Space-Efficient Bounded Model Checking

Author names

Abstract

Current algorithms for bounded model checking use SAT methods for checking satisfiability of Boolean formulae. Methods based on the validity of Quantified Boolean Formulae (QBF) allow an exponentially more succinct representation of formulae to be checked, because no "unrolling" of the transition relation is required. These methods have not been widely used, because of the lack of an efficient decision procedure for QBF. In this paper we present an algorithm for bounded model checking that uses as succinct representation of formulae as possible with QBF-based techniques. We also provide a comparison of our technique with SAT-based and QBF-based ones, using a few available solvers, on real-life industrial benchmarks.

1 Introduction

Model checking is a technique for the verification of the correctness of a finite-state system with respect to a desired behavior. The system is traditionally modeled as a labeled state-transition graph, and the behavior is specified by a temporal logic formula [1]. Early implementations, based on explicit-state model checking [2], suffered from the state explosion problem. The introduction of symbolic model checking with BDDs [3, 4] and other recently developed methods succeeded in partially overcoming this problem and enabled industrial applications of model checking for real-life systems, mostly in the hardware industry. However, all those methods still suffer from the memory explosion problem on modern test cases. In this work we present a method which is much more space-efficient, sometimes by orders of magnitude, in comparison with the existing techniques.

In symbolic model checking the states of the state-graph are encoded by a vector of Boolean encoding variables; sets of states are represented with characteristic functions; transitions in the graph are represented with a transition relation: that is, a propositional formula over two sets of encoding variables. Properties are usually specified in a temporal logic. In this work we restrict our attention to safety properties – those that can be disproved by examining a finite computation path.

Symbolic model checking uses image computation to verify properties. Despite the increased capacity, compared to an explicit-state model checking, BDD-based techniques still suffer from the state explosion problem for models beyond a few hundred state variables, because BDDs do not always represent Boolean functions compactly.

In [5, 6], the authors evaluate SAT methods instead of BDDs for image computation. They use an explicit quantifier elimination to reduce the problem of reachability checking to the problem of propositional satisfiability, and then apply SAT solvers to check the satisfiability. The explicit quantifier elimination, however, causes exponential blow-up in the size of the generated formulae in the worst case.

More recent papers [7-13] propose algorithms for SAT-based reachability analysis, where the quantifier elimination is implicitly implemented in the SAT solvers. The SAT solvers are modified such as to find all possible solutions rather than one solution, by adding a blocking clause for each new solution found. Storing all the solutions in a compact data structure is a challenge, though. There have been attempts to use BDDs, zero-suppressed BDDs, or disjunctive normal form, but all of them are still of exponential size in the worst case.

Bounded Model Checking (BMC), introduced in [14, 15], is based on the representation of computation paths of a bounded length that falsify the property being checked. BMC with a specific bound k represents the paths of length k in the system by "unrolling" the transition relation k times, and examines whether the set of states falsifying the property is reached by these paths. The composed formula is sent to a SAT solver, and any satisfying assignment represents a counter-example of length k for the property. The usage of BMC increased the capacity of model checkers to thousands of state variables, however, at a price: one no longer gets a fully certified answer to the verification problem, but rather an assurance that there are no counterexamples of a given length. To implement a complete model checking procedure the bound should be increased iteratively up to the length of the longest simple path in the system. Determining the sufficient bound for BMC is generally intractable, but it is exponential in the number of encoding variables in the worst case. Hence, the number of copies of the transition relation within the formulae being checked for validity increases from iteration to iteration up to an exponential number of times, leading, again, to a memory explosion for large systems and large

bounds.

Induction based methods [16] provide another technique for estimating whether a bound is sufficient to ensure a full proof. Induction is particularly successful for local properties, but there are still many cases where the induction depth is exponential in the size of the model.

Finally, in [17] the author uses Craig interpolation as an over-approximation technique for image computation aimed at reducing the number of iterations for a complete model checking procedure. The interpolants are obtained as a by-product of the SAT solver used to check BMC problems. This technique, like other techniques based on image computation, also suffers from a potential memory blow-up.

In this paper we present an algorithm for BMC, which does not perform unrolling of the transition relation and, thus avoids the memory explosion problem.

The rest of the paper is organized as follows. Section 2 presents formulations of the bounded reachability analysis problem with propositional and quantified formulae. Section 3 describes our new algorithm for such an analysis; and Section 4 compares the performance of our algorithm to the usage of the existing SAT and QBF solvers in BMC. In section 5 we conclude and present future directions.

2 Formulations of bounded reachability checking problem

Given a system M=(S, I, TR), where S is the set of states, I is the characteristic function of the set of the initial states, and TR is the transition relation, the problem of reachability of the final states given by a characteristic function F in exactly k steps can be expressed in a number of ways.

As in BMC [14], the fact that the state Z_k is reachable from the state Z_0 in exactly k steps may be formulated by "unrolling" the transition relation k times:

$$(1)\quad R_k(Z_0,Z_k) = \exists Z_1,\ldots,Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \bigwedge_{i=0}^{k-1} TR(Z_i, Z_{i+1})$$

The validity of this formula may be proven or disproved by performing the SAT decision procedure on the propositional formula:

$$(2)\quad I(Z_0) \wedge F(Z_k) \wedge \bigwedge_{i=0}^{k-1} TR(Z_i, Z_{i+1})$$

Noticeably, the number of copies of the transition relation in this formula is as the number of steps being checked. When iteratively increasing the bound k, each next iteration checks reachability of the final states in one more step than the previous iteration. Thus, for a complete check, SAT procedure needs to be invoked on formulae containing an exponential number of copies of the transition relation.

To partially overcome the potential memory explosion, a QBF formulation of bounded reachability problem can be used:

$$(3)\quad \begin{aligned} &R_k(Z_0,Z_k) = \exists Z_1,\ldots,Z_{k-1} : I(Z_0) \wedge F(Z_k) \wedge \\ &\forall U,V : \left(\bigvee_{i=0}^{k-1} (U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1}) \right) \rightarrow TR(U,V) \end{aligned}$$

Note that (3) contains only one copy of the transition relation. Increasing the bound, thus, would mean an addition of a new intermediate state and a term of the form $(U \leftrightarrow Z_i) \wedge (V \leftrightarrow Z_{i+1})$. Hence, the formula increase from iteration to iteration does not depend on the size of the transition relation, which is usually the biggest formula in the specification of the model.

The solution of (3) with a QBF solver usually requires a transformation of the propositional part of the formula into a CNF. Linear-time translation of a propositional formula to an equisatisfiable CNF formula [18] introduces artificial variables, resulting with a QBF having $\exists\forall\exists$ pattern of the quantifier prefix. The number of the universally quantified variables does not change in the QBF from iteration to iteration.

This approach to reachability checking partially solves the issue of formula growth, reducing the growth of the formula from iteration to iteration, but still requires an exponential number of iterations to fully verify the reachability.

To reduce the number of iterations, it is possible to apply the "iterative squaring" technique, similar to the one used in BDD-based model checking [1]. In this technique, each successive iteration checks the reachability of a final state in twice as many steps as the previous iteration. Given a formula $R_{k/2}(X,Y)$ for checking reachability in $k/2$ steps, the following formula checks the reachability in k steps:

$$(4)\quad \begin{aligned} &R_k(Z_0,Z_k) = \exists Z : I(Z_0) \wedge F(Z_k) \wedge \forall U,V : \\ &\left[(U \leftrightarrow Z_0) \wedge (V \leftrightarrow Z) \vee (U \leftrightarrow Z) \wedge (V \leftrightarrow Z_k) \right] \rightarrow R_{k/2}(U,V) \end{aligned}$$

The transition relation appears in (4) only once, as in the previously described technique. However, the number of universally quantified variables and the number of quantifier alternations grows from iteration to iteration.

This technique allows reducing the number of iterations to be as the number of the state encoding variables in the model, since it would then cover the

```
InitializeCurrentState();
while (true) {
   bCurrentAndNextStatesDecided = SelectDecisionVariable();

   If (bCurrentAndNextStatesDecided == true) {
      If (AllStatesDecided() == true)  return true;

      bUnresolvableConflict = AdvanceCurrentState();
      if (bUnresolvableConflict == true) return false;
   }
   do {
      bConflictProduced = BCP();
      if (bConflictProduced == true) {
         if (ResolveConflict() == false) return false;
      }
   } while (bConflictProduced == true);
}
```

Fig. 1a - Pseudo code for jSAT algorithm

```
ResolveConflict()
{
   nBacktrackingLevel = AnalyzeConflict();

   bFirstUndecidedState = Backtrack(nBacktrackingLevel);

   bUnresolvableConflict =
      RetractCurrentState(bFirstUndecidedState);

   if (bUnresolvableConflict == true) return false;
   return true;
}
```

Fig. 1b - Pseudo code for jSAT algorithm

worst-case diameter of the model. Note that not all bounds are checked by this technique, but only the bounds that are a power of 2. It is possible, however, to overcome this problem by adding a self-loop in each state of the model, which would not change the reachability between states, but rather make (4) check reachability in *k or fewer* steps, instead of *exactly k* steps.

We have used a bounded model checker to generate the three kinds of formulae mentioned above. We have evaluated a few available state-of-the-art DPLL-based SAT and QBF solvers, to check the feasibility of the QBF formulations of the reachability checking problem on a number of real-life industrial examples. Section 4 presents the details of the evaluation. Unfortunately, the QBF solvers were unable to solve practically any of the formulae, while many of the corresponding propositional formulae were solved by the SAT solvers, many of them in a matter of seconds.

Noticeably, all the three kinds of formulae contain exactly the same information, but in different form. In (2) the formula contains explicitly the relation between any two successive states in the path from the initial state to the final one. Such an explicit representation often allows a solver to restrict the choice of a next state in the path immediately as the previous one has been chosen. For example, when in (2) the state Z_0 has been chosen by the algorithm, the Boolean Constraint Propagation (BCP) process [20] would possibly deduce the values of some variables in Z_1. Also, the choice of an impossible value for one of Z_1's variables could immediately cause a conflict. We may say that, in a sense, the SAT-based approach examines for being final *only* the states within the set of states reachable from the initial ones.

In the QBF-based approaches this is not the case. The information about the relation between any two successive states is not found explicitly in the formula. Therefore, in the formula (3) the DPLL-based solvers are unable to deduce anything about Z_1, when Z_0's value is set. This is because the relation between Z_0 and Z_1 is dependent on all possible choices of U and V. Additionally, a general-purpose DPLL-based QBF solver is restricted in the decision process to first set the values for the variables quantified in the outer level (Z_0, Z_1, etc.), before proceeding to the inner ones (U and V). Essentially, this means that the solver first chooses values for Z_0, Z_1, …, Z_k and only then checks whether such a choice constitutes a path in the model. In the solution of (4), not all states in the path have variables representing them in the formula, which further complicates the solution process.

3 jSAT decision procedure

Our research was motivated by the inefficiency of QBF solvers demonstrated on formulae of the form (3). To improve the efficiency of the solution, it is required to achieve a similar way of exploration of the state space as a SAT solver would make on an equivalent propositional formula of the form (2). Indeed, our algorithm tries to emulate in some sense the behavior of a SAT solver on (2), while maintaining in memory the representation similar to that of (3). In fact, as in (3), jSAT holds in memory the encoding variables representing the states Z_0, Z_1, …, Z_k, U and V, but only holds the following propositional formula:

(5) $I(Z_0) \wedge TR(U,V) \wedge F(Z_k)$

The states Z_i represent a path; the states U and V represent two neighboring states in that path. Instead of explicitly holding the fact that U and V represent a pair of neighboring states as done in (3) with assistance of the terms of the form $(U \leftrightarrow Zi) \wedge (V \leftrightarrow Z_{i+1})$, our algorithm implicitly assumes this information. The idea of the algorithm is to iteratively associate U and V with a pair of successive states, called the current state and the next state, until all states are decided. We call U and V aliases, since at each point of time during the algorithm they act as the states they are associated with.

The pseudo code of the algorithm is shown in Fig. 1. It

is very similar to the classic DPLL algorithm [19, 20, 21], which is widely used in the current state-of-the-art SAT and QBF solvers. Given a CNF representation, DPLL algorithm iteratively chooses variables to assign with a value, as long as the partial assignment to the variables does not falsify the formula. When the partial assignment is found to falsify the formula, i.e. a conflict is discovered, the algorithm backtracks by unassigning some of the assigned variables, and assigning an opposite value to one of them. The process by which the algorithm decides which variables to unassign and which variable to assign with an opposite value is called conflict analysis. The algorithm also incorporates an optimization called Boolean constraint propagation (BCP) or unit propagation [20], which aims at speeding up the search by making obvious decisions as soon as they can be made, based on the unit literal rule: if the current partial assignment causes all but one literal of a clause to have the value false, then the remaining literal must be assigned true in order not to falsify the clause and, consequently, the whole formula.

Most of the current state-of-the-art SAT and QBF solvers use additional optimization techniques, such as conflict-driven learning and non-chronological backtracking [20]. Our approach does not rely on any of above optimizations to be present, but these optimizations provide a significant advantage, as with other existing solvers. Numerous other optimization techniques exist, which may also prove helpful in jSAT.

Intuitively, jSAT algorithm can be seen as a depth-first search in the state graph of the system from the initial states to the final ones. The algorithm starts by associating U with Z_0 and V with Z_1; thus the formula (5) becomes semantically equivalent to:

$$(6) \quad I(Z_0) \wedge TR(Z_0, Z_1) \wedge F(Z_k)$$

The states Z_0 and Z_1 are then decided, if possible, so that Z_0 is an initial state and Z_1 is its successor. As soon as they are decided, the algorithm makes Z_1 to be the current state and Z_2 to be the next one: U becomes an alias to Z_1, and V becomes an alias to Z_2. The algorithm proceeds so on, until all states are successfully decided, or until it discovers that such a decision is impossible. Decision on each state involves assignment for each of the corresponding encoding variables.

The described process may also be seen as two nested search procedures: one looking for a path in the model, and another, nested one, looking for a suitable next state (V) from the last found state (U) in the path. Whenever all encoding variables of the next state V are successfully assigned, this state becomes the current-state by associating its variables with U and the variables from the next state with V. Whenever a conflict caused by an assignment to a variable from the current-state U is discovered, the path search backtracks by setting U to be the previous state and V to be the unsuccessfully chosen current state; it then tries to find another suitable such state. Eventually, either all states have been successfully chosen, which means (3) is valid, or all the initial states have been unsuccessfully enumerated, which means (3) is not valid.

The procedure SelectDecisionVariable() in Fig. 1 selects a still unassigned variable out of the encoding variables of the current state and, if all the encoding variables of the current state are assigned, out of the next state. It returns true, if all the encoding variables of the current and the next states have been decided – at this moment the current state is advanced. We restrict the decision strategy to selecting decision variables in the order of states in the path: encoding variables of the state Z_0 are selected first, then the variables of Z_1, then the variables of Z_2, and so on. Such a restriction causes the algorithm to implement a depth-first search of the state graph and to "visit" only the states actually reachable from the initial states. The order of selection of the encoding variables within one state is not important, and heuristics similar to the ones existing in SAT/QBF solvers can be used.

Adjustment of the current and the next state happens not only when a next state is successfully chosen. When, as part of conflict resolution by ResolveConflict(), the algorithm backtracks and a variable of the current state becomes unassigned, the current and the next state are retracted so that the next state V is associated with the earliest undecided state in the path.

When the current and the next states are adjusted, new relations between the encoding variables become apparent. Thus, for example, when U and V are moved from the pair of states (Z_0, Z_1) to the next pair (Z_1, Z_2), the relations between the encoding variables of Z_1 and Z_2 become explicit in TR(U, V). Since the newly discovered information may contradict some of the already made decisions, conflicts may arise during the adjustment operations.

An important difference of our algorithm from the SAT solvers follows from the fact that U and V represent different states among Z_i at different points of time. It is, therefore, generally incorrect to produce learned conflict clauses that involve variables of U, V or any artificial variable resulting from the translation of TR(U, V) to CNF, as they will become useless as soon as U and V are adjusted to represent another pair of states. Therefore, the learned clauses must be formulated in terms of encoding variables of Z_i. Our conflict analysis technique achieves this by using only decision variables in the learned clauses, somewhat similar to Last UIP learning scheme described in [27].

	# state vars	jSAT	[22]	Intel solver
test08	10	16	18	18
test12	11	18	18	18
test10	12	18	18	18
test03	39	18	18	18
test06	160	1	12	17
test09	160	18	18	18
test05	199	0	18	18
test11	220	17	18	18
test04	626	1	6	18
test13	662	18	18	18
test02	914	0	6	15
test07	1055	0	11	18
test01	2013	18	5	9
Total (out of 234):		*143*	*184*	*221*

Table 1 - Number of bounds solved by each solver per test case.

	jSAT		[22]		Intel solver	
	sec	**MB**	sec	**MB**	sec	**MB**
test08	*>300*	**2.5**	0.3	**3.4**	0.0	**3.1**
test12	0.0	**3.1**	0.0	**3.1**	0.0	**3.1**
test10	0.2	**2.6**	1.2	**4.9**	0.5	**4.9**
test03	0.0	**3.1**	0.0	**3.1**	0.0	**3.1**
test06	*>300*	**4.4**	*>300*	**46.1**	*>300*	**50.3**
test09	0.0	**3.1**	3.8	**20.3**	0.8	**36.0**
test05	*>300*	**9.7**	74.5	**40.8**	9.8	**36.8**
test11	205.1	**4.8**	8.9	**25.6**	2.7	**44.2**
test04	*>300*	**19.5**	*>300*	**264.0**	292.4	**227.0**
test13	2.0	**9.7**	109.1	**95.9**	22.2	**104.1**
test02	*>300*	**15.7**	*>300*	**80.4**	*>300*	**133.8**
test07	*>300*	**19.5**	*>300*	**145.5**	293.1	**144.7**
test01	5.4	**44.6**	*>300*	**624.8**	14.2	**>1024**

Table 2 - Run-time and memory usage of each solver for bound 20 per test case.

4 Experimental results

We have implemented jSAT algorithm to measure its applicability to the problem of BMC. Our implementation is based on the single-threaded version of the solver described in [22], which is reported to have slightly slower performance than zChaff [23].

In our implementation the clause set is represented with Watched Literals data-structure. Our decision strategy selects variables from earlier states of the path first; among the variables of the same state the decisions are chosen according to VSIDS heuristics [23].

The implication graph [27] is not explicitly built in our algorithm. Instead, each variable is associated with the clause that implied it – the antecedent clause, if the variable was assigned by the BCP process. Because of this, there is some information loss incurred by the operation of adjustment of the current and the next states: if a variable's antecedent clause belongs to TR(U, V) part of the formula, then it is incorrect to consider that clause antecedent after the adjustment of U and V to another pair of states. Thus, some or all of the edges of the implication graph are lost during the adjustment. This fact significantly affects the efficiency of conflict-driven learning, as more variables need to be included in the learned clauses than would be possible if the information were not lost.

We implemented non-chronological backtracking and conflict-driven learning, but no restarting or other advanced optimizations.

To measure the applicability of our algorithm we used a bounded model checker to generate formulae of the forms (2), (3), (4) and (5). The formulae of the form (2) were generated in DIMACS format and can be fed into many available SAT solvers. The formulae of the forms (3) and (4) were generated in QDIMACS format and can be fed into available QBF solvers. The formulae of the form (5) are generated in a slightly customized DIMACS format, which adds the specification of the encoding variables to the formula description.

We used a set of thirteen proprietary Intel® model checking test cases of different sizes to compare the run-time and memory consumption of our algorithm to those of the SAT and QBF solvers on formulae described above. For each test case we generated formulae of all kinds for the bounds in range from 3 to 20, resulting in the total amount of 234 formulae of each kind. Twenty of the formulae of the form (3) and (4) were publicly disclosed and participated in the QBF solver evaluation during SAT2004 conference. We used a dual Intel® Xeon™ 2.8 GHz Linux RH7.1 workstation with 4GB of memory for the experiments, and set a 300 seconds time out and 1 GB memory out limits on all the solvers.

We used QuBE [24] and Semprop [25] QBF solvers to solve formulae of the form (3) and (4) for all the test cases. Unfortunately, both solvers were unable to solve practically any of the 234 formulae (QuBE, in fact, managed to solve three of them). We do not relate to the performance of the QBF solvers in the following paragraphs for brevity.

To solve formulae of the form (2), we used the solver described in [22], and an Intel proprietary DPLL-based solver, which performs as good as the best current publicly available solvers. We compare the run-time and memory usage of these solvers with jSAT. Table 1 shows the sizes of the test cases in terms of the state variables in the model, and the number of formulae each of the solvers successfully coped with. As evident, jSAT managed to solve more than a half of the formulae, significantly more than the general-purpose QBF solvers.

Table 2 shows the run-time and the memory usage of

the three solvers on all the test cases for bound 20. As expected, the memory consumed by jSAT is significantly lower than by the other solvers, in some cases by orders of magnitude. Noticeably, in the largest test case "test01", jSAT achieved a better run-time, in addition to the significantly lower memory usage.

The slower run-time of jSAT can be attributed mainly to the following reasons:

- The overhead of the state adjustment operation is very high in our implementation. Indeed, the number of path backtracks in the depth-first path search in a highly connected state graph is very large.
- There is a loss of information about antecedent clauses, as described above.
- Our implementation does not use many of the advanced optimizations, which are implemented in the other solvers.

5 Conclusions

We presented an algorithm for checking whether a set of final states is reachable from a set of initial states in a state graph of a system for use in BMC. Our algorithm solves quantified formulae of the form (3) for real-life industrial examples, which cannot be solved by the to-date state-of-the-art QBF solvers. The main contribution of this work is in that our algorithm is significantly more space-efficient than the existing SAT-based BMC algorithms, as it does not require "unrolling" of the transition relation.

A number of improvements to our algorithm can be made, and they are subject for the future research:

- Data structures for efficient state adjustment operations.
- Alternative representation of the implication graph to avoid information loss incurred by state adjustments.
- Incorporation of additional optimization techniques used in the current state-of-the-art solvers.

6 References

[1] E. Clarke, O. Grumberg, D. Peled. "Model Checking". MIT Press, 2000.

[2] E.M. Clarke, E.A. Emerson, A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". ACM Transactions on Programming Languages and Systems, 8(2):244-263, 1986.

[3] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". IEEE Trans. Computers 35(8), 1986.

[4] K. L. McMillan. "Symbolic Model Checking". Kluwer Academic Publishers, 1993.

[5] P.A. Abdulla, P. Bjesse, N. Een. "Symbolic Reachability Analysis based on SAT Solvers". TACAS, 2000.

[6] P.F. Williams, A. Biere, E.M. Clarke, A. Gupta. "Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking". CAV, 2000.

[7] O. Grumberg, A. Schuster, A. Yagdar. "Reachability using Memory Efficient All-Solutions SAT Solver". FMCAD, 2004.

[8] K.L. McMillan. "Applying SAT Methods in Unbounded Symbolic Model Checking". CAV, 2002.

[9] H. J. Kang, I.-C. Park. "SAT-Based Unbounded Symbolic Model Checking". DAC, 2002.

[10] A. Gupta, Z. Yang, P. Ashar, A. Gupta. "SAT-Based Image Computation with Application in Reachability Analysis". FMCAD, 2000.

[11] P. Chauhan, E. M. Clarke, D. Kroening. "Using SAT based Image Computation for Reachability Analysis". Technical Report CMU-CS-03-151, CMU, School of Computer Science, 2003.

[12] B. Li, M. S. Hsiao, S. Sheng. "A Novel SAT All-Solutions Solver for Efficient Preimage Computation". DATE, 2004.

[13] M. Iyer, G. Parthasarathy, K.-T. Cheng "SATORI -- A Fast Sequential SAT Engine for Circuits". ICCAD, 2003.

[14] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. "Symbolic Model Checking without BDDs". TACAS, 1999.

[15] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu. "Symbolic Model Checking Using SAT Procedures instead of BDDs". DAC, 1999.

[16] M. Sheeran, S. Singh, G. Stalmarck. "Checking Safety Properties Using Induction and a SAT-Solver". FMCAD, 2000.

[17] K.L. McMillan. "Interpolation and SAT-based Model Checking". CAV, 2003.

[18] D. A. Plaisted, S. Greenbaum. "A structure-preserving clause form translation". Journal of Symbolic Computation 2 (1986), 293-304.

[19] M. Davis, G. Logemann, D. W. Loveland. "A machine program for theorem proving". Journal of the ACM, 394-397, 1962.

[20] I. Lynce, J. P. Marques-Silva. "An Overview Of Backtrack Search Satisfiability Algorithms". 5th Intn'l. Symp. on Artificial Intelligence and Mathematics, 1998.

[21] J. Gu, P. W. Purdom, J. Franco, B. W. Wah, "Algorithms for the satisfiability (SAT) problem: A survey", URL: http://citeseer.ist.psu.edu/56722.html, 1996.

[22] Y. Feldman, N. Dershowitz, Z. Hanna. "Parallel Multithreaded Satisfiability Solver: Design and Implementation". PDMC, 2004.

[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. "Chaff: engineering an efficient SAT solver". DAC, 2001.

[24] QuBE QBF solver. URL: http://www.qbflib.org/~qube/.

[25] R. Letz. "Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas". TABLAUX, 2002. URL: http://www4.informatik.tu-muenchen.de/~letz/semprop/.

[26] D. Le Berre, l. Simon, A. Tacchella. "Challenges in the QBF arena: the SAT'03 evaluation of QBF solvers". SAT, 2003.

[27] L. Zhang, C. F. Madigan, M. H. Moskewicz, S. Malik. "Efficient Conflict Driven Learning In A Boolean Satisfiability Solver". ICCAD, 2001.