The Evolution from LIMMAT to NANOSAT

The Evolution from LIMMAT to NANOSAT

Technical Report #444

Dept. Computer Science, ETH Zürich, CH-8092 Zürich, Switzerland

Armin Biere
Computer System Institute

15. April 2004

Abstract

In this technical report we summarize the evolution of SAT solvers implemented at the formal methods group at ETH Zürich. We start with LIMMAT and COMPSAT, which both took part in the SAT'03 SAT solver competition. From the more ambitious design of FUNEX we reach the minimal implementation of NANOSAT. We close the discussion with an overview on how our QBF solver QUANTOR can be used as a preprocessor for SAT. We highlight differences to similar implementations, emphasizing new ideas. No detailed experimental comparison is provided, since the main purpose of this technical report is to give a reference point for the SAT'04 SAT solver competition.

1 Introduction

The most successful technique for SAT is the classical DPLL style algorithm [3]. The last decade has seen a tremendous increase in reasoning power of SAT solvers. The improvements are mainly based on the following techniques: *non chronological backtracking* as in GRASP [6], *learning of conflict clauses* as in RELSAT [1], GRASP and SATO [9], *unique implication points* pioneered by GRASP, *lazy data structures* following SATO [9] and CHAFF [7], and, most important, the new dynamic decision heuristics VSIDS of CHAFF [7] and its variances implemented in BERKMIN [5]. Our solvers follow this evolution of techniques and implement some more optimizations.

2 SAT Solver LIMMAT

The LIMMAT project was started in 2001 before the source code of the SAT solver CHAFF became available. Our main motivation was to cross check the impressive results the authors of CHAFF reported. From the publications alone, without access to the source code, various details were still unclear. As a consequence it turned out, that our first version of LIMMAT was hardly able to match the performance of CHAFF, after it became available as executable. Nowadays plenty of reimplementations of CHAFF are publicly accessible in source. These reimplementations more or less follow the design decisions made by CHAFF and its ancestor GRASP.

One such important design decision is concerned with how exactly clauses are learned. In our first version of LIMMAT we implemented the simplest scheme: traverse the implication graph starting from the conflict and collect the negation of all decisions encountered. As an optimization we terminated the traversal at *unique implication points*. The idea is to generate smaller conflict clauses, as pioneered in GRASP. In our experiments, this optimization turned out to be important, which was also the conclusion of the detailed analysis in [10]. We believe that the real power of this optimization lies in the fact, that less variables in the conflict clause potentially may result in much larger back jumps in non chronological backtracking.

However, what we did not realize, and which hardly could be deduced from the literature, was the optimization, employed in GRASP and CHAFF, to stop traversal of the implication graph, whenever a literal assigned at a lower

decision level than the current one is reached. The effectiveness of this optimization can be explained by the same intuition which is behind VSIDS and the decision heuristics of BERKMIN: localize search as much as possible.

After the source code of CHAFF became available we could trace the learned clauses in CHAFF and compare them to those learned in LIMMAT. Only then our unfortunate design decision became clear. In principle, a detailed analysis of the source code of GRASP could also have revealed the problem. However, it was not clear that CHAFF followed GRASP's design decision in this case. Also CHAFF was proven to be much faster than GRASP.

After we changed LIMMAT accordingly by adding another termination criteria to the implication graph traversal, the performance of LIMMAT increased considerably. The change only involved adding one single boolean expression to the C program. The lesson learned is, that important details are often omitted in publications and can only be extracted from source code. It can be argued, that making source code of SAT solvers available is as important to the advancement of the field as publications.

2.1 Fast Access

The original two-watched literal scheme of CHAFF has the following drawback. Assume, that a clause is satisfied and the first watched literal pointer is moved to a satisfied literal. It may often happen, that later this clause is visited again through the second watched literal. In our analysis we realized that the original implementation of CHAFF would still need to traverse the literals of the clause, until either the first satisfied literal, another satisfied literal or an unsatisfied literal is found. This traversal is unnecessary if another level of indirection is used.

In CHAFF and LIMMAT the occurrence lists of literals are both implemented as stacks of pointers. In CHAFF the pointers directly point to the watched literals in the clause, while in LIMMAT they point to the clause head. In the clause head the indices of the watched literals are saved. In order to distinguish the two watched literals, we use the low level bit of the pointers in the occurrence lists. This additional level of indirection, allows constant time lookup of the other watched literal in the same clause and avoids those superfluous traversals in the example discussed above. The disadvantage of this technique is the additional pointer dereference necessary to access a watched literal.

2.2 Optimizing Reasons

In LIMMAT the assignments of variables generated during boolean constraint propagation (BCP) are stored in the *assignment queue*, which is not a unique feature of LIMMAT. Using a queue instead of a stack results in breadth first BCP instead of depth first BCP. The former heuristically minimizes the depth of the implication graph, which may result in shorter conflict clauses.

In LIMMAT we made the design decision to postpone the actual assignment of variables until the corresponding assignment is dequeued from the assignment queue. This has the following effect. During clause traversal when moving watched literal pointers, literals that are already known to be assigned false, may still be used as new target of a watched literal pointer.

This *lazy assignment* of variables allows the following two optimization. They are both triggered, when an assignment for an unassigned variable is enqueued to the assignment queue the second time in the same BCP. There are two cases, depending on, whether the values in the two assignments match or not. If the values do not match, then an *early conflict* is detected and the current BCP terminates. However, if the two values match, then we have two *reasons*, e.g. clauses implying the assignment of the variable to the common value.

This opens up the possibility to drop either reason, which we call *reason simplification*. For the decision which reason to use, we employ the greedy heuristics to pick the smaller reason. This choice may decrease the depth of the implication graph, again with the goal to minimize the size of conflict clauses.

Our experiments showed that chances for simplification of reasons are rare, but early conflict detection can often be applied. Since early conflict detection is automatically detected in eager assignments of variables, it turns out that, there is not much benefit in using lazy assignments. Finally, note, that *reason simplification* can not immediately be combined with eager assignments of variables, since overwriting reasons without care may introduce cycles into the implication graph.

3 SAT Solver COMPSAT

LIMMAT was our first implementation of the ideas of CHAFF [7]. Although partially successful in the SAT'02 SAT Solver competition, where LIMMAT won the category of satisfiable industrial benchmarks, it was clear that the plain reasoning speed of LIMMAT was much slower than the one of CHAFF.

In a first attempt to analyze the reason for this effect, we tried to produce the same decision order within LIMMAT as in CHAFF. After quite some effort, we finally gave up. It became apparent, that visiting the clauses in the occurrence lists in different order or traversing literals of clauses differently, will result in different conflict clauses generated. This directly influences, which decisions are taken. In essence, it is impossible, to compare even slight variants of data structures, without enforcing the same decisions explicitly.

Since a direct comparison turned out to be impossible, we just guessed that the often observed slow down of LIMMAT compared to CHAFF was due to the slightly more bulky data structures of LIMMAT and the two level indirection when accessing a watched literal from the occurrence lists through a clause head. In order to show that this is one of the main problems with LIMMAT, we decided to build a new SAT solver COMPSAT with the main goal of a compact memory layout.

Just reimplementing CHAFF's data structures, would not have given any new insight and we also wanted to keep the successful feature of LIMMAT, that allows fast access to the other watched literal and avoids superfluous traversals of literals, if one of the watched literals satisfies the clause.

Around that time van Gelder published a new lazy data structure in [8], based on moving the watched literals to the first two positions of the array of literals of a clause. As in LIMMAT the occurrence lists contain pointers to the clauses instead and not to the literals as in CHAFF. Since the two watched literals are the first two literals in the clause, they can be accessed immediately and no additional indirect access is necessary as in LIMMAT. Therefore this was the base for our new implementation.

The resulting memory layout is very simple and compact. A clause is represented by the index of its first literal in the literal pool, which is just a large array of integers. The last literal of a clause is followed by a separator, the number 0, which does not correspond to any legal literal. Occurrence lists of literals are implemented as stacks of integer. They contain the indices of the clauses in which the literal is watched.

As described in [8] BCP involves visiting all clauses in which a literal is watched. Then the literals of such a clause are traversed, searching for another literal which is not assigned to false. This literal is swapped with the currently watched literal. Before the traversal of the literals of the clause is started, it is checked that the other watched literal, which can be accessed in constant time, does not already satisfy the clause. In this case no traversal is necessary.

Compared to CHAFF the lazy data structure of COMPSAT allows fast access to the other watched literal, and does not need bit-stuffing to mark watched literals. However, it always involves starting the traversal of the literals of a clause at the first position, as opposed to CHAFF, where the traversal starts at the watched literal. In theory, this may result in quadratic many more traversals, although we have not seen this to occur in practice. As a remedy to this problem, one may cache the last traversed position, similar to the cache for the direction of the last traversal in CHAFF. So far we have not seen it necessary to implement this optimization.

As we learned from LIMMAT, the idea of simplifying reasons does not pay off, and accordingly we implemented eager assignments to variables: as soon a unit is found the variable is immediately assigned, as opposed to LIMMAT, where the assignment is just enqueued to the assignment queue, and the assignment is only performed, when it is dequeued from the queue.

In COMPSAT, on a low level of implementation, mainly one dynamic data structure is used, which is a generic stack of integers. In order to obtain and even more compact memory layout, we implemented a *compact stack implementation* of integer stacks, which has the following characteristics.

Its anchor only needs two 32 bit words, which beside saving space also fits better to modern cache architectures. This is only possible by restricting the maximal number of elements on the stack to 2^{28}. The compact stack allocates an additional heap object only on demand. In addition, it uses consecutive 16 bit short words on the heap instead of 32 bit words, if the integers stored on the heap are small

enough. This is particular useful if the number of variables is smaller than 2^{15}, and, in this case, reduces the memory foot print by almost a factor of two.

At compile time the user can specify, whether the compact stack implementation is used or the standard *fast stack implementation*, which resembles the one of the vector type in STL for C++, as used for instance in CHAFF. The fast stack needs 3 pointers for its anchor and the array allocated on the heap always contains 32 bit integers.

Despite the much smaller memory requirements using compact stacks, in practice the trade off of time for space does not pay off. It seems that memory contention is not the main problem for SAT solvers and the gained cache efficiency of compact stacks is offset by an increase in executed instructions. Therefore, the default is to fall back to the fast stack implementation. However, for certain combinations of operating systems, compilers and processors, the compact stack implementation may be superior.

As the SAT solver COMPSAT entered the SAT'03 SAT solver competition it still contained one major flaw. This flaw was revealed in an early stage and resulted in the exclusion of COMPSAT from the competition. The problem consisted of failing to flush the assignment queue after restart and was introduced when moving the call to the restart function from the decision to the backtracking function. Since the restart interval is very large, restart was seldom triggered in the test suite. In those test case where a restart occurred the overall result was still correct. To avoid this problem in the future, we added a test case with a restart interval of one to our regression tests, which forces restart every other decision.

4 SAT Solver FUNEX

Initially, our SAT solver LIMMAT was an attempt to cross check the impressive results of CHAFF [7]. We also implemented some new ideas, which helped LIMMAT to actually improve on CHAFF in certain problem domains. In general, LIMMAT turned out not to be as robust as CHAFF. Our hypothesis was, that LIMMAT's two levels of indirection, when accessing watched literals, slow down BCP considerably. To test this hypothesis we implemented COMPSAT. The main goal was to have the smallest memory foot print possible and to simplify the implementation.

With COMPSAT we achieved this goal and our experiments confirmed that COMPSAT was an improvement over LIMMAT. Nevertheless COMPSAT still was not as robust as CHAFF. Originally, we started to work on LIMMAT again, but its complex data structures and some unfortunate design decisions, including lazy assignments stood in the way. It became clear that a new start would probably be better.

At the same time the new decision strategy of BERKMIN [5] was published. The results on standard benchmarks were slightly better than those of CHAFF. So we decided to support BERKMIN's decisions strategy in our new SAT solver FUNEX as well. Our experience with implementing LIMMAT and COMPSAT suggested that a more generic design was necessary to allow an easy transition between different heuristics without essentially rebuilding a whole SAT solver.

4.1 Watchers

We also followed the lesson from COMPSAT: indices instead of pointers are not much slower to handle and simplify the complexity of the implementation considerably. Clauses are stored in a separate clause data base and are accessed through their index. Each clause consists of a clause head, which beside the size of the clause contains an offset into the literal pool, a large integer stack.

Separated from the clause data base we maintain a stack of *watchers*. Watchers and clauses with the same index are mapped to each other. Each watcher contains two indices of the watched literals of the clause with the same index. Watchers are also used for caching the direction of the last traversal. During BCP, as in LIMMAT and in COMPSAT but not in CHAFF, watchers allow fast access to the other watched literal of a clause, avoiding superfluous traversals, when the other watched literal is satisfied.

Beside empty clauses and units, FUNEX distinguishes three classes of clauses: binary, small and large clause. The clauses in these classes only differ in their size. A binary clause has two literals. The bound for the size of a small clause is determined at run-time on start-up of the solver. The default is to classify non binary

clauses with at most four literals as small.

This 3-way classification allows to implement dedicated BCP procedures for each class. In particular, only large clauses are watched. The watchers of smaller clauses are not used, which is not possible if clause heads and watchers are merged.

4.2 Shorter Reasons First

The optimization in LIMMAT of *simplification of reasons* prefers shorter reasons. The intuition is, that shorter reasons decrease the size of the implication graph, which heuristically should produce shorter conflict clauses and longer back jumps in non chronological backtracking.

Following the same intuition, FUNEX changes the order of BCP in the following way. First the occurrence lists of variables are also separated into occurrences in binary clauses, small clauses and large clauses. Then all propagations are performed that involve only binary clauses, before small clauses are visited. Only if all binary and all small clauses of assigned variables are visited and did not produce any further assignment, then large clauses are visited. This technique has the additional advantage that cheap propagations are executed first, before more expensive propagations involving visits of large clauses are tried.

4.3 Satisfied Cache

In FUNEX we implemented the decision heuristics of BERKMIN, which in essence traverses the stack of learned clauses backward to find a still unsatisfied clause. Among the literals and its negations contained in this first unsatisfied clause, the unassigned literal with the largest activity according to the VSIDS heuristics of CHAFF is chosen as next decision. As in BERKMIN, if all learned clauses are satisfied, then we fall back to the original CHAFF heuristics and pick the literal with the largest activity among *all* still unassigned literals.

Even though using BERKMIN's decision heuristics helped to improve robustness of FUNEX, certain benchmarks became very slow. Profiling revealed that most of the time was spent in the search for a satisfied clause. To speed up this search we cache for each decision level the interval of indices of learned clauses, which have been searched earlier at this decision level and contains only indices of satisfied clauses. With this optimization we were able to reduce the percentage of time spent in the search for satisfied clauses from 90% of the total run-time to less than 10% in many cases.

4.4 Decision Strategies

Already in the first version of FUNEX as used in the SAT'03 SAT solver competition we allowed the specification of decision strategies at run-time. The specification is an omega regular expression over basic decision functions. Originally no choice operator was included and we only allowed *dlis*, *horn*, *chaff* and *berkmin* as basic decision function.

The decision functions *dlis*, the arguably best decision function implemented in GRASP, and *horn* as implemented in SATO [9], need to traverse all unsatisfied clauses. Even after restricting the traversal to the original clauses only, too much time is still spent in the decision function and we removed *dlis* and *horn* for the new version of FUNEX submitted to the SAT'04 SAT solver competition.

In the new version of FUNEX we introduced a *random* decision function. It randomly picks a still unassigned literal. Initially we picked a random variable index and incremented it modulo the number of variables until an unassigned variable is found. Again profiling revealed that for certain benchmarks, most of the time is spent in this decision function. The effect seems to be the same as with open addressing in hash tables: linear probing results in clustering and degrades performance considerably.

We use the same solution as in open addressing hashing. If the first randomly picked variable is assigned, then we generate another random number δ which is relative prime to the number of variables. Then instead of incrementing the variable index we add δ modulo the number of variables. As our experiments suggest this approach avoids the problems seen before. A typical strategy is specified as follows:

```
random^100.
(random.berkmin^3000)^infinity
```

It starts off with 100 random decisions, followed by an infinite repitition of 3000 berkmin decisions after one random decision.

Beside random decisions we implemented a *static* decision function, which is based on the original variable index alone. It picks an unassigned variable with the smallest variable index

as decision variable. The phase is determined by the literal activity. If partitioning is enabled, as described in the next section, a static precomputed variable order is used instead of the variable index order.

In addition, it is possible to randomly chose sub-strategies by using the choice operator. Only the Kleene star, which represents arbitrary, but finite, repetition, can not be used yet. The default strategy in the new version is:

```
random^100.
restart.
static^3000.
restart.
berkmin^3000.
(random|static|berkmin^99)^infinity
```

4.5 Partitioning

We strongly believe that the process of generating a CNF leaves its traces in the CNF. For instance if a circuit is translated into CNF via the standard Tseitin construction, and the gates are processed in depth first search, then the variable order reflects the levelized structure of the original circuit to some extent. For some equivalence checking benchmarks, we observed, that static decisions heuristics result in orders of magnitude less decisions.

The benchmarks used in previous SAT solver competitions have always been randomized, which destroys the natural variable order. To allow FUNEX to reclaim the original order in this or similar applications, we experimented with a min cut based partitioner. Our current implementation is slow and the quality of the results, measured in average cut size, is often poor. Therefore we seed the static order with a simple DFS ordering algorithm. Then the min cut partitioner is run. Only if it improves the average cut width the static order obtained from the partitioner is used. At the moment it is also not clear, whether average cut width is a good quality measure for variable orders.

4.6 Garbage Collection

A discussion with Niklas Eén destroyed our belief that garbage collection in SAT solvers is unnecessary. If all learned clauses are kept, then BCP becomes too expensive, particularly for hard combinatorial problems (usually in the old "handmade" category). Therefore we added an activity based garbage collector to FUNEX using similar techniques as described in [5].

5 SAT Solver NANOSAT

Our SAT solver NANOSAT is a reaction to the big success of the closed source solvers SIEGE and BERKMIN in version 561. In some of our experiments, these solvers outperformed all other solvers by a large margin. From the very sparse information we could obtain about implementation details, we deduced that the efficiency stems from a carefully low-level optimized implementation and not necessarily from algorithmic advances.

As first design decision of NANOSAT was to stick to statically allocated memory as much as possible. This should in principle allow compilers to produce faster and more compact code. For instance the variable stack is allocated statically and thus its start is never moved. Various other data structures, such as the *trail*, which stores the assigned variables in chronological order of their assignment, is allocated once and forall and will never be moved. No bound checks are necessary and instead of offsets to the trail, we can directly use pointers to its elements.

The occurrence lists of literals are implemented as real lists. The list nodes, the clause heads and the literals of large clauses are all managed by a moving garbage collector. This keeps the list nodes for the occurrence lists close together for better memory locality. As reported for the new version of FUNEX, NANOSAT follows the ideas of [5] to determine which clauses are collected. In addition, the moving garbage collector removes all top level satisfied clauses.

Binary clauses are replaced by two implications which are stored in separate implication lists. For instance if the CNF contains the clause $(a \vee \neg b)$ then the implication list of b contains a and the implication list of $\neg a$ contains $\neg b$. Again the moving garbage collector is helpful in keeping the implications of one literal in consecutive memory. We inherit from FUNEX the idea, that implications should be processed first before the literals of "real" clauses are visited, whenever a new assignment is deduced.

At the top level, after all units are propagated, the pure literal rule is applied in a further preprocessing step and, if successful, is followed by a call to the garbage collector. This is re-

peated for a fixed number of times and of course aborted if no more pure literals are detected.

After preprocessing the CNF a static variable order is determined by DFS. The decision heuristics follow the BERKMIN strategy. If the variables of an unsatisfied clause are compared the static variable order is used as last comparison criteria. The first comparison criteria is as usual the activity as in CHAFF. If all learned clauses are satisfied we fall back to the CHAFF heuristics. To find the most active variable, we use an $O(1)$ scheduler, inspired by JERUSAT.

Finally, NANOSAT contains one novel feature, which in its current version may actually result in slow BCP on instances with many large original or learned clauses. We improved the classical single counter based occurrence list scheme, that uses a single counter per clause, to count the number of its non false literals.

In the classical version, as soon the counter becomes 1, the whole clause has to be traversed, to find the unit, which then triggers an assignment. This search can be avoided as follows. In each clause head we not only have the counter of non false literals, but also a *sum* field that contains the XOR (exclusive or) of all the pointers representing non-false literals. If the counter becomes one, the sum field is equal to the new unit.

As always with counter based structures, the drawback compared to lazy data structures as in CHAFF is, that many more visits to clauses are necessary, though in our novel approach at least the literals itself do not have to be visited. The literals are only used in the analysis of the implication graph and in the BERKMIN style decision heuristics. In addition, during backtracking the counters and the sum fields have to be restored, which is not necessary for lazy data structures.

In order to reduce the number of visits to clauses, we do not connect large learned clauses. In the submitted version, learned clauses with more than 100 literals are not connected. Disconnected learned clauses influence the activities of variables as usual. They are also used in the search for still not satisfied clauses, which is organized as in FUNEX. As consequence a learned clause may be found with all its literals assigned to false. In this case the decision level has to be *adjusted* to the maximum decision level of all the literals in the contradictory clause, before standard conflict analysis is invoked. In practice, we observed that only a very small number of adjustments are necessary.

6 QUANTOR

The QBF solver QUANTOR removes subsumed clauses and uses unit propagation, the pure literal rule, equivalence reasoning and resolution to eliminate existential variables. Since all variables of a SAT problem seen as QBF problem are existentially quantified, we can use QUANTOR as preprocessor to simplify a SAT problem, before it is handed to a standard SAT solver.

6.1 Equivalence Reasoning

To detect equivalences we search for pairs of dual binary clauses. A clause is called *dual* to another clause if it consists of the negation of the literals of its dual. If such a pair is found, we take one of the clauses and substitute the larger literal by the negation of the smaller one throughout the whole CNF.

The search for dual clauses can be implemented efficiently by hashing binary clauses. In more detail, whenever a binary clause is added, we also save a reference to it in a hash table and check, whether the hash table already contains a reference to its dual. If this is the case an equivalence is found. After an equivalence is found, it is used to eliminate one of the variables of the equivalence.

6.2 Resolution

Variables in a propositional CNF can be eliminated by resolution as was proposed in the original DP paper [4]. In [2] we present an efficient on-the-fly resource driven scheduler that determines the order in which variables are eliminated. Cheapest to eliminate variables are eliminated first. Elimination by resolution continues as long the number of added literals remains below a certain threshold, 100 literals by default.

6.3 Subsumption

Resolution may produce a lot of redundant subsumed clauses. Therefore, subsumed clauses should be removed. If a new clause is added, all old clauses are checked for being subsumed by this new clause. This check is called backward subsumption and can be implemented efficiently

on-the-fly, by using a signature-based algorithm. However, the dual check of forward subsumption is very expensive and is only invoked periodically, for instance at each expansion step. More details on the subsumption algorithm can be found in [2].

6.4 Lifting Partial Assignments

After elimination by resolution becomes too costly and no further simplifications are possible, then the CNF is handed over to our SAT solver FUNEX. If FUNEX determines unsatisfiability, the original formula is also unsatisfiable, which immediately is reported by QUANTOR. However, if a satisfying assignment to the simplified CNF is found, this assignment still has to be lifted to a full assignment of all variables of the original CNF.

In our implementation, we simply copy the original CNF, add the partial assignment obtained from the first run of FUNEX together with eliminated units and pure literals as unit clauses. We also include pairs of binary clauses representing equivalences for all variables eliminated by equivalence reasoning. The resulting CNF is satisfiable. In practice FUNEX easily finds a full satisfying assignment, which is then reported to the user by QUANTOR.

The second CNF handed to FUNEX is usually much larger than the first. Therefore the expensive min cut partitioning algorithm in FUNEX is switched off. It usually is used to obtain a good static variable order. For the version of FUNEX linked to QUANTOR submitted for the SAT competition, partitioning is switched off for the first run of FUNEX and if the CNF is satisfiable also for the second run.

7 Conclusion

We have described the evolution of our SAT solvers. These solvers do not share much code, but of course they share many ideas and particularly design decisions. Some pitfalls discovered in earlier solvers have been avoided in recent ones. Nevertheless, it is clear that plenty more optimizations exist and should be investigated. Breakthroughs are still possible.

References

[1] R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1997.

[2] A. Biere. Resolve and expand. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, 2004.

[3] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394–397, July 1962.

[4] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[5] E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust Sat-Solver. In *Proceedings of Design Automation and Test in Europe*, pages 142–149, March 2002.

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

[7] M. H. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the* 38^{th} *Design Automation Conference*, pages 530–535, June 2001.

[8] A. Van Gelder. Generalizations of watched literals for backtracking search. In *Seventh International Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002.

[9] H. Zhang. SATO: An Efficient Propositional Prover. In *Proc. of the International Conference on Automated Deduction*, pages 272–275, July 1997.

[10] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, 2001.