chaff-style Boolean search with BCP and conflict clauses. It stores the clausal part of the formula (i.e. any part of the formula already in CNF, and any conflict clauses generated during the search) separately from the non-clausal part, and does BCP on the clausal part only. The default strategy for CVC Lite is to use the simple DFS heuristic on the non-clausal part of the formula, and fall back to Chaff's VSIDS heuristic when there are no splitters left in the non-clausal part.

In the present work we add the caching heuristic to CVC Lite. The main difference between our implementation and the implementation in SVC is how effective splitters are determined. In SVC, if there are two conflicts in a row at the same decision level, that means that both polarities for the most recent splitter have been tried without needing to split again. With BCP and conflict clauses, the first conflict causes a conflict clause to be added, and subsequent BCP causes the most recent splitter to be asserted in the opposite polarity (this is known as a *failure-driven assertion*). If there is a second conflict before choosing another splitter, then the formula is unsatisfiable under the current partial assignment with the splitter in either assignment, which corresponds to our definition of an effective splitter.

Consider Figure 2, which shows part of a decision tree for a simple formula. First p is chosen as a splitter, which implies $x = 1$. Next q is chosen as a

$$(a \wedge b) \vee (c \wedge d) \equiv$$
$$(e_1 \vee e_2) \wedge (e_1 \vee \neg a \vee \neg b) \wedge (\neg e_1 \vee a) \wedge (\neg e_1 \vee b) \wedge (e_2 \vee \neg c \vee \neg d) \wedge (\neg e_2 \vee c) \wedge (\neg e_2 \vee d)$$

Fig. 3. Definitional CNF

splitter, which implies $x = 2$. The first-order decision procedure detects that this combination of literals is unsatisfiable.

When the basic DPLL search detects the contradiction it backtracks one level in the tree and asserts $\neg q$, which implies $x = 3$. Again the first-order decision procedure detects that this is unsatisfiable. Since asserting q in either polarity results in a contradiction, q is an effective splitter.

When DPLL augmented with BCP and conflict clauses detects the contradiction, it backtracks one level in the tree and adds the conflict clause $(\neg p \vee \neg q)$. Next it does BCP, and because p is asserted it deduces $\neg q$ from the newly-added conflict clause, which implies $x = 3$, causing another contradiction. It then backtracks again and adds the conflict clause $\neg p$ (since p is the only assumption). Since two conflicts have occurred in a row without an intervening decision, q is an effective splitter. Because in the core DPLL algorithm, the current Δ is lost when the algorithm backtracks, our implementation of the caching heuristic saves Δ as soon as it reaches a conflict, in case it turns out that the last splitter is effective.

3.1 Non-Clausal Boolean Constraint Propagation

If we translate a formula to CNF, we may be able to make valuable deductions by BCP that we would not be able to make on the formula in non-clausal form (since BCP as described works only on clauses). But the non-clausal heuristics depend on the structure of the formula, which is lost in translation to CNF. We can get the benefits of both approaches by keeping both the non-clausal formula and its CNF translation, and using the first for non-clausal heuristics and the second for BCP. (We convert to an equisatisfiable formula in definitional CNF, introducing a fresh variable for each non-atomic sub-expression to avoid the potentially exponential blowup in the size of the result.)

Consider Figure 3, which shows the translation of a simple formula to definitional CNF. Fresh variables e_1 and e_2 are introduced to represent $(a \wedge b)$ and $(c \wedge d)$, respectively, and the formulas $e_1 \leftrightarrow (a \wedge b)$ and $e_2 \leftrightarrow (c \wedge d)$ are translated into 3 clauses each. Now if we make a decision $\neg a$ then BCP will deduce $\neg e_1$ by the third clause, e_2 by the first clause, c by the sixth clause, and d by the seventh clause.

Following [9] we tried implementing propagation directly on Boolean connectives rather than doing BCP on the CNF translation. For example, if the

result of an AND expression is known to be true, then both of its child expressions must be true. This direct propagation can be done more efficiently than BCP on the CNF translation. However, in our present implementation the cost of the first-order decision procedure is much greater than that of BCP, so this optimization does not significantly improve the overall results.

4 Results

Figure 4 gives empirical results for the various methods we have discussed on a number of benchmarks from verification efforts. We treat the "simple" DPLL search with the DFS splitter heuristic as a baseline, and compare it to the "fast" search (incorporating BCP and conflict clauses), the fast search with additional clauses (generated by CNF conversion of the original formula), the simple search with the caching heuristic, and the combination of the caching heuristic with the fast search and additional CNF clauses.

The table shows the number of splits, the number of splits normalized to the simple search with DFS, the time in seconds, and the normalized time in seconds for each benchmark and method. Smaller numbers in the normalized fields in the table mean that the method does better than simple search with DFS. At the bottom we show the geometric mean of the normalized numbers to provide an overall comparison.

With simple search, the caching heuristic improves on DFS in both number of decisions and time. The fast search with DFS improves on the simple search in number of decisions, and the addition of CNF clauses to the fast search improves further on the number of decisions. Finally, the combination of the fast search with CNF clauses and the caching heuristic does better than either method alone in number of decisions, but is somewhat slower than the caching heuristic alone.

Notice that in general, the current implementation of the "fast" search achieves fewer splits, but requires more time. The implementation of the "fast" search is not the subject of this paper, but in future work, we expect to be able to optimize its performance significantly. One primary reason for our optimism is that our previous system, CVC, whose implementation is similar to that of the "fast" engine, except that the Boolean part does not produce proofs, is able to do many more splits per second than the "fast" engine of CVC Lite. We do not believe that proof-production accounts for all of the performance difference. This work shows that we can vastly decrease the number of splits by combining non-clausal and SAT-based heuristics. With further work we should be able to achieve a similar improvement in performance.

	Simple DFS				Fast DFS				Fast DFS + CNF			
	Splits		Seconds		Splits		Seconds		Splits		Seconds	
a	156	1	0.15	1	1005	6.44	0.85	5.66	930	5.96	0.69	4.63
b	56469	1	30.84	1	43581	0.77	47.86	1.55	38608	0.68	132.82	4.31
c	5534	1	2.78	1	4241	0.77	4.17	1.50	529	0.10	3.36	1.21
d	159	1	0.09	1	184	1.16	0.18	1.96	28	0.18	0.15	1.55
e	23674	1	8.49	1	155	0.01	0.35	0.04	186	0.01	0.58	0.07
f	703	1	0.10	1	703	1.00	0.29	2.89	26	0.04	0.41	4.12
g	4895	1	1.51	1	3114	0.64	3.81	2.51	40951	8.37	440.19	290.56
h	282	1	0.27	1	279	0.99	0.64	2.40	1976	7.01	15.69	59.20
i	1533	1	0.50	1	1187	0.77	1.57	3.13	21445	13.99	182.80	363.41
j	17484	1	5.01	1	13323	0.76	11.48	2.29	813	0.05	2.53	0.50
k	21294	1	6.58	1	20621	0.97	19.55	2.97	8	< 0.01	0.14	0.02
l	73484	1	21.40	1	54713	0.74	53.92	2.52	2902	0.04	9.56	0.45
m	25156	1	5.64	1	23906	0.95	14.54	2.58	10781	0.43	12.89	2.29
n	154238	1	22.18	1	412	< 0.01	0.17	0.01	407	< 0.01	0.23	0.01
o	134815	1	55.74	1	95910	0.71	221.14	3.97	610	< 0.01	5.18	0.09
p	121200	1	48.69	1	82295	0.68	174.64	3.59	696	0.01	6.62	0.14
q	3547	1	1.21	1	3547	1.00	6.34	5.23	3506	0.99	16.30	13.44
r	595	1	0.19	1	595	1.00	0.71	3.73	7	0.01	0.11	0.60
s	1863	1	0.75	1	2543	1.37	1.69	2.26	2418	1.30	2.36	3.16
t	314	1	0.30	1	301	0.96	0.45	1.50	277	0.88	1.19	3.99
u	331101	1	107.31	1	168984	0.51	126.03	1.17	1133	< 0.01	0.96	0.01
v	1282	1	0.56	1	1112	0.87	0.56	1.00	98	0.08	0.20	0.36
w	668	1	0.21	1	4	0.01	0.02	0.08	3	< 0.01	0.04	0.19
Mean	1.00		1.00		0.47		1.38		0.10		1.08	

	Simple DFS				Simple caching				Fast caching + CNF			
	Splits		Seconds		Splits		Seconds		Splits		Seconds	
a	156	1	0.15	1	179	1.15	0.21	1.43	548	3.51	0.46	3.05
b	56469	1	30.84	1	19486	0.35	6.85	0.22	399	0.01	0.69	0.02
c	5534	1	2.78	1	2350	0.42	1.00	0.36	557	0.10	1.17	0.42
d	159	1	0.09	1	118	0.74	0.10	1.03	21	0.13	0.13	1.37
e	23674	1	8.49	1	6512	0.28	3.24	0.38	124	0.01	0.37	0.04
f	703	1	0.10	1	703	1.00	0.11	1.09	26	0.04	0.22	2.24
g	4895	1	1.51	1	936	0.19	0.83	0.55	1917	0.39	27.27	18.00
h	282	1	0.27	1	443	1.57	0.58	2.18	737	2.61	13.63	51.44
i	1533	1	0.50	1	1533	1.00	1.46	2.91	3795	2.48	60.18	119.65
j	17484	1	5.01	1	22010	1.26	6.21	1.24	784	0.04	2.54	0.51
k	21294	1	6.58	1	490	0.02	0.19	0.03	8	< 0.01	0.14	0.02
l	73484	1	21.40	1	90861	1.24	26.30	1.23	786	0.01	2.53	0.12
m	25156	1	5.64	1	29943	1.19	6.58	1.17	389	0.02	0.56	0.10
n	154238	1	22.18	1	130702	0.85	21.60	0.97	407	< 0.01	0.23	0.01
o	134815	1	55.74	1	100208	0.74	86.57	1.55	576	< 0.01	11.30	0.20
p	121200	1	48.69	1	87814	0.72	41.03	0.84	662	0.01	6.74	0.14
q	3547	1	1.21	1	3547	1.00	1.51	1.24	3464	0.98	29.38	24.22
r	595	1	0.19	1	180	0.30	0.08	0.41	7	0.01	0.11	0.56
s	1863	1	0.75	1	594	0.32	0.37	0.49	876	0.47	1.21	1.61
t	314	1	0.30	1	43	0.14	0.06	0.21	37	0.12	0.27	0.89
u	331101	1	107.31	1	1210	< 0.01	0.39	< 0.01	1821	0.01	3.88	0.04
v	1282	1	0.56	1	315	0.25	0.12	0.22	95	0.07	0.28	0.49
w	668	1	0.21	1	408	0.61	0.20	0.96	3	< 0.01	0.04	0.20
Mean	1.00		1.00		0.41		0.53		0.04		0.55	

Fig. 4. Results

References

[1] Clark Barrett and Sergey Berezin. A proof-producing boolean search engine. In *CADE-19 Workshop: Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, July 2003. Miami, Florida, USA.

[2] Clark Barrett and Sergey Berezin. CVC-Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV)*, April 2004. To appear.

[3] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

[4] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, 2002. Copenhagen, Denmark.

[5] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

[6] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[7] Leonardo de Moura, Harald Ruess, and Maria Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *18th International Conference on Automated Deduction*, 2002.

[8] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James Saxe. Theorem Proving using Lazy Proof Explication. In *15th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[9] Malay K. Ganai, Pranav Ashar, Aarti Gupta, Lintao Zhang, and Sharad Malik. Combining strengths of circuit-based and cnf-based algorithms for a high-performance sat solver. In *Proceedings of the 39th Conference on Design Automation*, pages 747–750. ACM Press, 2002.

[10] Jeremy R. Levitt. *Formal Verification Techniques for Digital Systems.* PhD thesis, Stanford University, December 1998.

[11] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[12] M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

[13] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.