# SAT-solving in practice

Koen Claessen, Niklas Een, Mary Sheeran and Niklas Sörensson

*Abstract*— **Satisfiability solving, the problem of deciding whether the variables of a propositional formula can be assigned in such a way that the formula evaluates to true, is one of the classic problems in computer science. It is of theoretical interest because it is the canonical NP-complete problem. It is of practical interest because modern SAT-solvers can be used to solve many important and practical problems. In this tutorial paper, we show briefly how such SAT-solvers are implemented, and point to some typical applications of them. Our aim is to provide sufficient information (much of it through the reference list) to kick-start researchers from new fields wishing to apply SAT-solvers to their problems.**

## I. INTRODUCTION

Given a propositional formula, the *Boolean Satisfiability* or *SAT* Problem is to determine whether there exists a variable assignment such that the formula evaluates to true. This is the classic NP-complete problem [1], and has therefore attracted much attention from researchers. In this tutorial paper, we explain SAT and how it can be implemented, and give pointers to how it is used in practice, including many references. Our hope is to attract researchers from new fields to explore applications of SAT. Companion papers explore applications of SAT in a variety of problem domains [2] and to supervisory control [3]. The final paper in this session considers Satisfiability Modulo Theories (SMT) [4], a thriving research direction that opens the way for new applications of SAT at abstraction levels requiring greater expressiveness than that provided by propositional logic.

### A. Why SAT is interesting from a practical point of view

The fact that SAT-solving is hard on average in no way precludes its use in solving the particular SAT instances that arise in real problems. Recent progress in practical applications of SAT has built upon two bases: improved SAT-solving engines and innovative ways to encode real problems in ways that can exploit those engines. Recent SAT-solvers have been developed in a scientific community that has placed great store in practical applicability, and the development of the solvers has in turn spurred work on new ways to exploit such solvers. The resulting positive spiral has led, for instance, to the development of commercial hardware verification tools in which SAT-solvers are a vital component.

K. Claessen, M. Sheeran and N. Sörensson, Department of Computer Science and Engineering, Chalmers University of Technology, SE-41296 Göteborg, Sweden {koen,ms,nik}@chalmers.se

N. Een, Cadence Research Labs, 2150 Shattuck Avenue, 10th Floor Berkeley, CA 94704, USA niklas@cadence.com

### B. State of the art until 1999

Before about 1999, SAT was largely a theoretical subject. Manipulation of boolean functions had long been important in circuit synthesis and verification. However, work in this area concentrated on the use of Binary Decision Diagrams [5]. A glance at the conference proceedings of the first international conference on Formal Methods in Computer Aided Design of Electronic Circuits (1996) [6] makes this very clear.

However, SAT solvers and their applications were also making strides. One of the classic application areas has been planning [7]. In electronic design automation, some typical applications that began to appear in the 1990s were timing analysis [8], test pattern generation [9], [10] and FPGA routing [11]. Motivated by the problem of generating test vectors for combinational and sequential circuits, Kunz and Pradhan introduced the notion of *recursive learning* and demonstrated that it had application not only in test generation, but also in optimization and verification [12]. Indeed, recursive learning could, in the early nineties, be seen as a new approach to the boolean satisifiability problem. Stålmarck's patented method of SAT-solving [13] was used during the 1990s in the formal verification of railway signalling systems [14], [15], [16]; indeed the method is still used commercially for this purpose. The formulas that result from such verification are truly gigantic, but Stålmarck's method copes well with the large but easy SAT instances that result. This work on railway signalling verification, perhaps because of its scalability, was one of the first really successful projects in practical, industrial, formal methods.

Also in an industrial setting, Siemens Corporate Research, already in the 1980s, started a major initiative to explore the potential of formal methods for the company's own products and systems. For circuit design verification, a particularly successful solution called Circuit Verification Environment (CVE) was developed. In the mid 90s the basic methodology and proving machinery of CVE was radically changed from symbolic model checking to a SAT-based approach. The property language Interval Temporal Logic (ITL) was developed, which expresses system behaviour over bounded time intervals. Proving such bounded properties can be mapped to a SAT instance. This new paradigm was quickly adopted in Siemens design flows and became standard practice already in 1996/1997, a few years before academic work on bounded model checking was published. The success of this approach triggered intensive efforts to improve SAT-solving procedures in Siemens, and later at Infineon and OneSpin (the spin-off company that now develops and markets the technology). In parallel, Kunz was developing similar ideas about checking bounded intervals, working with Mentor (a major tool-vendor in Electronic Design Automation). At this time, industry was the driving force in innovation in applied formal methods, and unfortunately the work was not published; but the impact of academic research was soon to increase. Developed in academia, the GRASP [17] and SATO [18] solvers were early examples of solvers intended for use on large-scale problems, and they influenced later developments.

### *C. The SAT revolution*

In 1999, the notion of *Bounded Model Checking (BMC)* was published, and immediately recognised to be of great practical interest [19]. It can be thought of as checking that a property holds not for all possible behaviours of a system but only for the first $n$ steps (from the initial states), for a given fixed $n$. This apparently simple idea has proved extremely effective in practical hardware verification, and is now included in all formally-based tools. It will be considered in more detail in section III.

At around the same time, the high-performance SAT-solver Chaff became available, and was widely used by researchers in applications of SAT [20]. For two of the authors (Een and Sörensson), seeing a presentation about Chaff provided inspiration to build small well-structured, yet high-performance, solvers, culminating in miniSAT [21]. This solver and its associated description can act both as a tutorial and as a starting point for researchers wishing to modify it for their purposes. One of the driving forces in the development of efficient solvers has been the international SAT competition, which has led to the creation of benchmark sets, and also makes the competing solvers available to the research community [22]. Section II presents the basics of SAT-solving.

Independently of the developments in BMC, researchers from Chalmers worked with Stålmarck, who proposed a form of induction for use in complete (rather than bounded) model checking [23], [24]. The method, and also BMC, demands satisfiability checking of many related SAT-instances, which in turn led to an *incremental* version of miniSAT [25]. Section IV presents the basics of *temporal induction*.

## II. THE BASICS OF A MODERN SAT-SOLVER

### *A. Formal Definition of The SAT Problem*

A propositional logic formula is said to be in CNF, *conjunctive normal form*, if it is a conjunction ("and") of disjunctions ("ors") of literals. A literal is either $x$, or its negation $\neg x$, for a boolean variable $x$. The disjunctions are called *clauses*. The satisfiability (SAT) problem is to find an assignment to the boolean variables, such that the CNF formula evaluates to true. An equivalent formulation is to say that *each clause* should have at least one literal that is true under the assignment. Such a clause is then said to be *satisfied*. If there is no assignment satisfying all clauses, the CNF formula is said to be *unsatisfiable*.

Propositional formulas that are not in CNF can be transformed into CNF in a standard way [26], [2], a process that is called *clausification*. Clausification is still an active research area, see for example [27].

### *B. Boolean Constraint Propagation*

During the search for a satisfying assignment, the solver will maintain a *partial assignment*, with some variables assigned to either 0 or 1, and others still unassigned. For a given partial assignment, a clause may find that all its literals except one are false. When this happens, the only way to satisfy that clause is to *fix* (or assign) the variable of the last literal to the appropriate value that makes the literal true. This observation defines a process of deriving new variable assignments from the current partial assignment. It can be implemented very efficiently [20], and when run to saturation (no more assignments can be derived) is referred to as *Boolean Constraint Propagation* (BCP).
**Example.** Consider the following three clauses:

$$\{\text{-a, b}\}\ ,\ \ \{\text{-a, c}\}\ ,\ \ \{\text{-b, -c, d}\}$$

If the partial assignment consists of one fixed variable "a=1", then from the first two clauses, BCP will derive "b=1" and "c=1"; which in turn will imply, through the last clause, "d=1".

### *C. Conflicts, Learning, and Backtracking*

A simple and complete SAT algorithm can be achieved by a standard backtracking search: Pick an unassigned variable, fix it to either 0 or 1 (this is called a *decision*) and recursivly solve the resulting subproblem. If no solution was found, flip the variable to the other value and recurse again. After each branching, the partial assignment is investigated to see if there is an unsatisfied, or *conflicting*, clause (all literals are false). If so, there is no need to branch further (return NOSOLUTION). If on the other hand all variables have been fixed without a conflict, a satisfying assignment has been found.

The procedure can be improved by running BCP after each fixed variable to get all the cheap implications. This procedure, backtracking + BCP, is commonly referred to as DPLL [28], [29], and until the inception of modern SAT solvers was the predominant approach to SAT.

Modern SAT, through a series of improvments to DPLL, has been refined to an algorithm that is sufficently different from the original to deserve its own name. We will refer to it has *Conflict Drivern SAT Solving* (CDSS). It differs from DPLL in three important respects:

1) It is not a recursive procedure. Instead an explicit stack of assignments (referred to as the *trail*) is used for backtracking.
2) It derives and adds new clauses through a learning mechanism. This procedure takes place each time a conflicting clause is detected during the search. The added clauses are redundant in the sense that the resulting problem is logically equivalent, but they assist

the BCP in fixing literals throughout the remainder of the search.

3) Backtracking is no longer restricted to return to the previous decision. The outcome of clause learning is actually two-fold: while producing a learned clause, it also analyses which of the decisions contributed to the conflict. If the $k$ latest decisions were irrelevant for the conflict, the procedure will undo all those $k$ decisions (and their BCP implications) rather than just the last.

Putting it all together in pseudo-code, the modern SAT algorithm is:

```
forever {
    bcp
    if no conflict {
        if no unassigned variable { return SAT }
        make decision
    } else {
        if no decisions were made { return UNSAT }
        analyze conflict
        undo assignments
        add learned clause
    }
}
```

For a more detailed description of this procedure, see references [20], [21]; a list of improvements can be found in [30].

### *D. Making decisions*

The key to making the above algorithm effective is to tie the variable decision heuristic to the clause learning. This is done by increasing the so called *activity* of all the variables present in any of the clauses contributing to a conflict. It will bias the search to stay in the region of the most recent conflicts while ignoring variables that were not involved in those conflicts. In effect, the heurstic forces the solver to exhaust all possible conflicts in a subregion, typically resulting in a set of short, learned clauses that captures, more concisely than the original clauses, the reason why that region of the search space is unsatisfiable. To further focus the search, all activites are *decayed*, in other words periodically multiplied with a number $< 1$, to give higher weight to more recent conflicts. This variable heuristic VSIDS (Variable State Independent Decaying Sum) has emperically been proven to successfully localize large industrial SAT problems and solve them by homing in on the relevant part [31].

### *E. State of the art in SAT*

Improving on the state of the art of SAT has turned out to be a really hard task, as indicated by the slow progress made this century in the development of core SAT algorithms. Since the beginning of the SAT revolution, research effort has been mostly directed towards investigating new applications, possible extensions, and exploring different techniques for encoding particular problems. For instance, looking at the papers accepted to SAT'06 there is barely a single paper that can be considered to attempt to improve on the core algorithms of a DPLL type SAT solver.

Here is a list of noteworthy work that has improved upon core SAT technology since the appearance of Chaff in 2001 [20].

*Conflict clause minimization (2005)* – an improvement in the conflict clause construction algorithm which generally makes conflict clauses stronger, and therefore the proof search more efficient [32].

*Variable-elimination-based preprocessing (2005)* – most practical SAT problems can be greatly simplified before being fed to a SAT solver, reducing their size and complexity, and subsequently reducing solving time; this was first implemented in the tool SatELite [33].

*Improvements to decision heuristics (2002-2007)* – a lot of work has gone into heuristics for how to choose the next variable to branch on in the proof search, and what value it should have first [34], [35], with visible effects on solving efficiency for industrial problems.

*Improvements to restart heuristics (2007)* – modern SAT solvers, in order to avoid getting stuck in one corner of the search space, perform a "restart" every once in a while in the middle of a search; some search parameters are reset and the search restarts at top-level. Special heuristics have been developed for when and how often to do this during search [36], [37].

*Datastructure improvements for clauses (2000-2007)* – new datastructures have been developed for representing clauses; e.g. improving the representation of clauses with two literals [32], and improving memory access patterns in general [38], [39].

The next two sections describe two particular and very common applications of SAT-solvers in formal verification of properties of systems, namely Bounded Model Checking and Temporal Induction.

## III. BOUNDED MODEL CHECKING

The most widespread use of SAT-solvers in industrial property-based system verification today is in *Bounded Model Checking*. Model checking (in general) is an automated technique for checking if a given implementation of a system satisfies a given property, specified in some logic. The answer can either be "yes", in which case the property holds, or "no", in which case the model checker produces a counter example to the property at hand. A counter example is a concrete *path* (i.e. a sequence of consecutive states that starts in the initial state) for which the desired property is false.

Up to the late 1990s, model checking for hardware systems was dominated by *symbolic* model checking methods based on Binary Decision Diagrams (BDDs) [5], a datastructure providing a canonical representation for boolean formulas. BDDs can be used for representing sets of reachable states of a system. A model checker computes the set of all reachable states by a repeated use of boolean variable quantification, an operation that is supported well by the BDD datastructure.

Though rather successful for certain classes of circuits, BDD-based model checkers suffer from a potential *BDD-blowup*, when the size of the BDD datastructures becomes too large to handle in memory. Many techniques have been developed for battling BDD blowup in certain situations, but the actual problem in general remained.

At the end of the 1990s, several research groups were independently trying to alleviate the problem of BDD-blowup by replacing BDDs by other technologies. For example, the model checker FixIt [40], [41], developed by Abdulla, Bjesse and Een, replaced BDDs in the standard model checking algorithms by regular non-canonical propositional formulas. They developed their own cunning variable quantification algorithm, and used a SAT-solver to reason about the formulas. The result was a model checker that complemented the existing BDD-based model checkers. Unfortunately, the variable quantification turned out to be a memory bottle neck, often leading to excessive memory usage.

Bounded Model Checking (BMC) was first presented at the conference on Formal Methods in Computer Aided Design (FMCAD) in 1998. In an unprecedented move, the Chairs of FMCAD'98 permitted the inclusion of an extra talk about BMC. This was followed up by several publications in 1999, including the first paper outlining the idea [19] and one describing its application at Motorola to the verification of a PowerPC processor [42].

A BMC model checker is parametrized by a natural number, a *bound* $n$, and only tries to find counter examples (paths) that consist of no more than $n$ transitions. The answer a BMC model checker thus produces is "no", with a counter example, or "could not find a counter example of length $n$ or smaller". The big gain of this approach is that the BMC model checker does not have to perform variable quantification, the precise thing that was the bottle neck for the model checker FixIt.

Let us look concretely at how a BMC model checker works. For simplicity, we assume that the property we are checking is a so-called *simple safety property*. This means that given a single state of the system, we can decide if this is a good state (the desired property holds) or a bad state (the property does not hold). The original BMC paper presents a symbolic, SAT-based algorithm that can deal with more general properties than these [19].

To model the system under verification and the property in the BMC framework, we are assuming that the state of the system is represented by a finite vector of boolean variables $s$. The safety property is represented by a formula $P(s)$, which is true precisely for the good states, the states where the property holds. To model the system itself, we split it up into two parts: the initial states and the transitions between the states. The set of initial states is modelled as a formula $I(s)$, which is true if and only if $s$ represents an initial state of the system. Finally, the transitions of the system are modelled by a formula $T(s, s')$ which is true if and only if the system can make a transition between the states represented by $s$ and $s'$.

Now, in order to check whether or not there exist counter examples of length $n$ or smaller, we create a sequence of vectors of boolean variables $s_0, s_1, \ldots, s_n$, and build the following formula:

$$\begin{aligned} I(s_0) \quad &\wedge \quad (T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{n-1}, s_n)) \\ &\wedge \quad (\neg P(s_0) \vee \neg P(s_1) \vee \ldots \vee \neg P(s_n)) \end{aligned}$$

The above formula expresses restrictions on the values of the variables in $s_i$; namely that $s_0$ should be an initial state, that there must be a transition from $s_j$ to $s_{j+1}$ for all $j < n$, and that at least one of the states visited must be a bad state. Any satisfying assignment to the above formula therefore represents a counter example to the property. A BMC model checker now simply invokes a SAT-solver to check whether or not that is the case.

The original BMC paper [19] also discusses ways of finding out how large $n$ should be in order to be sure that no counter example of any length can be found. This so-called *diameter* turned out to be expensive to compute in practice.

The resulting method turned out to be very successful for finding bugs. The way this works is that the user can start with small values of $n$, for which the method is very cheap, and successively increase $n$ when no counter examples are found. In this way, very quick feedback is provided about the status of properties, without having to perform a full general model checking procedure. This has led to a paradigm shift in industrial applications of formal methods; instead of concentrating on correctness, the focus has turned more to bug finding.

For a discussion on the benefits of BMC in an industrial setting, we point the reader to a paper by Copty et al. from 2001, discussing experiments with BMC conducted at Intel in Haifa [43]. In the same session at CAV, Bjesse et al. reported on bug-finding in the memory subsystem of an Alpha microprocessor [44]. For some properties, SAT-based BMC reduced verification run-time from days to minutes on real, deep microprocessor bugs. Thus, by mid-2001, the SAT revolution was already well under way.

After the initial publication of the BMC idea, many optimizations and implementation techniques have been developed to improve on the original method. To name a few: tightly integrate the iterations with larger and larger $n$ with an *incremental* SAT-solver in order to be able to reuse work between iterations [25]; use *reparameterization* in order to reduce the size of formulas that are generated for large $n$ [45]. The 2003 journal paper on advances in BMC is a good place to start for those who want to explore the technical ideas behind BMC further [46].

Nowadays, BMC is a key component in any industrial formal verification set-up. Kunz' recent invited talk at FMCAD'07 illustrates this point [47].

## IV. TEMPORAL INDUCTION

We have seen the use of Bounded Model Checking in safety property checking. What if, instead, we wish to prove that a property holds in *all* reachable states of a transition system; without the restriction to searching for counter examples of length $n$ or shorter? One option is, of course, to use the familiar BDD-based (unbounded) symbolic model checking [48], but here we wish, again, to explore the use of SAT. We must therefore find a way to encode the problem directly as SAT instances, without using quantifiers.

Let us first introduce some notation. The symbol $T^k$ stands for a "chain" of $k$ transition relations:

$$T^k(s_0, \cdots, s_k) \;:=\; T(s_0, s_1) \wedge T(s_1, s_2) \wedge \ldots \ldots \wedge T(s_{k-1}, s_k)$$

Now, consider the sequence of formulas $Base_0, Base_1, \ldots$, defined as follows:

$$Base_k \;:=\; I(s_0) \wedge T^k(s_0, \cdots, s_k) \wedge \neg P(s_k)$$

$Base_k$ is satisfiable if there is a path of length $k$ through the transition relation $T$ from an initial state to a bad state, and it is *unsatisfiable (UNSAT)* if there is no such path. Iterating through all $Base_i$, from $i = 0$ and upwards, and checking if they are SAT or UNSAT, roughly corresponds to BMC, and is a bug-finding algorithm. If some $Base_k$ is satisfiable, the satisfying assignment of values to the bit-vectors in the state variables $s_0$ to $s_k$ will give a shortest path from an initial state to a bad state.

The big question is "At what stage can we safely stop and conclude that there can be no such bug?".

One way to provide an answer is by *induction*. Let us call a path that only consists of good states a *good path*. If we can show that each path of length $k$ starting in the initial state is a good path (induction base case), and that each good path of length $k$ starting anywhere can only be extended by transitions that lead to a good state (induction step), then, by induction, all paths of any length starting in the initial state must be good paths.

The base case of the induction for $k$ has already been defined above. In order to define the step case, let us first introduce some notation for asserting that $P$ holds in a sequence of states:

$$P^k(s_0, \cdots, s_k) \;:=\; P(s_0) \wedge P(s_1) \wedge \ldots \wedge P(s_k)$$

We can then define a step case formula as follows:

$$Step_k \;:=\; T^{k+1}(s_0, \cdots, s_{k+1}) \wedge P^k(s_0, \cdots, s_k) \wedge \neg P(s_{k+1})$$

If $Step_k$ is satisfiable for some $k$, then there is a good path of length $k$ that can be extended by going to a bad state.

A simple first induction algorithm can now be defined as follows:

```
i=0
while True do {
    if Sat($Base_i$)
        return False    % counter example
    if Unsat($Step_i$)
        return True
    i=i+1
}
```

If we can find an $i$ for which the base case is satisfiable, then we have a counter example. If we can find an $i$ for which both base case and step case are unsatisfiable, we have shown the property to hold for all reachable states.

The above algortithm is simple, but it is not complete; there are cases where the property holds but where the above algorithm does not terminate, i.e. no induction proof is found. This happens when there are paths of arbitrary length that satisfy the induction step $Step_k$. These paths must necessarily lie outside of the reachable state space. However, since our state space is finite (by assumption), there cannot be paths like this of arbitrary length unless they contain a loop. And since we are really only interested in shortest paths, and thus loop-free paths, we may add to the induction step that all considered paths must be loop-free.

We thus define a formula that expresses loop-freeness ("uniqueness" of states):

$$U^k(s_0, \cdots, s_k) \;:=\; \bigwedge_{0 \le i < j \le k} (s_i \neq s_j)$$

And redefine the step case formula as follows:

$$Step_k \;:=\; T^{k+1}(s_0, \cdots, s_{k+1}) \wedge U^{k+1}(s_0, \cdots, s_{k+1}) \wedge P^k(s_0, \cdots, s_k) \wedge \neg P(s_{k+1})$$

The presented induction algorithm using the above step formula is indeed sound and complete. The soundness and completeness of the algorithm are easily shown, see [23], [25].

However, to make temporal induction work in practice, one must carefully consider exactly what SAT instances to present to the SAT solver, and in particular how to deal with the possibly expensive requirement that the paths considered in the termination check be loop-free. These aspects of the algorithm, its implementation using an incremental SAT solver, and an experimental evaluation of several variants of it, are presented in reference [25].

Another way of improving on this basic algorithm is to *strengthen* the property $P$, i.e. to find a stronger property $P'$ that implies $P$, and prove $P'$ instead. The advantage of doing so is that the $k$ that is needed to prove a stronger property might be much smaller, and thus the formulas that we have to deal with become smaller. An early implementation of this idea (automatically finding an equivalence relation between points in a hardware circuit) was developed for a BDD-based induction-like algorithm by van Eijk [49], and later adapted to SAT-based induction [24]. Recently, some alternative new techniques have been developed for automatically strengthening the induction hypothesis [50], [51].

## V. DISCUSSION AND CONCLUSION

We have very briefly catalogued the simulataneous development of modern SAT-solvers and their applications. A companion paper from this session on SAT provides a tutorial on applications of SAT [2]. Here, we include further references for the interested reader. Another popular model checking technique, based on interpolants, was introduced in 2003 by McMillan [52]. For a survey of SAT-based Formal Verification, see references [53], [54]. Bryant and Kukula's

2002 survey paper on Formal Methods in Functional Verification [55] is a fascinating journey from the early attempts to use inefficient decision procedures up to the period just after the SAT revolution. It ends by cautioning that although the successes in industrial application are encouraging, improvements in speed and capacity of the basic engines are still needed. That is still true today, so that the new developments outlined in section II-E are eagerly awaited by the users of SAT-based tools.

It should also be noted that SAT is often mixed with other technologies (such as BDDs or dynamic (simulation-based) verification) in industrial-strength tools. IBM's SixthSense system for circuit verification is a good example of this development [56]. Bentley's invited talk on microprocessor verification, given at the Computer Aided Verification conference in 2005, not only shows the enormity of the verification problems that we face but also points towards the use of SAT as a means to bridge dynamic (simulation-based) and formal verification [57]. For a good recent overview of the use of all of SAT, Bounded Model Checking and temporal induction, the reader is referred to FMCAD 2007 [58].

As the complexity of the hardware and software systems that we wish to verify grows inexorably, it is increasingly clear that automated reasoning at a level of abstraction above the bit level is needed. This has led to a surge of interest in *Satisfiability Modulo Theories (SMT)* – the combination of SAT with additional theories such as linear arithmetic or bit-vectors. A companion paper in this session on SAT explores this development [4]. The move upwards in level of abstraction is also reflected in increasing research activity in automated reasoning for Quantified Boolean Formulas and even First Order Logic.

We hope that this introduction to SAT-solving in practice will whet the appetite of researchers in new fields, outside our familiar area of Computer Aided Design of Electronic Circuits. In the area of Discrete Event Systems, an early application of both BDD- and SAT-based verification to PLC systems is reported in reference [59]. In this session of WODES, Voronov et al present their work on the use of SAT in supervisory control [3]. We look forward to fruitful collaboration between our research communities.

## VI. Acknowledgments

The authors gratefully acknowledge Joao Marques-Silva for useful comments on earlier drafts of this paper, and Bengt Lennartson and Martin Fabian from the Signals and Systems Department at Chalmers for taking the initiative to include a special session on SAT at the International Workshop on Discrete Event Systems, May, 2008.

## References

[1] S. Cook, "The complexity of theorem-proving procedures," in *Proc. 3rd ACM Symp. on the Theory of Computing*. ACM Press, 1971.

[2] J. Marques-Silva, "Practical Applications of Boolean Satisfiability," in *Int. Workshop on Discrete Event Systems, WODES08 (this volume)*, 2008.

[3] A. Voronov and K. Åkesson, "Supervisory Control using Satisfiability Solvers," in *Int. Workshop on Discrete Event Systems, WODES08 (this volume)*, 2008.

[4] A. Cimatti, "Beyond Boolean SAT: Satisfiability Modulo Theories," in *Int. Workshop on Discrete Event Systems, WODES08 (this volume)*, 2008.

[5] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comp.*, vol. c-35:8, 1986.

[6] M. Srivas and A. Camilleri, Eds., *Int. Conf. on Formal Methods in Computer-Aided Design*, ser. LNCS. Springer, 1996, vol. 1146.

[7] H. Kautz and B. Selman, "Planning as Satisfiability," in *Proceedings ECAI-92*. John Wiley & Sons, Inc, 1992.

[8] L. Guerra e Silva, J. P. Marques-Silva, L. M. Silveira, and K. A. Sakallah, "Timing Analysis Using Propositional Satisfiability," in *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE Press, 1998.

[9] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 6–22, 1992.

[10] J. Marques-Silva and K. Sakallah, "Robust search algorithms for test pattern generation," in *Fault-Tolerant Computing Symposium (FTCS)*. IEEE Computer Society, 1997.

[11] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar, "Satisfiability-Based Layout Revisited: Detailed Routing of Complex FPGAs Via Search-Based Boolean SAT," in *International Symposium on Field Programmable Gate Arrays*. ACM Press, 1999.

[12] W. Kunz and D. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization," *IEEE Trans. CAD*, vol. 13, no. 9, 1994.

[13] M. Sheeran and G. Stålmarck, "A Tutorial on Stålmarck's Proof Procedure for Propositional Logic," *Form. Methods Syst. Des.*, vol. 16, no. 1, pp. 23–58, 2000.

[14] G. Stålmarck, "A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula," 1989, swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995).

[15] M. Säflund, "Modelling and formally verifying systems and software in industrial applications," in *second Int. Conf. on Reliability, Maintainability and Safety (ICRMS '94)*. International Academic Publishers, 1994.

[16] A. Borälv and G. Stålmarck, "Prover Technology in Railways," in *Industrial-Strength Formal Methods*. Academic Press, 1998.

[17] J. Marques-Silva and K. Sakallah, "GRASP: A New Search Algorithm for Satisfiability," in *International Conference on Computer-Aided Design (ICCAD'96)*. IEEE Press, 1996, pp. 220–227.

[18] H. Zhang, "Sato: An efficient propositional prover," in *International Conference on Automated Deduction, CADE-14*, ser. LNCS, vol. 1249. Springer, 1997, pp. 272–275.

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, ser. LNCS, vol. 1579. Springer, 1999.

[20] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Design Automation Conf. (DAC'01)*. IEEE Press, 2001.

[21] N. Een and N. Sörensson, ""An Extensible SAT Solver"," in *Proc. of Theory and Applications of Satisfiability Testing (SAT'03)*, ser. LNCS, vol. 2919. Springer, 2003.

[22] "The International SAT Competition web page." [Online]. Available: http://www.satcompetition.org/

[23] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design (FMCAD'00)*, ser. LNCS, vol. 1954. Springer, 2000, pp. 108–125.

[24] P. Bjesse and K. Claessen, "SAT-Based Verification without State Space Traversal," in *Formal Methods in Computer-Aided Design (FMCAD'00)*, ser. LNCS, vol. 1954. Springer, 2000, pp. 372–389.

[25] N. Een and N. Sörensson, "Temporal Induction by Incremental SAT Solving," *First Int. Workshop on Bounded Model Checking, ENTCS*, vol. 89, no. 4, 2003.

[26] G. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in Constr. Math. and Math. Logic*, 1968.

[27] N. Een, A. Mishchenko, and N. Sörensson, "Applying logic synthesis for speeding up SAT," in *Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'07)*, ser. LNCS, vol. 4501. Springer, 2007.

[28] M. Davis and H. Putnam, "A computing procedure for quantification

theory," *Journal of the ACM*, vol. 7, pp. 201–215, 1960, reprinted in [60].

[29] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962, reprinted in [60].

[30] N. Een, "Practical SAT, Invited Tutorial at Int. Conf. on Formal Methods in Computer Aided Design," 2007, slides available at http://minisat.se/Papers.html.

[31] O. Shacham and E. Zarpas, "Tuning the VSIDS Decision Heuristic for Bounded Model Checking," in *Proc. Fourth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2003)*. IEEE Computer Society, 2003.

[32] N. Een and N. Sörensson, "MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization, System description for the SAT competition," 2005, available at http://minisat.se/Papers.html.

[33] N. Een and A. Biere, "Effective Preprocessing in SAT through Variable and Clause Elimination," in $8^{th}$ *Int. Conf. Theory and Applications of Satisfiability Testing (SAT'05)*, ser. LNCS, vol. 3569. Springer, 2005.

[34] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT solver," in *Design and Test in Europe (DATE'02)*. IEEE Computer Society, 2002.

[35] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers." in *Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'07)*, ser. LNCS, J. Marques-Silva and K. A. Sakallah, Eds., vol. 4501. Springer, 2007, pp. 294–299.

[36] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal Speedup of Las Vegas Algorithms," in *Israel Symposium on Theory of Computing Systems*, 1993, pp. 128–133. [Online]. Available: citeseer.ist.psu.edu/article/luby93optimal.html

[37] J. Huang, "The effect of restarts on the efficiency of clause learning," in *IJCAI*, 2007, pp. 2318–2323.

[38] A. Biere, "Short History on SAT Solver Technology and What is Next?, invited talk," in *Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'07)*, 2007, slides available at http://fmv.jku.at/biere/talks/Biere-SAT07-talk.pdf.

[39] ——, "PicoSAT Version 535, System description for the SAT competition," 2007, available at http://www.satcompetition.org/2007/picosat.pdf.

[40] N. Een, "Symbolic Reachability Analysis Based on SAT-solvers," Master's thesis, Department of Computer Systems, Uppsala University, Sweden, 1999.

[41] P. A. Abdulla, P. Bjesse, and N. Een, "Symbolic Reachability A.nalysis based on SAT-solvers," in *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, ser. LNCS, vol. 1785. Springer, 2000.

[42] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs," in *Int. Conf. on Computer Aided Verification (CAV'99)*, ser. LNCS, vol. 1633. Springer, 1999.

[43] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi, "Benefits of bounded model checking at an industrial setting," in *Int. Conf. on Computer Aided Verification (CAV'01)*, ser. LNCS, vol. 2102. Springer, 2001.

[44] P. Bjesse, T. Leonard, and A. Mokkedem, "Finding bugs in an alpha microprocessor using satisfiability solvers," in *Int. Conf. on Computer Aided Verification (CAV'01)*, ser. LNCS, vol. 2102. Springer, 2001.

[45] P. Chauhan, D. Kroening, and E. Clarke, "A SAT-Based Algorithm for Reparameterization in Symbolic Simulation," in *Design Automation Conference (DAC'04)*. ACM, 2004.

[46] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, 2003.

[47] W. Kunz, "Formal Verification of Systems-on-Chip – Industrial Experiences and Scientific Perspectives," 2007, invited Talk at Int. Conf. on Formal Methods in Computer Aided Design, slides available at http://www.fmcad.org/2007 (under Advance Program).

[48] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[49] C. van Eijk, "Sequential equivalence checking without state space traversal," in *Int. Conf. on Design Automation and Test in Europe (DATE'98)*. IEEE Computer Society, 1998.

[50] M. Case, A. Mishchenko, and R. Brayton, "Automated extraction of inductive invariants to aid model checking," in *Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'07)*. IEEE Press, 2007.

[51] A. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'07)*. IEEE Press, 2007.

[52] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Int. Conf. on Computer Aided Verification (CAV'03)*, ser. LNCS, vol. 2725. Springer, 2003.

[53] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment," in *Conference on Correct Hardware Design and Verification Methods (CHARME'05))*, ser. LNCS, vol. 3725. Springer, 2005.

[54] M. Prasad, A. Biere, and A. Gupta, "A Survey of Recent Advances in SAT-based Formal Verification," *Intl. Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 2, 2005.

[55] R. E. Bryant and J. H. Kukula, "Formal methods for functional verification," in *Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*. ACM, 2002.

[56] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations," in *International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, ser. LNCS, vol. 3312. Springer, 2004.

[57] B. Bentley, "Validating a Modern Microprocessor, invited talk," in *17th International Conference on Computer Aided Verification (CAV'05)*, ser. LNS, vol. 3576. Springer, 2005, slides available at http://www.cav2005.inf.ed.ac.uk/Fbentley_CAV_07_08_2005.ppt.

[58] J. Baumgartner and M. Sheeran, Eds., *Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'07)*. IEEE Press, 2007.

[59] K. Loeis, M. Bani Younis, and G. Frey, "Application of Symbolic and Bounded Model Checking to the Verification of Logic Control Systems," in *10th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. IEEE, 2005.

[60] J. Siekman and G. Wrightson, Eds., *Automation of Reasoning*. Springer, New York, 1983.