Towards Lightweight Satisfiability Solvers for Self-Verification
(Invited Paper)

Fritjof Bornebusch[1] Robert Wille[1,2] Rolf Drechsler[1,3]

[1] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[2] Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
[3] Institute of Computer Science, University of Bremen, Germany
fritjof.bornebusch@dfki.de robert.wille@jku.at drechsler@uni-bremen.de

***Abstract*—Solvers for Boolean satisfiability (SAT solvers) are essential for various hardware and software verification tasks such as equivalence checking, property checking, coverage analysis, etc. Nevertheless, despite the fact that very powerful solvers have been developed in the recent decades, this progress often still cannot cope with the exponentially increasing complexity of those verification tasks. As a consequence, researchers and engineers are investigating complementarily different verification approaches which require changes in the core methods as well. Self-verification is such a promising approach where e.g. SAT solvers have to be executed on the system itself. This comes with hardware restrictions such as limited memory and motivates lightweight SAT solvers. This work provides a case study towards the development of such solvers. To this end, we consider several core techniques of SAT solvers (such as clause learning, Boolean constraint propagation, etc.) and discuss as well as evaluate how they contribute to both, the run-time performance but also the required memory requirements. The findings from this case study provide a basis for the development of dedicated, i.e. lightweight, SAT solvers to be used in self-verification solutions.**

1. Introduction

Embedded and cyber-physical systems, as they frequently occur in microchips in almost all parts of our life, are usually expected to be error-free. In order to guarantee correctness, verification methods such as simulation, emulation, or formal verification are applied before production and deployment of circuits and systems. However, the ever-increasing complexity as well as time-to-market constraints often force designers to terminate the verification process before full functional correctness has be ensured. This frequently allows bugs to escape into the final product – a problem which is usually associated with the term *verification gap*.

Most of the existing approaches to address this problem rely on iterative improvements of existing solutions, e.g. raising the abstraction level [1]–[3] or apply a combination of complementary verification techniques [4]. However, these developments will hardly be able to comprehensibly tackle the (exponentially increasing) complexity posed by most verification problems. Hence, researchers and engineers need to investigate fundamentally different approaches to deal with this problem.

One possible direction towards this has recently been proposed in terms of a method called *self-verification* [5], [6]. Here, the general idea is to equip a system with capabilities that allow for completing all the verification tasks that could not been mastered before. While this still would require a verification phase prior to deployment (in order to particularly detect safety-critical errors), 100% completeness could eventually been achieved by the system itself in the field[1].

However, this direction also requires changes in the underlying methods which are employed for verification. Thus far, verification mainly relies on the availability of powerful reasoning engines. Here, solvers for *Boolean satisfiability* (SAT solvers, [7]) received significant interests and are employed for many verification tasks such as equivalence checking [8], model checking [9], coverage analysis [10], test pattern generation [11], etc. In the past, corresponding solvers have been significantly improved (leading e.g. to several derivatives such as *Satisfiability modulo theories* (SMT) solvers [12], word-level solvers [13], *Quantifier-free bit-vector* (QBF) solvers [14], etc.) – eventually yielding and enabling very powerful methods for verification.

For the purpose of self-verification, however, these solvers have to be executed on the system itself, i.e. usually with restricted hardware resources. However, most of the modern SAT solvers are optimized towards the best possible run-time performance – usually relying on high-speed CPU's and potentially a large amount of memory. Since those solvers cannot be applied on cyber-physical or embedded systems, corresponding *lightweight* versions of them have to be available. This significantly changes the requirements of the verification methods needed for self-verification: While pure run-time performance is not the main criteria anymore (in fact, as the verification tasks are completed after deployment anyway, it is acceptable to somewhat loose performance), the limited resources of the system becomes a crucial factor. Designers eventually need to trade-off between run-time performance and limited resources.

In this work, we aim to provide a better understanding of these contradictory issues. To this end, we consider several core techniques of SAT solvers (such as clause learning, Boolean constraint propagation, etc.) which got established in the past. Then, we discuss how they contribute to both, the run-time performance but also the required hardware requirements. For the latter, we consider thereby the memory consumption as this is the most obvious limitation of a cyber-physical or embedded system. Based on these discussions, several lightweight versions of a state-of-the-art SAT solver are derived and experimentally compared. The results provide insights on how lightweight solving engines for self-verification should mainly depend on.

1. For details on the idea of self-verification as well as a discussion of possible application scenarios, we refer to [5], [6].

The remainder of this paper is structured as follows: The next section gives a short overview about modern SAT solver and the techniques they use. Afterwards, possible directions towards lightweight SAT solvers are discussed in Section 3. This eventually yields several possible configurations which are described in Section 4 and evaluated in Section 5. Finally, the paper in concluded in Section 6.

2. Background

This section gives a brief overview on the SAT problem as well as modern SAT solvers which have been developed to solve it. Besides that, we briefly review main techniques of SAT solvers and discuss their effect towards a lightweight solution.

2.1. Boolean Satisfiability

The *Boolean satisfiability problem* (SAT problem) is defined as follows: Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$, where $n \in \mathbb{N}$ is a Boolean function. Then, the SAT problem is to determine an assignment to the variables of f such that f evaluates to 1 or to prove that no such assignment exists. In other words, SAT asks if $\exists X\, f$ for an f over variables X and determines a satisfying assignment in this case. The Boolean formula f is often given in *Conjunctive Normal Form* (CNF). A CNF is a set of clauses, each clause is a set of literals, and each literal is a Boolean variable or its negation. The CNF formula is satisfied if all clauses are satisfied, a clause is satisfied if at least one of its literals is satisfied, and a variable is satisfied when 1 is assigned to the variable (the negation of a variable is satisfied under the assignment 0).

Example 1. *Let $f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. A possible satisfying solution would be $x_1 = 1$ and $x_2 = 0$, i.e. f is satisfiable.*

2.2. SAT Solvers

The Boolean Satisfiability problem (SAT) is under research for decades. In 1971, the Cook-Levin theorem showed that this problem is NP-complete, what means there is no algorithm known to solve it in polynomial time [15]. Nevertheless, very powerful reasoning engines (so called *SAT solvers*) have been proposed in the past (see e.g. [7], [16]–[19]), which are capable of solving instances composed of hundreds of thousands of variables and clauses. Most of them apply the steps depicted in Fig. 1: While there are free variables left (a), a decision is made to assign a value to one of these variables (c). Then, implications are determined due to the last assignment (d). This may cause a conflict (e) that is analyzed. If the conflict can be resolved by undoing assignments from previous decisions, backtracking is done (f). Otherwise, the instance is unsatisfiable (g). If no further decision can be made, i.e. a value is assigned to all variables and this assignment did not cause a conflict, the CNF is satisfied (b).

In this context, advanced techniques such as *efficient Boolean constraint propagation* [18] or *conflict analysis* [17] as well as *efficient decision heuristics* [19] are common in state-of-the-art SAT solvers today.

3. Towards Lightweight SAT Solvers

As reviewed in Section 1, SAT solvers have mainly been optimized with respect to run-time performance. With the emergence of alternative directions such as self-verification, also the required memory consumption becomes an issue.

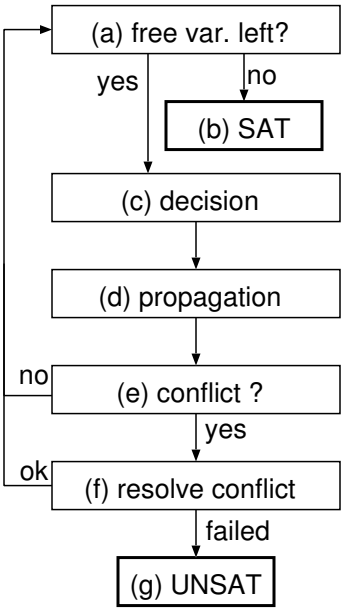

Figure 1. Main flow of modern SAT solvers

This has hardly been considered thus far. Exceptions might be the works proposed in [20] and [21], which however focused more on CPU caches and dedicated hardware designs for SAT solvers, respectively. In this work, we aim to investigate the effects of state-of-the-art SAT solving techniques such as Boolean constraint propagation, conflict analysis, and heuristics on the resulting memory consumption and in trade-off with the run-time performance. To this end, these techniques are reviewed in a bit more detail and discussed with respect to this issue in this section. This eventually motivates certain lightweight setups of SAT solvers which are eventually described and evaluated afterwards in Section 4 and 5, respectively.

3.1. Boolean Constraint Propagation

Every time a variable is assigned a new truth value, *Boolean Constraint Propagation* (BCP) checks whether this assignment satisfies or dissatisfies a clause. Since all clauses are ANDed in a CNF, a dissatisfied clause would immediately render the current assignment unsatisfiable – a conflict occurs which needs to be resolved (e.g. by backtracking previously made assignments). Moreover, besides pure checking, BCP also allows to deduce (propagate) further assignments: if a clause contains variables that are all set to *false* except one, this variable has to be assigned *true* in order to avoid a conflict. Such clauses are called *unit clauses*.

In its naive implementation BCP would observe every variable in every clause and propagate its value through the formula. This technique is very CPU-intensive and takes a lot of time. In order to improve that, a more sophisticated approach called *Two-Watched-Literal scheme* (TWL scheme) has been introduced in [18]. Here, only two literals of a clause are observed as long as those are unassigned or evaluate to true. If two literals are unassigned, no deductions can be made and the clause should not need to be inspected. If a literal evaluates to true, the clause is satisfied and also does not need to be inspected. This way, a significant number of inspections can be saved – yielding substantial speed-ups.

With respect to memory consumption, it does not make a significant difference whether the naive or the TWL scheme of BCP is applied – in both cases the total amount of clauses needs to be stored. However, one might argue that, in case of the TWL scheme, an additional data-structure storing the watched literals is required. While this is true, this overhead is usually negligible as it usually only requires an array constant in the size of the number of variables which includes pointers to clauses. Hence, from a perspective of developing lightweight SAT solvers, dropping state-of-the-art methods for BCP such as the TWL scheme does not seem beneficial as it would significantly decrease performance while not yielding significant improvements with respect to memory.

3.2. Conflict-Driven Clause Learning

One challenge all SAT solver have is how to deal with conflicts. A conflict occurs if the SAT solver determines a (partial) variable assignment which indeed may satisfy some (or even almost all) clauses, but leaves one clause unsatisfied. In this case, the conflict is resolved by undoing variable assignments – usually through backtracking-like methods. In its original implementation, SAT solvers simply backtrack to the previous level and change to the last-made variable assignment. However, this is often not efficient as usually several levels (and, hence, assignments) can be backtracked at once without pruning any part of the search space that might include satisfying solutions.

Motivated by that, dedicated conflict analysis methods have been introduced [17]. They cannot only analyze how far to backtrack but also offer *Conflict-Driven Clause Learning* (CDCL). Here, information on the conflict is utilized in order to generate a new clause for the SAT instance. This clause basically represents a partial assignment which eventually led to the conflict. Adding such a *conflict clause* could help the SAT solver to enter similar conflicts again – allowing to prune huge parts of the search space without missing any potential satisfying solution.

However, SAT solvers usually run into a substantial number of conflicts and, hence, generate a huge number of conflict clauses. Furthermore, for a single conflict, several different conflict clauses can be generated. Accordingly, further methods have been developed to remove conflict clauses (in particular if they did not help much in search space pruning) as well as to minimize conflict clauses (i.e. to reduce their number of literals). Either way, conflict clauses increase the number of clauses of a SAT instance and, by this, usually require a larger memory consumption. Hence, from a perspective of developing lightweight SAT solvers, conflict-driven clause learning significantly certainly has an impact which shall be evaluated.

3.3. Branching Decision Heuristics

SAT solvers are search algorithms. Despite trivial SAT instances where all variable assignments can be deduced, this requires "guessing" variable assignments (and backtracks as described above if these "guesses" yielded conflicts). In this context, the order in which variables are assigned is crucial to the efficiency of the solver (as different variables may have different impacts e.g. on possible implications for further variable assignments). Because of this, modern SAT solvers employ branching decision heuristics in order to decide which variable should be assigned a value next in case no further implications can be conducted.

There are different branching decision heuristics available. One approach is to choose the variable that satisfies most of the clauses [18], another approach is to store the number of conflicts a literal is part of. The Berkmin [19] and *Variable State Independent Decaying Sum* (VSIDS, [18]) heuristics are known for those conflict-based heuristics.

However, even though these heuristics are statistic-based and additional data is needed to create those statistics, their memory consumption is linear, as only one value is stored for each literal and the number of literals is fixed. Hence, from a perspective of developing lightweight SAT solvers, dropping branching decision heuristics does not seem beneficial as it would significantly decrease performance while not yielding significant improvements with respect to memory.

3.4. Preprocessing

Preprocessing is a technique to simplify a given SAT instance before the actual solving process starts. If a clause can be simplified, e.g. either the number of clauses or the number of literals inside a clause can be decreased, the solver can potentially determine a solution faster. An efficient preprocessing technique that is based on clause subsumption has e.g. been proposed in [22].

Since preprocessing usually decreases the size of a SAT instance, it may also affect memory consumption. Hence, from a perspective of developing lightweight SAT solvers, preprocessing certainly has an impact which shall be evaluated.

4. Considered Configurations

The discussions from above motivate a consideration of conflict-driven clause learning as well as preprocessing for the development of lightweight SAT solvers. Based on that, we considered corresponding configurations of a SAT solver for a comparative experimental analysis. In this section, the respectively considered considerations are briefly reviewed.

4.1. Original

As baseline, we employed MiniSAT in version 2.2.0 and its original configuration with no default values changed. This configuration is meant to provide a reference point in order to compare the results of all other configurations with respect to their run time and memory consumption. This is also a good reference point for memory consumption of SAT solvers in general, as MiniSAT implements a lot of techniques of other SAT solvers, like CDCL [18], VSIDS [18] and Two-Watched-Literal Scheme [18]. Furthermore, this SAT solver is widely used and investigated and won many of SAT competitions already.

In the remainder of this paper, this configuration is denoted by *orig*.

4.2. Preprocessing

This configuration uses the internal preprocessing implementation of MiniSAT (based on the principles of SatELite [22]) in order to try shrinking and simplifying a given SAT instance. On account of the internal implementation, we expect that this configuration consumes a bit more memory as the *original* one, as it stores lists internally to mark clauses and variables. It depends highly on the given instance how much benefit we gain from preprocessing. If the instance cannot be simplified for any reason than there probably would be no significant difference between the original configuration and this one.

In the remainder of this paper, this configuration is denoted by *pre*.

4.3. No Conflict Clause Minimization

MiniSAT usually aims for minimizing conflict clauses generated by conflict analysis. More precisely, each time a new conflict clause is learned, MiniSAT tries to minimize it, e.g. by removing redundant literals. This abbreviates the solving time, as smaller clauses allow for earlier implications (a smaller number of variables need to be assigned in order to imply an assignment or detect a conflict). Furthermore, a minimized clause may also decrease the memory consumption, as small clauses need less memory compared to larger ones. This technique can be deactivated by changing the default value using command line parameters.

Changing this configuration should have an effect on the averaged memory consumption, as larger clauses needs more space than smaller ones. However, this depends on the structure of the learned clauses and if there are redundancies to delete.

In the remainder of this paper, this configuration is denoted by *no ccmin*.

4.4. No Conflict Clauses

Finally, a configuration is considered which avoids the generation of conflict clauses whenever possible.

The last configuration is called no conflict clauses (no cc). As already mentioned MiniSAT implements the CDCL technique which has clause learning as its underlying concept. During the solving process a huge amount of clauses is learned. This configuration removes the learned clauses as soon as they are learned[2]. This reduces the number of learned clauses to a minimum but also significantly affects the run-time performance of the solver. We expect that this configuration will significantly reduce the memory consumption, as significantly less clauses have to be stored.

In the remainder of this paper, this configuration is denoted by *no cc*.

5. Experimental Evaluation

In this section, we summarize the results obtained by the experimental evaluation with the four configurations introduced in the previous section. To this end, we accordingly modified the SAT solver *MiniSAT* [7] and evaluated the resulting configurations using 177 problem instances of different categories provided by the SAT competition 2016[3]. All evaluations have been conducted on an AMD64 machine with 3.4 GHz and 32 GB of main memory. Additionally a timeout of 1h was applied.

As discussed in the previous sections, the amount of additionally added conflict clauses poses the biggest threat to any memory limitations. Hence, a first series of evaluations considered this aspect. Fig. 2 summarizes the obtained results, i.e. the minimal (*min*), maximal (*max*), and average (*avg*) number of originally given clauses as well as the number of conflict clauses generated by the respective configurations (summarized for all 177 problem instances).

The *orig* configuration has the largest amount of conflict clauses on average. Compared to that the *pre* as well as the *no ccmin* configuration have slightly less conflict clauses on average. Nevertheless, it can be concluded that all these three configurations do not significantly differ in this regard. In contrast, the *no ccmin* configuration obviously stands out. Only a negligible amount of conflict clauses is generated.

2. Note that cases exists where conflict clauses cannot immediately by removed; this is however negligible and not further considered here.

3. https://baldur.iti.kit.edu/sat-competition-2016

	given			conflict		
	min	max	avg	min	max	avg
orig	510	35m	703k	2	409k	63k
pre	510	35m	703k	0	442k	61k
no ccmin	510	35m	703k	2	372k	61k
no cc	510	35m	703k	2	319	108

Figure 2. Number of given clauses and conflict clauses

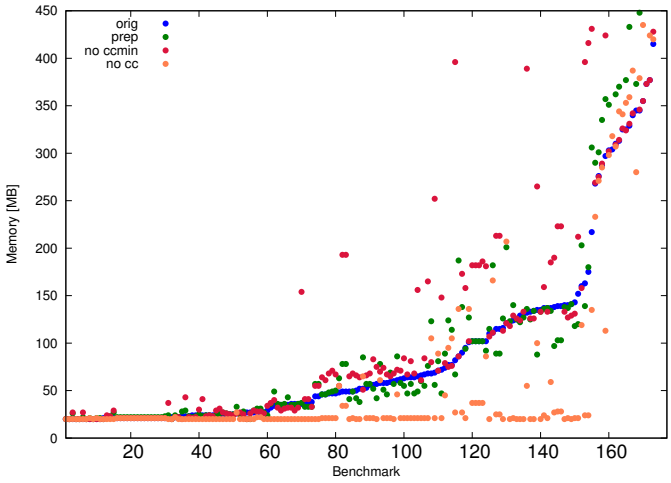

Figure 3. Memory consumption

Now, the question remains how this "translates" into the required memory consumption. Fig. 3 provides a corresponding summary of that. More precisely, the figure shows the required memory consumption (in MB) for all instances and all configurations. Note that instances which could not be solved by a configuration within the given timeout are not displayed in this plot (corresponding performances are evaluated and discussed later in this section).

Again, the configurations *orig*, *pre*, and *no ccmin* have a similar memory consumption – even though there are a few distortions. In contrast, the configuration *no cc* has the smallest memory consumption. This confirms that conflict clauses have a significant impact on memory consumption and that disabling the generation of them could prove beneficial towards the development of lightweight SAT solvers with limited memory requirements.

However, despite this benefit, basically disabling conflict clause learning obviously has an effect on the run-time performance of the solver. Hence, we evaluated this in a second series of evaluations. To this end, we considered how many instances could be solved within the time-limit of one hour by each configuration (summarized in Fig. 4) as well as how much run-time was required for this purpose (summarized in Figure 5; the run-times are provided in minutes).

As can clearly be seen, dropping conflict clauses significantly degrades the performance of SAT solvers. Nevertheless, it can also be observed that such a configuration does not render the solver completely useless. In fact, a significant portion of the instances can still be solved in reasonable time. Additionally considering that significantly less memory is needed, this could provide a good starting point for the development of lightweight SAT solvers.

orig	pre	no ccmin	no cc
75	82	75	57

Figure 4. Number of solved instances out of a total of 177

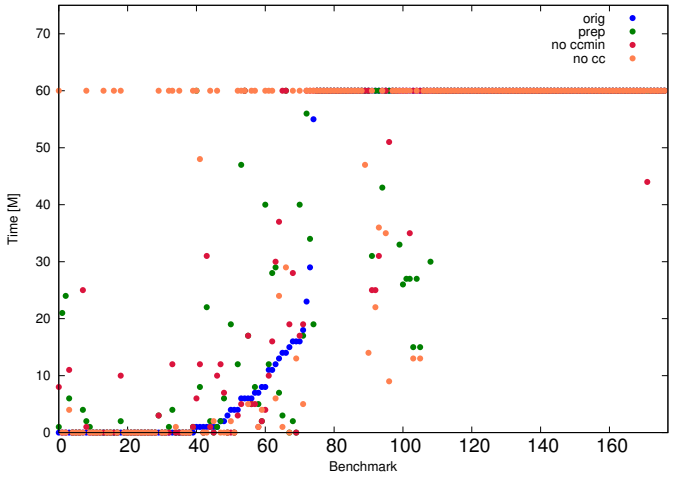

Figure 5. Run-time

6. Conclusion

In this paper, we conducted evaluations towards the development of lightweight SAT solvers – with a particular focus on memory consumption. To this end, we first discussed what core techniques of modern SAT solvers have what impact on the total memory consumption. Based on that, different configurations have been defined and experimentally evaluated. The results clearly showed that disabling the learning of conflict clauses yields the best solution with respect to memory requirements. At the same time, the run-time performance is not completely unacceptable – although a significant amount of instances could not have been solved anymore when conflict clauses are deactivated.

Overall, this motivates to re-think "old" SAT solving strategies when it comes to applications such as self-verification which do not necessarily rely on the best possible run-time performance but may have severe hardware limitations. Future work obviously is focused on developing SAT solvers and verification methods following this direction. Before, however, more detailed evaluations (e.g. considering more sophisticated configurations as well as a broader variety of instances) shall be conducted first.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001.

References

[1] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[2] Rolf Drechsler, Mathias Soeken, and Robert Wille. Formal specification level: Towards verification-driven design based on natural language processing. In *Forum on Specification and Design Languages (FDL)*, pages 53–58, 2012.

[3] Rolf Drechsler. *Formal System Verification State-of the-Art and Future Trends*. Springer, 2017.

[4] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. The mathsat 3 system. In *International Conference on Automated Deduction*, pages 315–321, 2005.

[5] Rolf Drechsler, Martin Fränzle, and Robert Wille. Envisioning self-verification of electronic systems. In *Symposium on Reconfigurable Communication-centric Systems-on-Chip*, pages 1–6, 2015.

[6] Rolf Drechsler, Hoang Minh Le, and Mathias Soeken. Self-verification as the key technology for next generation electronic systems. In *Symposium on Integrated Circuits and System Design (SBCCI)*, pages 15:1–15:4, 2014.

[7] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International Conferences on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

[8] João P. Marques Silva and Thomas Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Design, Automation and Test in Europe (DATE)*, pages 145–149, 1999.

[9] Poul Frederick Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, pages 124–138, 2000.

[10] Koen Claessen. A coverage analysis for safety property lists. In *International Conf. on Formal Methods in CAD*, pages 139–145, 2007.

[11] Stephan Eggersglüß, Robert Wille, and Rolf Drechsler. Improved sat-based ATPG: more constraints, better compaction. In *Computer Aided Verification*, pages 85–90, 2013.

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[13] Robert Wille, Görschwin Fey, Daniel Große, Stephan Eggersglüß, and Rolf Drechsler. SWORD: A SAT like prover using word level information. In *International Conference on Very Large Scale Integration of System-on-Chip*, pages 88–93, 2007.

[14] Horst Samulowitz and Fahiem Bacchus. Using SAT in QBF. In *International Conference on Principles and Practice of Constraint Programming*, pages 578–592, 2005.

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *Symp. on Theory of Computing*, pages 151–158, 1971.

[16] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:506–521, 1960.

[17] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Comp.*, 48(5):506–521, 1999.

[18] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pages 530–535, 2001.

[19] E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *Design, Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[20] Norbert Manthey and Ari Saptawijaya. Towards improving the resource usage of sat-solvers. In *POS-10. Pragmatics of SAT*, pages 28–40, 2010.

[21] Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions SAT solver and its application for reachability analysis. In *International Conf. on Formal Methods in CAD*, pages 275–289, 2004.

[22] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *International Conferences on Theory and Applications of Satisfiability Testing (SAT)*, pages 61–75, 2005.