Integration of Supercubing and Learning in a SAT Solver

Domagoj Babić and Alan J. Hu

Department of Computer Science
University of British Columbia
{babic,ajh}@cs.ubc.ca

Abstract— Learning is an essential pruning technique in modern SAT solvers, but it exploits a relatively small amount of information that can be deduced from the conflicts. Recently a new pruning technique called supercubing was proposed [1]. Supercubing can exploit functional symmetries that are abundant in industrial SAT instances. We point out the significant difficulties of integrating supercubing with learning and propose solutions. Our experimental solver is the first supercubing-based solver with performance comparable to leading edge solvers.

I. Introduction

The problem of satisfiability of boolean formulas (SAT) is a well-known NP-complete problem. In short, given a boolean function f, one needs either to find a satisfying assignment or to prove that such doesn't exist. SAT has been intensively used in many domains. Our focus is the application of SAT to EDA problems, like FPGA routing [2], ATPG [3], processor verification [4], and especially bounded model checking [5]. Such industrial applications require complete SAT solvers, meaning that the solver must be capable of proving that problem is either satisfiable or definitely unsatisfiable.

A. Previous work

Zhang et al. [6] have analyzed different learning schemes, and found that first Unique Implication Point (1-UIP) learning is superior to other schemes. In general, a common belief is that the shorter the learned clause is, the more information it contains, and therefore it's supposed to be more useful. Ryan [7] has correctly pointed out that 1-UIP clauses are often longer than all-UIP clauses, which are not that efficient in search space pruning. He conjectured that 1-UIP clauses are working that well because fewer resolutions are needed for their generation.

This work was supported in part by a research grant from the Natural Science and Engineering Research Council of Canada and a graduate fellowship from the University of British Columbia.

Although much work remains to be done on different learning schemes, it is clear that learning schemes proposed so far can use only a fraction of information inferrable from the conflicts. Due to memory constraints, it is impossible to add all the clauses that can be learned to the clause database. Recently, a theory of essential points [1] has been proposed. The theory unifies many existing search space pruning schemes (like pure literal rule, Conflict Directed Backtracking (CDB) [8], and learning) under a single framework and serves as a tool for developing new pruning techniques. A new pruning technique called supercubing was proposed as an example of application of the theory of essential points. Their solver was a proof of concept, and although supercubing reduced the number of decisions, no actual speedup has been reported.

Reducing the number of decisions, although certainly desirable, doesn't necessarily mean that the total runtime has been decreased. First, the new technique might be too expensive to be performed after every conflict. And second, it might be incompatible with existing efficient pruning schemes. In the case of supercubing, it turns out that it is a bit more expensive than previously mentioned schemes, but the major issue is actually the incompatibility with learning.

Supercubing reduces the number of decisions during the SAT solving process by exploiting inherent functional symmetries in the problem. When SAT symmetries are mentioned, a reader familiar with SAT solving will first think of breaking structural symmetries by introducing additional clauses as presented in [9, 10]. Introducing new symmetry-breaking predicates prevents the solver from exploring multiple symmetrical regions of the search space. If the instance is unsatisfiable, it will remain so, and if it is satisfiable, symmetry-breaking will reduce the number of solutions. A new tool by the same authors, Saucy [11], is very efficient in breaking structural symmetries. It is used as a preprocessing step that can result in a speedup of several orders of magnitude on highly symmetrical instances. But there is a class of functional symmetries that symmetry-breaking tools based on graph isomorphism can't detect.

On the other hand, supercubing can detect functional

symmetries, and guide the solver to stay within a cluster of related variables, thus effectively localizing the search. Hence, we see supercubing as complementary to the previous research on SAT symmetries.

A solver which could take advantage of both supercubing and traditional learning would be of great interest to the EDA community. But there are significant obstacles in integrating supercubing with existing techniques. After explaining the integration issues from a new perspective, we offer innovative solutions and present experimental results. Our work lays down the foundations for using far more advanced techniques than supercubing and encourages further research.

II. Supercubing

This section explains the theory and implementation of supercubing [1, 12]. We discuss the essential difference between supercubing and structural symmetries, providing the reader with deeper insight into the problem. Finally, we propose an algorithm for efficient computation.

A. Essential points

The theory of essential points is a formalization of the symmetries present in the search space. It is a basic formalism behind the supercubing technique. Here we give shortened versions of the most important definitions and a theorem from [1, 12].

Definition II.1 *Let ψ be a partial or complete assignment of a set of variables in some CNF formula f, such that ψ doesn't satisfy f. ψ can be seen as a point in a multidimensional space. Also, let $\mathcal{C}$ denote a set of all unsatisfied clauses of f by assignment ψ. Then, point ψ is called* ***l-essential*** *if $\forall c \in \mathcal{C},\ l \in L(c)$, where l is some literal and $L()$ represents a set of all literals in some clause.*

For example, given $f = (-2 \vee 1 \vee 4)(-2 \vee 1 \vee -4)(4 \vee -3 \vee 2)(1 \vee 4 \vee -3)$, and $\psi = \{-1, 2, 3, -4\}$, there are two unsatisfied clauses $(-2 \vee 1 \vee 4)$ and $(1 \vee 4 \vee -3)$, hence point ψ is 1- and 4-essential.

Definition II.2 *Let ψ and ψ^* be two assignments in 2^n Boolean space. Those two are said to be* ***l-symmetric*** *if one is obtained from the other by flipping literal l.*

Theorem II.1 *If a DPLL algorithm has explored one branch of some decision variable x finding no solutions, then if there are any solutions under the second branch, they must lie in the set of points that are x-symmetric to the set of essential points under the first branch.*

The proof is given in [12].

B. Supercubing algorithm

For every decision variable in the current search space, the algorithm maintains an array that we will call a supercube. The supercube is initially empty when the variable is first chosen for branching.

After every conflict, the algorithm resolves the conflict, following implication edges in the implication graph backwards until the resolvent consists only of decision variables. If we flip the sign of all literals in such a resolvent, we get a decision conflict clause. Adding a decision conflict clause to the clause database is known as a decision learning scheme, and it is usually inferior to 1-UIP

Definition II.3 *Let $\mathcal{S}$ be a supercubing operator over two resolvents, say s_1 and s_2, which computes their minimum cube. $\mathcal{S}(s_1, s_2)$ is called the supercube of s_1 and s_2. If some variable doesn't have a supercube at all, we say it has an empty supercube.*

For example, given two resolvents $s_1 = \{-1, 2, 3, -4\}$ and $s_2 = \{1, 2, 3, 5\}$, their supercube would be $\mathcal{S}(s_1, s_2) = \{2, 3\}$.

Let's assume that a resolvent contains all the decision variables involved in the conflict. For all those variables, the algorithm computes the supercube of their current supercubes and the newest resolvent. If some decision variable has an empty supercube, meaning that it has been involved in some conflict for the first time, we can initialize its supercube to the current resolvent.

Finally, when the algorithm backtracks to some decision variable x, of which only one branch has been explored so far, it examines its supercube. Based on the supercube, the algorithm can make two decisions:

- If x has a supercube, that means that literals present in the supercube were essential in all the resolvents under the first branch of x. Hence the solver can immediately assign them appropriate values, without exploring their second branch, as it is definitely void of solutions. For example if the supercube of x contains literals $\{1, -3\}$, than after flipping x, the algorithm can immediately assign 1 to true and 3 to false. If the supercube doesn't contain any elements, the solver proceeds by standard means (making a new decision or solving implied variables).

- An empty supercube means that x wasn't involved in any conflicts under its first branch. Therefore, x was not responsible for the conflict, so flipping x will not lead to the solution either. Thus, the algorithm can safely backtrack over x, by the same principles as in CDB. It is worth mentioning that CDB can also be formalized through the theory of essential points.

Two questions come naturally. First, isn't it possible to prune the same search space through some kind of learning? And second, how much can we gain by supercubing?

The first question is answered in [1] by a small example that clearly demonstrates that there are cases where supercubing can prune some part of the search space, while learning can't. We offer the answer to the second question in Section IV.

C. What are we actually pruning ?

Supercubing doesn't prune structural symmetries, as Saucy and Shatter [9] tools do. Rather, it detects a set of essential points lying under the first branch of some decision variable x, and prunes its counterpart under the second branch of x. Here we give a small example, which not only serves the purpose of illustrating the supercubing technique, but also demonstrates the significant differences between the mechanisms of traditional CDB and supercubing. The goal is to nudge the reader to think about the integration — the main topic of the paper.

Let's consider a formula $f = (1 \vee -2 \vee 4)(1 \vee -2 \vee -4)(1 \vee 2 \vee 4)(1 \vee 2 \vee -4)(-1 \vee -2 \vee -4)(-1 \vee 3 \vee -4)(-1 \vee -3 \vee -4)(2 \vee -3 \vee 4)(2 \vee 3 \vee 4)$, and the search tree in Fig.1. Decision nodes are represented by white squares. The variable is in the central box, and the assigned phase of the literal is to the left and right of the variable. For example, first we branch on variable 1, and assign it false (0), following the edge to variable 2, which is a new decision. Conflicts are marked as **X**, while implied variable 4 is represented as a gray box with two compartments, the right contains the variable and left its value. Supercubes can be associated only with decision variables. If there are one or more variables in the supercube, they will be represented in brackets above the associated decision variable.

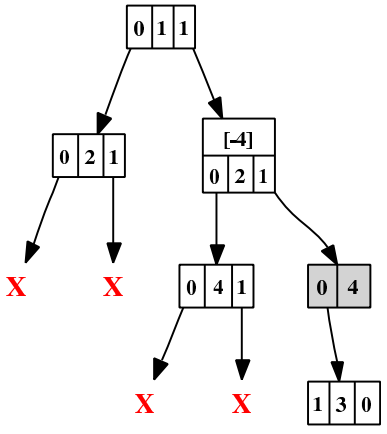

Fig. 1. Supercubing search process

Let's consider the more interesting right branch of the search tree. The resolvents of the first two conflicts under the right branch are $\{-2, -4\}, \{4, 1\}$. From those two, the the algorithm can infer that after flipping decision variable 2, it can immediately assign false to variable 4, satisfying the formula.

The dynamics of solving the same problem is significantly different for classical solvers. After assignment $\{1, -2, -4\}$, classical learning would construct learned clause $(2 \vee 4)$, and backtrack to decision level two, without flipping variable 2. At that point, the new learned clause would be a unit clause, implying assignment 4. Solving 4 as an implied literal would result in a new conflict clause $(-4 \vee -1)$ that would force the solver to backtrack to decision level one, and then solve 4 as an implied unit literal. At that point the first learned clause will imply 2 and the formula is satisfied.

A learned clause that causes a flip of the variable is called an asserting clause. It is important to notice that decision variables are never directly flipped, but rather forced to the opposite value by the asserting learned clause after backtracking.

Pruning by supercubing has one significant shortcoming. It can be applied only in the second branch of some decision variable, while learned clauses can be useful long after the solver leaves the current search space. In spite of this limitation, supercubing effectively reduces the number of decisions. In addition, the decision variable and the variables in the associated supercube are most likely in the same cluster of related variables. Thus, immediately after flipping a decision variable, the solver will continue solving variables in the same cluster, localizing the search and increasing the probability of getting more unit literals related to the most recent decision variable.

Seemingly, the larger the supercubes are, the more we can gain by the technique. This is only partially correct. Interaction between learning, decision heuristics and supercubing is extremely complex. Heuristics that tend to increase the priority of implied variables [7] result in a different sequence of decisions in two adjacent branches of decision variables, reducing the size of constructed supercubes. Increasing the priority of decision variables in resolvents yields much longer supercubes, but doesn't work that well with learning. Hence, it is important to find a balance. The heuristic that we are currently using is similar to BerkMin's heuristic [13], with the difference that we scale the priorities rarely and only by a factor of two.

Every decision variable that has participated in at least one conflict will have a supercube. If we compute a cumulative size of all supercubes and divide the sum by the number of decision variables of which we have explored both branches (flipped variables), we get a density of supercubes. This factor tells us the average size of supercubes that we can expect. As suggested before, the factor depends on the heuristics and learning used in combination with supercubing.

The benchmarks used in Table I are mostly well-known industrial SAT instances. Instance rand3 is a random 3-SAT benchmark with 200 variables, and SAT_dat.k20 is from the IBM formal verification benchmark suite[1]. Cho-

[1]Available at http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html

TABLE I
SUPERCUBING STATISTICS

Benchmark	# dec.SC	# flips.SC	density	dec.NSC	flips.NSC
barrel6	7805	6928	2.817	11759	8711
longmult7	14127	13845	0.027	14147	13862
rand3	123251	105467	0.200	127347	108881
fpga10_11	872061	761401	0.047	1038014	896738
SAT_dat.k20	7958	4582	0.203	8173	4360

sen instances are representable for their class as far as supercube densities are considered. Columns marked with SC denote runs of our experimental solver during which the supercubing was turned on. On all instances, except for SAT_k20[2], supercubing reduces the number of decisions and flips needed to solve the problem.

Densities are rather small on average, but every used variable from a supercube cuts the search subspace under the associated decision variable in half. Hence, even a small number of variables in supercubes can result in significant speedups.

Table II presents supercubing statistics after the problems were preprocessed with Shatter and Saucy. The densities remain roughly the same. The reduction of the number of decisions and flips achieved by supercubing is consistent too. We believe that the increase of the density for FPGA benchmark is due to better decisions as a result of additional implicates introduced by symmetry-breaking tools.

TABLE II
SUPERCUBING STATISTICS AFTER PREPROCESSING WITH SAUCY

Benchmark	# dec.SC	# flips.SC	density	dec.NSC	flips.NSC
barrel6	1540	1265	2.673	2386	1673
longmult7	18557	18203	0.040	18473	17591
rand3	123251	105467	0.200	127347	108881
fpga10_11	21689	16740	0.225	25624	19947
SAT_dat.k20	7687	4196	0.176	6742	3903

As mentioned previously, the relation between decision heuristics, learning, and supercubing is rather tangled. Many different decision heuristics have been proposed so far, e.g. [13, 7, 14]. Many of them are based on increasing the priorities of the variables involved in conflicts. Most modern solvers resolve the conflict until the first UIP is reached, and then increase the priorities of all or some number of variables present in the resolvent. Such resolvents tend to contain more implied then decision variables[3]. Increasing the priorities of implied variables means that those variables will be more likely candidates for new decisions after we flip some decision variable. The immediate consequence is that supercubes will be shorter on average, as resolvents will contain a more dispersed set of variables. Heuristics that boost the priorities of decision variables result in larger supercubes, but after all the variables from some supercube have been assigned, the solver has to make a new decision. And for highly dynamic problems it is usually preferable to select the variables that were recently involved in conflicts (most often those were implied variables).

Benchmark SAT_dat.k20 in Table I is an example of this phenomenon. The preference towards variables which were involved in recent conflicts is so strong, that even starting a new branch with supercubed variables results in slightly larger number of decisions. According to expectations, changing the heuristics to increase only the priorities of variables used for constructing UIP conflict clause, decreases the density of supercubes, but results in a smaller number of decisions, as shown in Table III.

TABLE III
SUPERCUBING STATISTICS WITH A HEURISTIC THAT FAVORS IMPLIED VARIABLES

Benchmark	# dec.SC	# flips.SC	density	dec.NSC	flips.NSC
SAT_dat.k20	4296	2989	0.110	5776	4225
SAT_dat.k20 +Saucy	3997	2584	0.121	5755	3805

D. Computation

The resolvents used for supercubing contain only decision variables. In most problems, the percentage of decision variables in assignment ψ is relatively small. In addition, the algorithm can skip over flipped decision variables, because their supercubes will not be used later. For each remaining variable v, its new supercube is computed by applying $\mathcal{S}$ operator to its current supercube and an array containing all the variables from R with higher decision level than the decision level of v.

The algorithm is simple and efficient. If resolvent R and $v.scube$ are sorted arrays of literals, then $\mathcal{S}$ operator can be computed in linear time.

```
for all v = var(l) | l ∈ R do
  if !isBranchable(v) then
    continue
  else
    tmpa ← ∅
    for all x ∈ R do
      if dlev(var(x)) > dlev(v) then
        tmpa ← tmpa ∪ x
      end if
    end for
    v.scube ← S(v.scube, tmpa)
```

[2]Explanation of this exception will be given later.

[3]We have checked this statement empirically, but we omit the results as those are not crucial for further discussion.

end if
end for

In the given pseudocode listing function *var*() returns a variable for a given literal. *dlev*() function returns the decision level of a given variable. Function *isBranchable*() returns TRUE for decision variables for which only one branch has been explored so far.

In our experimental solver construction of the resolvent and supercubing take a negligible amount of time (typically under 1%).

III. Integration with learning

A. Backtracking

Traditional CDB is tightly interwoven with learning. The 1-UIP [6] learning scheme adds only one literal (UIP) from the last decision level to the conflict. That literal is a dominator of the conflicting nodes in the implication graph and most often an implied variable. The solver backtracks to the penultimate decision level of all the literals in the conflict. At that point, the new learned clause will imply the opposite of the previous assignment of the UIP variable, indirectly flipping it.

Now, let's try to add supercubing to such a solver. After each conflict, the algorithm constructs a resolvent consisting only of decision variables. Such a resolvent is than used, as previously described, for computation of supercubes. Thus, the computation of the supercubes remains the same.

UIP variables are the only variables that are actually being flipped through learned assertion clauses. If the flipped variable was an implied variable prior to backtracking, it can't have a supercube, hence no additional pruning can be achieved. In the case when it was a decision variable, it had the highest decision level when the conflict was discovered and therefore the resolvent R of that conflict will not include any decision variables with a higher decision level[4]. A new supercube of the flipped decision variable x is $\mathcal{S}(x.scube, tmpa)$. As the flipped variable had the highest decision level, *tmpa* will be empty, and the solver will effectively delete all the variables currently existing in the supercube. So, the flipped variable has a supercube, but no elements in it.

Obviously, although computation of the supercubes remains the same, supercubing can't be exploited in the same way. Note that the last resolvent must be taken into account, as otherwise the supercube of x wouldn't represent the set of essential points in all the conflicts under the first branch of x.

The only way we found to integrate supercubing and UIP learning is to backtrack to the last branchable decision variable with a supercube, flip it and continue the computation with supercubed and implied variables.

[4]See the previously given algorithm.

B. Assertion clauses

Let's denote the decision level to which the supercubing based solver backtracks as *sc_bdl*, and the corresponding level for CDB solver as *cdb_bdl*. In general, those two levels are different. In addition, the mechanics of backtracking are different. A supercubing based solver will unassign all the literals having a decision level that is higher or the same as *sc_bdl*. Conversely, a CDB solver will only unassign literals whose decision level is higher than *cdb_bdl* and continue the computation at decision level *cdb_bdl*, at which UIP literal from the last learned clause will become a unit literal.

If *cdb_bdl* > *sc_bdl*, backtracking to the last branchable decision variable will either satisfy the assertion clause or leave at least two unassigned literals. A more interesting case is when *cdb_bdl* $\leq$ *sc_bdl*. There can be multiple branchable decision variables between those two levels and we will call them intermediate variables. Flipping any of them will not immediately satisfy the learned clause because the UIP literal is implied by assignments at higher decision levels, which do not change after backtracking to any of the intermediate variables.

The solution we propose, is to schedule the computation of the implied UIP literal after every intermediate variable. The scheduled implied variables can be assigned immediately after flipping the associated decision variable. Altogether, the order of computation is:

1. flipped decision variable
2. scheduled implied variables
3. all variables implied by either the last decision or scheduled implied variables
4. supercubed variables.

Typically, the number of levels that get unassigned after backtracking is small and the cases when there are many intermediate variables are rare. In industrial benchmarks the average number of scheduled implied variables is usually between one and two per decision node.

C. Implication graph

In this section we explain the order of computation in more detail. As pointed out in [1], supercubed variables can be assigned immediately after flipping the associated decision variable, but the order of evaluation of supercubed and implied variables is arbitrary in general. Hence, it seems natural to consider supercubed variables as implied, despite the fact that they have no antecedent clause.

On the other hand, integration with learning seems to demand that supercubed variables are handled as non-branchable decision variables. Every supercubed variable is assigned at a new decision level after all the implied variables have been assigned. In the case when a

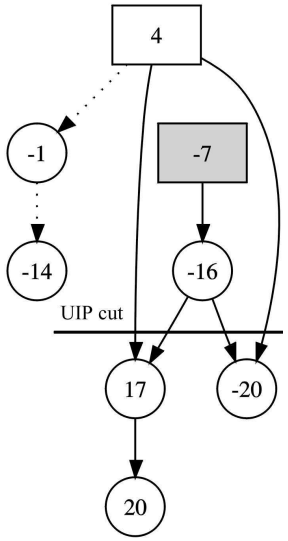

Fig. 2. Implication graph

conflict is found during Boolean Constraint Propagation (BCP), some supercubed variables might not be used at all. The percentage of the used supercubed variables depends heavily on the type of the problem and decision heuristics. Our solution is motivated by an example of the implication graph at Fig.2.

Decision variables are shown as white boxes, supercubed variables as gray boxes, and implied variables are represented by circles.

If supercubed and implied variables were computed at the same decision level, the implication graph could have two or more roots at the current decision level. It seems that finding UIPs in such a multi-rooted graph is undefined. For example, if all the variables in Fig.2 were assigned at the same decision level, the resolvent would be $\{4\}$ or $\{4, -7\}$, depending on whether the solver considers the supercubed variable to be a decision or an implication. The first resolvent is incorrect, while the second does not represent a UIP cut.

Assigning variables by the proposed order always results in correct construction of resolvents, as there is no difference between supercubed and non-branchable decision variables. Hence, in the given example the solver will first assign variables implied by the flipped decision variable (dotted implication edges). Afterwards, it will proceed with the supercubed variable that is evaluated at a new decision level. If there are more supercubed variables, the solver will assign all implied variables first, and if there is no conflict, assign the next unassigned supercubed variable.

IV. Experimental results

Our experimental solver is still in the early stage of development, so the results presented here should be taken as a proof of concept rather than as a definite evaluation of the capabilities of supercubing. All tests have been performed on 2.6GHz Pentium 4 with 2 Gb of memory. ZChaff version 2003.11.04 [15] was used for comparison. For evaluation, we use bounded model checking (BMC)[5] and FPGA routing[6] instances.

The timeout for BMC benchmarks was set to 300 seconds, while the timeout for FPGA instances was 3600 seconds. The number of instances on which the solver has timed out is given in parentheses.

TABLE IV
Experimental results

Benchmark set	Our solver	ZChaff	Speedup
BMC-barrel	332.4(1)	194.6	0.58
BMC-longmult	923.4	1280.2	1.38
BMC-queueinvar	6.8	3.0	0.44
FPGA-UNSAT	21855.8(4)	30387.1(8)	1.39
FPGA-SAT	4429.9	10903.1(2)	2.46

Our solver is on average a bit slower on BMC benchmarks, especially on the barrel9 instance that the solver couldn't solve in the given period of time. On FPGA routing instances, our solver performs significantly better. It is well-known that symmetry-breaking works very well on FPGA instances, so it seems that instances with a lot of structural symmetries can gain more from supercubing. Running Shatter and Saucy first decreased the runtimes of both solvers, but our solver was still faster on FPGA and longmult instances by approximately the same percentage as in Table IV.

V. Conclusion

The paper explores the integration of a new pruning technique called supercubing [1] with traditional techniques like conflict directed backtracking (CDB) and learning. After analyzing the mechanics of supercubing and CDB solvers, we have identified the main sources of incompatibility and proposed efficient solutions. Also, we suggested a simple and efficient algorithm for computation of supercubes.

Our findings are that heuristics implemented in leading edge solvers, like BerkMin [13] and ZChaff [15], don't work very well with supercubing. Heuristics that tend to increase the priority of decision variables result in larger

[5]Available at http://www-2.cs.cmu.edu/~modelcheck/bmc/BMC-dimacs-examples-0.0.tar.gz

[6]Available at http://www.eecs.umich.edu/~faloul/benchmarks.html

supercubes, but when the solver assigns all the supercubed variables, the decisions it makes become worse than if VSIDS [15] or BerkMin [13] heuristics are used. On the other hand, the mentioned heuristics generally cause a different order of decisions in adjacent branches of decision decision variables, effectively dispersing the resolvents and reducing the size of supercubes. It seems that good heuristics supercubing-based solvers should find a balance between increasing priorities of decision and implied variables, but more research is required before we can make strong conclusions.

The additional complexity of integration seems to be justified according to our preliminary results. We believe that improving our fragile heuristics, adding simple preprocessing, and overall solver optimization should yield even better results.

Our main contribution is the proposed solution for integration of supercubing in traditional SAT solving framework. We expect that more powerful pruning techniques than supercubing can be developed and that will be the focus of our future research.

References

[1] Evgueni Goldberg, Mukul R. Prasad, and Robert K. Brayton. Using Problem Symmetry in Search Based Satisfiability Algorithms. In *Proceedings of the conference on Design, Automation, and Test in Europe*, pages 134–142, 2002.

[2] G. Nam, K. Sakallah, and R. Rutenbar. A boolean satisfiability-based incremental rerouting approach with application to FPGAs. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 560–565. IEEE Press, 2001.

[3] P. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, Sept 1996.

[4] Miroslav N. Velev and Randal E. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. In *Proceedings of the 38th conference on Design automation*, pages 226–231. ACM Press, 2001.

[5] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 317–320. ACM Press, 1999.

[6] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer-aided Design*, pages 279–285. IEEE Press, 2001.

[7] Lawrence Ryan. Efficient algortihtms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[8] João P. Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, 1999.

[9] I. L. Markov F. A. Aloul, A. Ramani and K. A. Sakallah. Solving Difficult SAT Instances In The Presence of Symmetry. In *Proceedings of the Design Automation Conference*, pages 731–736. ACM/IEEE, June 2002.

[10] Karem A. Sakallah Fadi A. Aloul, Igor L. Markov. Shatter: Efficient Symmetry-Breaking for Boolean Satisfiability. In *Proceedings of the Design Automation Conference*, pages 836–839. ACM/IEEE, June 2003.

[11] Karem A. Sakallah Paul T. Darga, Mark H. Liffiton and Igor L. Markov. Exploiting Structure in Symmetry Detection for CNF. In *Proceedings of the Design Automation Conference*, pages 530–534. ACM/IEEE, 2004.

[12] Mukul Ranjan Prasad. *Propositional Satisfiability Algorithms in EDA Applications*. PhD thesis, University of California at Berkeley, 2001.

[13] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Proceedings of the conference on Design, Automation, and Test in Europe*, pages 142–149, 2002.

[14] João P. Marques Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, pages 62–74. Springer-Verlag, 1999.

[15] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535. ACM Press, 2001.