Thesis for the Degree of Doctor of Philosophy

SAT Based Model Checking

Niklas Eén

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2005

ISBN 91-7291-555-2
ISSN 0346-718X
Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie Nr 2237

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Printed at Chalmers, Göteborg, Sweden, 2005

Abstract

This Thesis is a study of automatic reasoning about finite state machines (FSMs). Two techniques used in hardware verification are presented. In both, the verification is carried out by a translation of the problem into propositional logic. Satisfiability and validity of propositional formulas are decided by the use of a SAT solver. For this reason, the fundamental techniques of a modern SAT solver are also presented.

The material belongs in the research field of *symbolic model checking* (SMC). The field comprises different methods of verifying (temporal) properties of finite systems, such as hardware designs, with a high degree of automation. The scope of current methods, and the level of automation, is such that SMC is frequently applied in industry.

One way to prove a property of a system is to explicitly enumerate all reachable states, and check the property for each one. This is known as *explicit state model checking*. SMC, on the other hand, works by reasoning *symbolically* about the system, using a compact representation of sets of states. There is no direct relation between the size of a set and its representation, which gives SMC the potential of handling systems with very large state spaces.

Conventional SMC is carried out by using *binary decision diagrams* (BDDs), a canonical representation of boolean functions (i.e. subsets of $Bool^n$), to compute and represent subsets of the state space. Although algorithms based on BDDs have been successful in many applications, there are limitations that cannot easily be overcome. In this Thesis, alternative approaches based on SAT are explored, in the hope of removing some of those limitations.

The first paper in the Thesis shows how *reachability analysis* (computing a representation of the reachable states) can be performed in much the same way as for BDDs, using a non-canonical representation of boolean functions. The method includes a translation from *quantified boolean formulas* (QBFs) to propositional formulas, and the use of a SAT solver for termination checks.

The second paper shows how safety properties can be proven by means of *temporal induction* (also known as *k-induction*). Several improvements are made to previous techniques, in particular by the introduction and novel use of an incremental SAT solver. The performance gain is documented by thorough testing.

The third paper documents in detail how a modern SAT solver is constructed and suggests some extensions. It shows how arbitrary boolean constraints can be added to a SAT solver, and also implements an incremental SAT interface.

The fourth and final paper proposes a solution to the important problem of generating good SAT encodings of domain specific problems. The approach is general in the sense that it is not limited to the typical translation from netlists, often used in hardware verification.

This Thesis collects together four articles, where my contribution were as follows:

- ***Symbolic Reachability Analysis based on SAT-Solvers***, written together with Parosh Abdulla and Per Bjesse, published in the *"Proceedings of the 6^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems, volume 1785 of LNCS, Springer-Verlag, 2000"* (TACAS'00).

  I wrote the tool on which the paper is based, as well as performed two of the three experiments. Parosh Abdulla took the initiative for the paper by producing the first draft. All parts were later rewritten by Per Bjesse and me. In particular I wrote section 4, 5 and parts of section 7.

- ***Temporal Induction by Incremental SAT Solving***, written together with Niklas Sörensson, published in the *"Proceedings of the 1^{st} International Workshop on Bounded Model Checking, 2003"* (BMC'03), affiliated with CAV'03. The article is also published in *"Issue 4, Volume 89 of Electronic Notes in Theoretical Computer Science (ENTCS)"*.

  The first prototype tool demonstrating the promising effect of incremental SAT in temporal induction was written by Niklas Sörensson. The final tool used in the benchmarking was produced by me. Section 1 was written by Sörensson, sections 2 to 5 by me, section 6 and 7 jointly. The experiments were performed by me.

- ***An Extensible SAT-solver***, joint work with Niklas Sörensson, published in *"Proceedings of the 6^{th} International Conference on Theory and Applications of Satisfiability Testing, 2003"* (SAT'03).

  The solver presented in the article was developed in equal parts by me and Sörensson, but the article was written in its entirety by me.

- ***Improved Subsumption Techniques for Variable Elimination in SAT***, joint work with Armin Biere, submitted for publication.

  The presentation was based on my tool SATELITE. I wrote sections 5 and 6, and carried out the experiments.

Acknowledgements

First and foremost, I would like to thank my supervisor Mary Sheeran. Without her excellent advice and her continuous support, this Thesis would not exist. Her knowledge of the field, and understanding of the research community as a whole, is admirable. I will dearly miss the regular pieces of wisdom she bestows on her students.

I would also like to express my gratitude towards my fellow Ph.D student Niklas Sörensson, with whom I had the pleasure to collaborate on a large part of this Thesis. He is a clear thinker with truly original ideas. The views, ideas, and painful criticism he expressed during our substantial coffee breaks have greatly improved the result of my work. More importantly, without his friendship, I would not have enjoyed my Ph.D nearly as much.

Due credit should also go to Satnam Singh, Koen Claessen, Jörgen Gustavsson, Per Bjesse, and Armin Biere; all of whom have contributed to the success of my Ph.D, one way or another. I also wish to thank those dear to me, my family and my friends. And most importantly I thank my wife, Luige, for her unwavering love and support throughout the long process.

Thank you all for making my years at Chalmers so gratifying!

Niklas, December 2004

Contents

Introduction

When I started my PhD in 1999, symbolic model checking was more or less synonymous with reachability analysis using BDDs. Since the beginning of the 90s, the data structure *binary decision diagrams* had been by far the most successful approach to fully automatic proof procedures for finite boolean systems, such as hardware designs. The succinct representation of boolean functions offered by BDDs, led to the development of efficient symbolic methods to explore the reachable state space of a system. By providing temporal properties, a designer could then algorithmically *check* desired behaviors of the system (or *model*) under design.

An important mile-stone in the development of BDD based model checking was the now well-cited paper by Randal Bryant from 1986 [Bry86], where reduced, ordered BDDs were introduced and proven to be canonical. The full potential of this data structure was revealed in an equally important mile-stone: the first SMV, symbolic model verifier, written by Ken McMillan in 1993 [McM93]. Although some work had preceded SMV [CMB90], McMillan's work was the first comprehensive and efficient implementation of symbolic property checking of finite state systems, containing a rich description language for modeling and specification. The tool showed the potential of algorithmic (i.e. without manual intervention) property checking, and set the standard against which later tools were compared—indeed it spawned a handful of facsimile SMVs from other research groups.

Model checking is not the only automatic technique for increasing confidence in a hardware design. Historically, *simulation* has been the principle vehicle for uncovering bugs. The hope is, that by stimulating the design with a large number of input vectors, errors will manifest themselves. Although simulation for large designs cannot be exhaustive, it is still a vital part of any design flow. The first *formal* technique regularly applied by industry was *equivalence checking*, where an untrusted design is exhaustively compared for functional equivalence towards a trusted, "golden" design (see for instance [MW+88,Mat96]). This progression, too, was sparked by the advent of BDDs, as were *symbolic simulation* and *symbolic trajectory evaluation*. STE uses symbolic simulation to perform a restricted form of model checking, where the temporal properties are limited in form [AJS98,BLM01].

Although most symbolic model checkers, SMCs, accept a temporal formula as specification, so-called *safety properties* are widely recognized as the most important type of specification. A safety property states something that should be true of the system at all times. States where the property fails are called *bad states*, and the goal of the model checker is to prove that there is no path going from a legal *initial state* to a bad state—or alternatively provide such a path, called a *counter-example* to the property.

A standard BDD based model checker achieves this goal by repeatedly applying *image computation* to the initial states. The system is viewed as a *transition relation*, relating the current state and the possible next states. In this thesis it will be assumed that the systems are gate-level representations of hardware circuits. A state is identified with the current value of the memory elements ("flip-flops"); the *inputs* are, at each clock-cycle, allowed to assume any value non-deterministically. Computing the image is the process of finding all states reachable in one transition from a set of current states. Once the repeated application of image computation reaches a fixed point, all states that the system can enter are captured in a single BDD. Any safety property can quickly be determined from this BDD.

Due to symmetry, the procedure can be carried out in the opposite direction, with the roles of initial and bad states interchanged. The backward image computation is referred to as *pre-image* computation, and the fixed point thus attained contains all states that can reach a bad state, which is equally useful for checking safety properties.

The attention brought to BDD based model checking by the success of SMV resulted in a surge of papers exploring improvements and related techniques. Particularly successful outcomes of this body of research include dynamic variable reordering of BDDs [Rud93], partitioned representation of the transition relation and improved quantification scheduling in image computation [HKB96], and counter-example based abstraction refinement techniques [CG+03].

However, by 1999, the boundaries for what could be achieved by BDDs had to a large extent been established. Contemporary publications often improved only on some small detail of previous work, or presented an idea that yielded only a small speedup on a handful of examples. At the time, I had just made the final touches to my Masters Thesis, during which I was fortunate to have access to the commercial SAT solver of Prover Technology [BS99]. This was before SAT solvers were popularized in hardware verification, but my two advisors Parosh Abdulla and Gunnar Stålmarck believed that somehow a SAT solver should be of use in the verification of sequential circuits. As a result, the project of my Masters Thesis was left very open. I made the beliefs of my supervisors concrete by constructing an image computation on propositional formulas that were represented with sharing of identical subformulas. Using this procedure, the basic reachability algorithm could be implemented on a non-canonical representation, rather than canonical BDDs (for which the representation of the transition relation can blow-up exponentially in the worst case). The termination checks—whether we have computed a fixed point, or found a counter-example—were facilitated by the SAT solver.

In short, a SAT solver, or *satisfiability* solver, provides an assignment to the variables of a boolean expression, such that the expression evaluates to true, or alternatively proves that there is no such assignment. Many questions about circuits can be formulated as SAT problems by a straight-forward, linear translation.

The work on my Masters resulted in the tool FixIt, which contained forward and backward reachability analysis, as well as a quantifier free bug-hunting algorithm I called *unrolling*—now known as *bounded model checking* for safety-properties [BCZ99]. BMC, which started the trend of using SAT in hardware verification, was introduced at almost exactly the same time as my Masters Thesis was finished, and the benchmarks on the unroll-technique were among the first results hinting at the promise of BMC for finding counter-examples.

During my Masters project, Per Bjesse, who had also worked on SAT based formal verification, took an interest in my work. When I became a fellow Ph.D. student at Chalmers, we decided to continue the development of FixIt and to publish the results at TACAS'2000. In that paper [*Paper 1*], we show that there are systems where BDD based model checking fails, but where my simple SAT based reachability analysis succeeds. The paper was very well received by the community—clearly looking for alternatives to the stagnating BDD research—and was awarded "the best paper presented at any of the ETAPS'2000 conferences". [1]

There were several ways to continue on the FixIt project. For instance, more BDD based verification could be adapted to the SAT based framework and evaluated. Per did some work to add support for full CTL with counter-example generation, and later adopted STE to SAT within the FixIt framework. As far as other researchers are concerned, a number of ideas relating to non-canonical image computation were later published [GYA00,McM02,GGA04].

My first instinct was to try to improve the raw performance of FixIt. By looking at the formulas produced by the image computation, one could see that they were highly redundant. It should be possible, I believed, to apply some minimization technique to remove the redundancy, and reduce the size of the formulas produced. Smaller formulas, in general, mean faster SAT and faster image computation. In the year 2000, not much work on formula minimization applicable to my problem had been published. The closest related area was *hardware synthesis*, which deals with the problem of creating a circuit from a boolean function. Unfortunately, typical synthesis algorithms run for hours or minutes, rather than seconds or fractions of a second.

Still, to investigate the potential of minimization techniques, I implemented a heavy-weight ZBDD based synthesis algorithm by Shin-Ichi Minato [Min96]. It works by flattening a formula (or circuit) into disjunctive normal form, then building a new multi-level circuit from that representation. Applying the technique to the various formulas produced by FixIt proved, as expected, to be very time consuming; but also somewhat surprisingly, it tended to give harder

[1] Undoubtedly it was a great start, having my first paper selected "best paper" from more than a hundred competitors, just months after starting my Ph.D. Now I knew for sure, that from this point, things can only get worse.

(but smaller) SAT problems. Apparently, synthesizing completely new formulas, without preserving structure at all, was a bad idea.

My next try was more incremental in nature; inspired by the work of [KSM94] in the *automatic test pattern generation* (ATPG) field. The idea was to minimize a formula by a number of small rewrite-steps, which would preserve some of the structure of the original formula, and allow for better control over runtime. I devised a contextual minimization algorithm using *saturation* [SS98]. Saturation can quickly find equivalent subformulas in a bigger formula F. For each conjunction "$A \wedge B$" inside F, the algorithm assumes "A" and detects equivalences in "B" under this assumption. Larger formulas are then substituted for equivalent smaller ones, essentially using "$\neg A$" as a "don't care" set. The approach had a significant impact on the formula size, regarded as a *tree*, but the actual *graph* representation was often unaffected. The algorithm made no use of information on shared nodes, and often it would transform a graph with a high degree of sharing to a similar sized graph with almost no sharing. Interesting as that might be, it did not improve the performance of FixIt significantly.

I had to conclude that the problem of minimizing formulas for the benefit of SAT (and image computation) was a hard task, and that a better understanding of the underlying SAT procedure seemed necessary. In the spring 2000 I started writing my first SAT solver, based on the material published by Prover Technology. My hopes were that it would not only give me a better understanding of SAT, but also better control and a closer integration with my application.

Concurrently, I began working on a *formula manipulation* framework, ForMani, to facilitate the implementation of the ten (or so) other ideas I had about formula minimization. I wanted to provide the YACC of model checking, where the user gives a data type specification—which could be rich enough to encompass first order logic formulas—and code is automatically produced to represent and manipulate the data type with all the clever low-level bit tricks you could imagine. Higher-order traversal primitives, such as "fold" and "for-each", should also be generated for the data type.

In hardware verification, it is notoriously difficult to evaluate a prototype tool, as often the performance on benchmarks in terms of speed is the key indicator of the tool's benefits. Thus, the motivating force behind ForMani was my need to build, essentially, a miniature "BDD package" for each formula representation I was considering. Each implementation should have the performance of a mature and optimized package. The goal of the ForMani system was to allow such code to be written quickly and succinctly from scratch.

In the autumn of 2000, due to a change of policy on Prover's behalf, my license for their SAT solver was revoked. My own solver was not yet up to speed, which left me without a good SAT solver. In concert with my supervisor Mary Sheeran, I decided to make ForMani the subject of my Licentiate Thesis.[2] FixIt was abandoned, to no little extent because Prover owned some of the source code and adamantly refused any kind of distribution, not even a crippled binary for research purposes.

Although performance-wise, the goals of ForMani were reached (by extensive use of C++ template meta programming), the functionality did not mature

[2] Ph.L. is a Swedish degree, corresponding roughly to half a Ph.D.

enough for me to make the framework public; not before the release of ZCHAFF [MZ01] dragged me back to the world of SAT solvers. In the summer of 2002, I met Armin Biere at CAV, and he showed me how simple it was to write a ZCHAFF like solver. I went on to write SATZOO, which won two categories in the SAT 2003 competition. By having a firm understanding (and source code) of a SAT solver, I had gathered the momentum needed for research involving a closer integration between SAT and model checking. A consequence of this was that I changed the subject of my Licentiate Thesis, and FORMANI remained an internal tool that I used during my research.

The Formal Methods group at Chalmers has a particular interest in symbolic model checking by means of *induction*. During a presentation of k-step induction (where a property is proven to hold at a clock-cycle, given that it holds in the previous k cycles) by Koen Claessen at the departmental Winter Meeting in 1998, Gunnar Stålmarck suggested a way to make the procedure complete. By adding constraints on the path, requiring each state to be unique, k would be bounded by the recurrence diameter of the system. At FMCAD'2000, Mary Sheeran presented this complete induction method [SSS00], including some stronger versions of the path constraints, to a somewhat disbelieving audience.[3] Still two years later, the proposed method had gone largely unnoticed, and the consensus was that induction needs an inductive invariant, which must be provided manually.

My colleague Niklas Sörensson, who had worked with me on developing SAT solvers, had taken up this work and started on an *incremental* version of the k-induction algorithm. Niklas had earlier worked on automatic ways of strengthening induction by methods based on the work of C. van Eijk [Eijk98], and firmly believed that induction, even without manual invariants provided, was a viable alternative to reachability based symbolic model checking. From my work on formula minimization, I was also aware of the successful attempts by Dominik Stoffel and Wolfgang Kunz to use induction for sequential equivalence checking [SK97]. In the spring of 2003, Niklas and I produced a tool TIP (Temporal Induction Prover) where induction proofs of successively larger k were attempted in an incremental fashion, resulting in a significant speed-up [*Paper 2*]. A great deal of work in preparing the paper was spent on collecting benchmarks from available sources on the net and packaging them in a uniform format (see section *Scientific Beliefs*).

After publishing the paper on TIP, I was invited to write a paper for the SAT 2003 proceedings as a result of SATZOOs performance in the competition. I decided to use that opportunity to help others appreciate the simplicity and elegance of a ZCHAFF style solver. Drawing from my experience of writing SATZOO, I decided to write a new SAT solver with clarity of exposition in mind. The result, MINISAT, lacked some of the less important features of SATZOO, but had two additional features, much influenced by discussions with Niklas Sörensson: an incremental SAT interface, and support for adding general boolean constraints, not just *clauses*. More importantly, the implementation was very concise, while still retaining competitive performance.

[3] During the questionary following the talk, she was confronted with "is induction really useful for anything?".

After three days of hectic implementation, I started writing the article, that would essentially be a tutorial for MiniSat. The organizers of SAT 2004 had been kind enough to notify me about the invitation a full ten days before the submission date. Finishing barely in time, I further learnt that the rules had been changed, and that my paper would not be treated as *invited* at all, but pass through the normal peer-review process.

As the paper was never intended to be reviewed for its scientific contributions, I was worried. My aim had been to provide a researcher using SAT with leverage to make his own solver and take advantage of the closer integration between SAT and the problem at hand; or, alternatively, to modify the now well-documented source code of MiniSat. Two of the reviewers tore the paper to shreds. They could not find one good thing about it, or indeed understand why it was even written. Luckily, the other two reviewers welcomed the publication as a much needed, long missing paper in the community. It was accepted [*Paper 3*].

During my work on Tip, I confronted the problem of translating netlists to *conjunctive normal form* (CNF), which is the standard internal representation in most contemporary SAT solvers. Until then, the problem of generating good CNFs had mostly been ignored. My interest was motivated by the poor translation of Muxes[4] when generating CNFs from *and-inverter graphs* [MS04]. Although the particular problem of Muxes could easily be solved, I was looking for a more generic solution, preferably inside the SAT solver. If SAT problems stated in CNF could in a generic fashion be improved after the translation, it would save the user from the obligation to produce good CNFs for his problem domain (which might not involve netlists at all). As an extra benefit, the solver would be able to apply the improvement technique repeatedly during the solving process, and thereby make use of facts derived by the learning mechanism of the SAT solver.

My work on this constitutes the final paper of this Thesis [*Paper 4*]. It turned out that my approach was very close to that of Armin Biere's in Quantor [Bier04], and when we realized we were both aiming for a submission to the same conference, we decided to write the paper jointly. The alternative approach of improving the netlist translation scheme to produce good CNFs directly has been recently investigated by Miroslav Velev [Vel04].

At the beginning of February 2005, this Thesis was[5] defended successfully and on time. Indeed, there is a first time for everything.

What is next?

What does the future hold for the micro-universe of symbolic model checking? The 90s was clearly the decade of BDDs. The first decade of this millennium seems to be given to SAT. Although my crystal ball does not reveal the technology to follow SAT, I believe that within the next five years, we will see more

[4] A Mux, or a *multiplexer*, is a three-input, one-output if-then-else gate, logically equivalent to $(s \wedge x) \vee (\neg s \wedge y)$, where s is the selector signal.

[5] hopefully

cross-fertilization between SAT and BDD research; not just by using the techniques side-by-side, as has been done already [GYA00,MA03], but also at a more fundamental level. The major difference between BDDs and SAT is, as I perceive it:

- The BDD approach is about manipulating formulas. The canonicity is the precondition for the efficient manipulation, and no equally efficient algorithm on non-canonical representations of boolean functions has been found. The strong constraints on BDDs mean that almost any algorithmically relevant information can be extracted in constant or linear time.
- SAT, on the other hand, is about having a static structure, and avoiding manipulation altogether. The rationale is that BDDs contain too much information (building a BDD solves a #P problem, believed to be harder than the complexity class NP in which SAT belongs). The lack of structure means that to extract necessary information for a SAT based model checking algorithm, expensive *search* is required.

There is a spectrum between these two extremes. For instance, conjunctive (or disjunctive) partitioning of BDDs relaxes the canonicity constraint at the cost of fast information extraction. A more fundamental merging of BDDs and SAT, however, is to jettison the rigid structure of SAT and dynamically rewrite the underlying CNF (or circuit), as part of the SAT solving.[6] There have been some first successful attempts in this direction, mostly applying the rewriting before the search [KGP01,MJB05] (and *Paper 4* of this Thesis). Modifying the circuit will affect what variables are introduced for internal wires in the CNF, and hence modify the search space for the SAT solver too. This was what I tried to achieve with my minimization efforts in FixIt: to combine the term-rewriting aspect of BDDs with SAT's procedure of systematically (and repeatedly) dismissing chunks of the search space. I believe more ideas in this direction will develop in the future.

Scientific Beliefs

I believe that publicly funded research should adhere to the important scientific principle:

Any result published should be reproducible.

If a result cannot be verified independently by other researchers, its value is reduced to hearsay. In the field of symbolic model checking, this principle has the following implications:

[6] Most SAT solvers currently operate on formulas in CNF. However, this representation is closely related to the circuit from which the SAT problem originated. A typical translation from a circuit netlist, introduces an extra SAT variable for each wire of the circuit in a one-to-one fashion. By keeping some meta information about what wires are primary inputs, and what wires are internal points, the difference between a CNF based SAT solver and a so called *circuit* SAT solver [GZAM02] is really just a matter of low-level representation. A propositional formula, represented with sharing of common subformulas, is just a circuit with a single output.

- Benchmarks used in publication must be open and accessible to fellow researchers.
- Engineering tricks should be revealed, either by releasing the source code or by allotting space in publications to discuss them. The small details are often half the result in our research field.

The current lack of good, publicly available benchmarks has created a deplorable situation where different research teams cannot compare their results. Whenever industrially relevant benchmarks are used in publication, they are kept hidden and listed as "Design 1", "Design 2" etc. This is bad for the research community, and greatly obstructs the progress of academic research. In fact, it is a valid question whether applied hardware verification should be researched in academia at all. Given the economic interests, which is the root of the problem, maybe industry is the appropriate place for research?

Researchers working on non-public benchmarks should, in my opinion, observe the following guidelines:

- Apply proper benchmarking methodology. During my Ph.D. I have spent weeks implementing results from papers—only to find that, in general, the proposed methods do not work as claimed.
- More negative results should be published—they are just as important as the positive ones.
- Release your software! If we cannot compare our results to your benchmarks, at least let us run your method on ours.

Sharing source code has several extra benefits. Although code is not valued as any significant academic contribution these days (I wonder if UNIX could ever have developed in the current research atmosphere?), it *will* have an impact on the rate of progress of the research field. If your code is well written, chances are that someone will choose to work in your field of research, simply because he or she can stand on your (virtual) shoulders. You may benefit from improvements and bugs fixed by others using your code, which will increase the confidence in your future research based on that code. Also, if you are a Ph.D. student, it will help you make contacts. When you have finished your Ph.D., having contacts is *the* most important thing.

Model Checking

The term "model checking" was coined by Edmund Clarke and Allen Emerson in the early 80s, as a verification technique for finite state concurrent systems (the term *model* refers to a formal description of the system under verification). Since then, the meaning of the term has broadened, and now includes some methods of verifying infinite state systems—infinite either due to continuous time or infinite data, such as integers—and non-concurrent software systems, such as *device drivers*. However, key features of model checking that have been preserved are the high degree of automation and the low degree of knowledge demanded from the user. The successful deployment of model checking in industry has largely depended on this low-threshold, high-output user model.

In the early 90s, *symbolic* model checking (SMC), the topic of this Thesis, was introduced. Previously, model checkers worked by traversing the state-space of the model *explicitly* in order to prove a property. Models with a big state-space could be verified by constructing a conservative *abstraction*, with a smaller state-space. However, the advent of symbolic model checking offered an alternative to this approach. By reasoning on compactly represented *sets* of states instead of individual states, systems with huge state-spaces could be verified directly, without using any abstraction technique [BC+90]. Today, symbolic model checking is the dominating variant for hardware verification, whilst explicit model checking has found important application in software verification [BR02,CD+00,Blast].

Current hardware model checkers work as follows: A VHDL or Verilog description (the so-called RTL, *register transfer level*, description) of a design is read in and passed onto a synthesis tool. The synthesizer constructs a gate-level description of the design, which is used in the logical reasoning of the SMC tool. Properties to be proven are stated in temporal logic ("propositional logic with time"). LTL/CTL [Pnu77,CES86] are typical logics used in the research community. The current trend in industry, however, is towards more complex specification languages, such as PSL and SVA [PSL04,SVA04]. How to algorithmically check properties on the gate-level description is the topic of *Paper 1* and *Paper 2* of this Thesis. A more comprehensive introduction can be found in [CGP01].

Formal methods are about increasing confidence in designs. It is a war on many fronts. One can, for instance, develop more expressive and less error prone RTL languages. Or one can improve the methodology used for developing complex designs, which in turn might require a better integration of the verification tools into the current design flow. Better instruction and further education of personnel might also improve the quality of end products.

To formally verify a design completely can be argued philosophically impossible. One can never be completely sure that the specification accurately captures the intention of the design. Furthermore, there is always room for uncertainty as to whether the model is a valid abstraction of the physical reality we wish to reason about. In SMC, the translation from RTL to a gate-level model is non-trivial and may introduce errors. Finally, the verifier itself is a big and complex piece of software; how do you know it is correct? Or indeed the computer running it?

However, none of the above concerns is treated in this Thesis. The focus is exclusively on the improvement of the underlying proof methods used in the verification; whose power defines the boundaries of what can be *formally* verified. Having said that, no less significance is assigned to any of the other ways of reaching better methods for constructing correct chips.

Modeling. This Thesis adopts the commonly used model for *finite state machines* (FSMs) based on gate-level synchronous circuits (see for instance [CGP01], chapter 2, or [Cla01]). The model is an abstraction of real circuits, ignoring such things as timing, layout, multiple clock-domains, and different memory elements. Instead, an (abstract) circuit consists of *logical gates*, *unit delays*, *input wires*, and *internal wires* connecting the gates and delays. A logical gate can be, for instance, an AND gate with two inputs and one output. A delay always has one

input and one output, where the output holds the value of the input one clock-cycle earlier. All wires are assumed to have a name, so that internal points can be referred to. The input wires are unconstrained, meaning they can assume any value at any clock-cycle. A more detailed description of this model is presented in *Paper 1*.

A state of the FSM is identified with a boolean vector corresponding to the values currently output by the delays. We will assume the initial values of the delays to be constrained (for instance, our model may require all delays to output zero during the first clock-cycle). Given this, the circuit may not be able to reach all possible states. It therefore makes sense to speak about *reachable* and *unreachable* states, a terminology that often causes confusion in the community of explicit state model checking.

Basic approaches of SMC. As discussed above, current symbolic model checking of safety properties are based on one of the two following ideas:

- **Breadth First Search.** The set of reachable states is computed iteratively, starting with the set of initial states and successively computing the states reachable after one clock-cycle, two clock-cycles and so forth. Central to this approach is the representation of sets of states and the *image computation*. The image of a set of states is those states reachable in one clock-cycle. Once a fixed point is reached, the safety property can be verified by an inclusion test.
- **Induction.** A safety property can be established by finding an *induction invariant*. The invariant must fulfill three properties: (1) It should imply the safety property, (2) it should hold in all legal initial states of the system (the base case), and (3) assuming it to hold for a particular state should imply that it holds for all possible next states (the induction step). A candidate for the induction invariant is the safety property itself. Often, however, the property is not strong enough to be inductive. Stronger induction schemes may then be used; assuming the property to hold for k consecutive states instead of just the previous state. Putting limitations on such sequences of k states can yield a complete method. Another option is to automatically strengthen the property to an inductive invariant.

A very interesting merge of these two ideas, based on calculating an *interpolant* from the resolution proof produced by the SAT solver [Cra57], was recently published by McMillan [McM03]. In the presented algorithm, an inductive invariant is calculated by an over-approximative reachability analysis. The result is a fully SAT based, complete model checking procedure.

Contributions

This Thesis contributes to the research community in the following ways:

- A first attempt at non-BDD based reachability is presented, including examples where BDD based reachability is outperformed.

– A clear exposition of the implementation details of a modern incremental SAT solver is included. The solver is extensible to arbitrary boolean constraints.
– A practical implementation of complete k-induction is presented, based on incremental SAT. The presentation includes a discussion on the relation between BMC and k-induction.
– The important problem of how to encode problems efficiently in CNF is addressed.

Further Reading

For further information on the area of hardware verification in practice, the reader is referred to the conference proceedings of *Formal Methods in Computer-Aided Design* (FMCAD), *Computer Aided Verification* (CAV), and *Correct Hardware Design and Verification Methods* (CHARME).

A growing forum for SAT is the *International Conference on Theory and Applications of Satisfiability Testing.* Further references can be found at the *SAT Live* web page (www.satlive.org), and in *Paper 3* of this Thesis.

References

[AJS98] M. Aagaard, R.B. Jones, C.J.H. Seger. **"Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment"** in *Proc. of Design Automation Conference (DAC)*, ACM Press, 1998.

[BC+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang. **"Symbolic Model Checking: 10^{20} States and Beyond"** in *Proc. of the 5^{th} Annual IEEE Symposium on Logic in Computer Science*, 1990.

[BCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. **"Symbolic model checking without BDDs"** in *Proc. of 8^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS:1579, Springer-Verlag, 1999.

[Bier04] A. Biere. **"Resolve and Expand"** in *The 7^{th} Int. Conf. on Theory and Appl. of Satisfiability Testing (SAT)*, 2004.

[Blast] T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, D. Beyer, A. Chlipala. **"BLAST – Berkeley Lazy Abstraction Software Verification Tool"**, http://www-cad.eecs.berkeley.edu/~rupak/blast

[BLM01] P. Bjesse, T. Leonard, A. Mokkedem. **"Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers"** in *Proc. of 12^{th} Int. Conf. on Computer Aided Verification (CAV)*, LNCS:2102, Springer-Verlag, 2001.

[BR02] T. Ball, S.K. Rajamani. **"Debugging System Software via Static Analysis"**, in *Symp. on Principles of Prog. Languages (POPL)*, 2002.

[Bry86] R.E. Bryant. **"Graph-Based Algorithms for Boolean Function Manipulation"** in *IEEE Transactions on Computers, vol. C-35, no. 8*, 1986.

[BS99] A. Borälv, G. Stålmarck. **"Formal verification in Railways"**, in *Industrial-Strength Formal Methods in Practise*, Springer-Verlag, 1999. ISBN 1-85233-640-4.

[CD+00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, R.H. Zheng. **"Bandera: Extracting Finite-state Models from Java Source Code"** in *Conf. on Software Engineering (ICSE)*, 2000.

[CES86] E.M. Clarke, E.A. Emerson, A.P. Sistla. **"Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications"**, in *ACM Transactions of Programming Languages and Systems*, Vol. 8, No. 2, April 1986.

[CG+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. **"Counterexample-guided abstraction refinement for symbolic model checking"** in *Journal of the ACM (JACM) vol. 50, issue 5, Sept.*, 2003.

[CGP01] E.M. Clarke, O. Grumberg, D.A. Peled. **"Model Checking"**, *The MIT Press*, 2001. ISBN 0-262-03270-8.

[Cla01] Koen Claessen. **"Embedded Languages for Describing and Verifying Hardware"**, Ph.D. Thesis, Chalmers University of Technology, 2001. ISBN 91-7291-014-3.

[CMB90] O. Coudert, J.C. Madre, C. Berthet. **"Verifying temporal properties of sequential machines without building their state diagrams"** in *Proc. of Int. Conf. on Computer-Aided Verification (CAV'90)*, American Mathematical Society, 1990.

[Cra57] W. Craig. **"Linear reasoning: A new form of the Herbrand-Gentzen theorem"** in *Journal of Symbooic Logic 22(3)*, 1957.

[Eijk98] C.A.J. van Eijk. **"Sequential Equivalence Checking without State Space Traversal"** in *Proc. Conf. on Design, Automation and Test in Europe*, 1998.

[GGA04] M. Ganai, A. Gupta, P. Ashar. **"Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring"** in *Proc. of Int. Conf. in Computer Aided Design (ICCAD)*, ACM Press, 2004.

[GYA00] A. Gupta, Z. Yang, P. Ashar, A. Gupta. **"SAT-Based Image Computation with Application in Reachability Analysis"** in *Proc. of Conf. on Formal Methods in Computer-Aided Design (FMCAD'00)*, 2000.

[GZAM02] M. Ganai, L. Zhang, P. Ashar, A. Gupta, S. Malik. **"Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver"** in *Design Automation Conference (DAC)*, 2002.

[HKB96] R. Hojati, S.C. Krishnan, and R.K. Brayton. **"Early Quantification and Partitioned Transition Relations"** in *Int. Conf. on Computer Design, VLSI in Computers and Processors (ICCD)*, 1996.

[KGP01] A. Kühlmann, M.K. Ganai, V. Paruthi. **"Circuit-based Boolean Reasoning"** in *Design Automation Conference (DAC)*, 2001.

[KSM94] W. Kunz, D. Stoffel, P.R. Menon. **"Multi-Level Logic Optimization by Implication Analysis"** in *Proc. of Int. Conf. in Computer Aided Design (ICCAD)*, ACM Press, 1994.

[MA03] K.L. McMillan, N. Alma **"Automatic Abstraction without Counterexamples"** in *Proc. of 9^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS:2619, Springer-Verlag, 2003.

[Mat96] Y. Matsunaga. **"An Efficient Equivalence Checker for Combinational Circuits"**, in *33^{rd} Design Automation Conference (DAC)*, 1996.

[McM02] K.L. McMillan. **"Applying SAT Methods in Unbounded Symboic Model Checking"**, in *Proc. of 13^{th} Int. Conf. on Computer Aided Verification (CAV'02)*, LNCS:2404, Springer-Verlag, 2002.

[McM03] K.L. McMillan. **"Interpolation and SAT-Based Model Checking"**, in *Proc. of 14^{th} Int. Conf. on Computer Aided Verification (CAV'03)*, LNCS:2725, Springer-Verlag, 2003.

[McM93] K.L. McMillan. **"Symbolic Model Checking"**, ISBN 0-7923-9380-5, Kluwer Academic Publishers, 1993.

[Min96] S. Minato. **"Fast Factorization Method for Implicit Cube Set Representation"** in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 4*, 1996.

[MJB05] A. Mishchenko, R. Jiang, R. Brayton. **"FRAIGs: A Unifying Representation for Logic Synthesis and Verification"** (Submitted to DAC'05)

[MS04] D. Mehta, S. Sahni (editors). **"Handbook on Data Structures and Applications / Chapter 56 (Fabio Somenzi) "Boolean Function Representation for VLSI Design"** in *CRC Press (C4355), ISBN 1584884355*, 2004.

[MW+88] S. Malik, A.R. Wang, R.K. Brayton, A.S. Vincentelli. **"Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment"**, in *Proc. of Int. Conf. on Computer Aided Design (ICCAD)*, 1988.

[MZ01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. **"Chaff: Engineering an Efficient SAT Solver"** in *Proc. of the 38^{th} Design Automation Conference*, 2001.

[Pnu77] A. Pnueli. **"The Temporal Logic of Programs"**, in *Proc. of 18^{th} IEEE Symp. on Foundation of Cumpter Science*, 1977.

[PSL04] **"Property Specification Language, Reference Manual, version 1.1 (2004)"** http://www.accellera.org

[Rud93] R. Rudell. **"Dynamic Variable Ordering for Ordered Binary Decision Diagrams"** in *Proc. of Int. Conf. on Computer-Aided Design (ICCAD)*, 1993.

[SK97] D. Stoffel, W. Kunz. **"Record & Play: A Structural Fixed Point Iteration for Sequential Circuit Verification"** in *Int. Conf. on Computer Aided Design (ICCAD)*, IEEE Computer Society, 1997.

[SS98] M. Sheeran, G. Stålmarck. **"A Tutorial on Stålmarcks's Proof Procedure for Propositional Logic"** in *Proc. of Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 1998.

[SSS00] M. Sheeran, S. Singh, G. Stålmarck. **"Checking safety properties using induction and a SAT-solver"** in *Formal Methods in Computer Aided Design*, LNCS:1954, Springer-Verlag, 2000.

[SVA04] **"System Verilog 3.1a, Language Reference Manual, Accellera's Extensions to Verilog (2004)"** http://www.accellera.org

[Vel04] M.N. Velev. **"Efficient Translation of Boolean Formulas to CNF"** in *Formal Verification of Microprocessors, Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2004.

Paper 1

Symbolic Reachability Analysis based on SAT-Solvers

Parosh Abdulla, Per Bjesse, Niklas Eén

Synopsis

In this paper, we start from classic BDD-based algorithms for reachability analysis, and replace the BDDs—a restricted form of propositional formulas—with the more standard representation using AND and NOT connectives. As for BDDs, the formulas are stored as graphs to allow sharing of common sub-formulas, but unlike BDDs, the representation is not canonical.

The reachability analysis is carried out by a fixed-point iteration. Starting from a formula representing the set of initial states, *image computation* is repeatedly applied. The image of a set is all states reachable from that set in one clock-cycle. Taking the union of all states reachable in n cycles will eventually lead to a least fixed-point ("the reachable states"), as we consider finite systems.

Efficient image computation using BDDs is a well studied topic. However, before applying BDD-methods, the system must be translated to BDDs. For sequential circuits—the finite systems we consider—it is known that the translation can lead to an exponential blow-up. Using standard propositional formulas instead, we guarantee a one-to-one mapping between the circuit and its internal representation. Therefore, we introduce an image computation based on this non-canonical representation. As an effect of this, detecting when a fixed-point has been reached is no longer trivial. For this purpose we use a SAT-solver. Supported by benchmarks, we claim that the resulting method can work well for systems where BDDs fail.

The paper was presented at *TACAS 2000* and won the best-paper award.

Symbolic Reachability Analysis based on SAT-Solvers

Parosh Aziz Abdulla[1], Per Bjesse[2], Niklas Eén[2]

[1] Uppsala University and Prover Technology, Sweden,
parosh@docs.uu.se
[2] Chalmers University of Technology, Sweden
{bjesse,een}@cs.chalmers.se

Abstract. The introduction of symbolic model checking using Binary Decision Diagrams (BDDs) has led to a substantial extension of the class of systems that can be algorithmically verified. Although BDDs have played a crucial role in this success, they have some well-known drawbacks, such as requiring an externally supplied variable ordering and causing space blowups in certain applications. In a parallel development, SAT-solving procedures, such as Stålmarck's method or the Davis-Putnam procedure, have been used successfully in verifying very large industrial systems. These efforts have recently attracted the attention of the model checking community resulting in the notion of *bounded model checking*. In this paper, we show how to adapt standard algorithms for symbolic reachability analysis to work with SAT-solvers. The key element of our contribution is the combination of an algorithm that removes quantifiers over propositional variables and a simple representation that allows reuse of subformulas. The result will in principle allow many existing BDD-based algorithms to work with SAT-solvers. We show that even with our relatively simple techniques it is possible to verify systems that are known to be hard for BDD-based model checkers.

1 Introduction

In recent years *model checking* [CES86,QS82] has been widely used for algorithmic verification of finite-state systems such as hardware circuits and communication protocols. In model checking, the specification of the system is formulated as a temporal logical formula, while the implementation is described as a finite-state transition system. Early model-checking algorithms suffered from *state explosion*, as the size of the state space grows exponentially with the number of components in the system. One way to reduce state explosion is to use *symbolic model checking* [BCMD92,McM93], where the transition relation is coded symbolically as a boolean expression, rather than explicitly as the edges of a graph. Symbolic model checking achieved its major breakthrough after the introduction of *Binary Decision Diagrams* (BDDs) [Bry86] as a data structure for representing boolean expressions in the model checking procedure. An important property of BDDs is that they are canonical. This allows for substantial

sub-expression sharing, often resulting in a compact representation. In addition, canonicity implies that satisfiability and validity of boolean expressions can be checked in constant time. However, the restrictions imposed by canonicity can in some cases lead to a space blowup, making memory a bottleneck in the application of BDD-based algorithms. There are examples of functions, for example multiplication, which do not allow sub-exponential BDD representations. Furthermore, the size of a BDD is dependent on the variable ordering which in many cases is hard to optimize, both automatically and by hand. BDD-based methods can typically handle systems with hundreds of boolean variables.

A related approach is to use satisfiability solvers, such as implementations of Stålmarck's method [Stå] and the Davis-Putnam procedure [Zha97]. These methods have already been used successfully for verifying industrial systems [SS00,Bor97,Bor98,SS90,GvVK95]. SAT-solvers enjoy several properties which make them attractive as a complement to BDDs in symbolic model checking. For instance, their performance is less sensitive to the size of the formulas, and they can in some cases handle propositional formulas with thousands of variables. Furthermore, SAT-solvers do not suffer from space explosion, and do not require an external variable ordering to be supplied. Finally, satisfiability solving is an NP-complete problem, whereas BDD-construction solves a #P-complete problem [Pap94] as it is possible to determine the number of models of a BDD in polynomial time. #P-complete problems are widely believed to be harder than NP-complete problems.

The aim of this work is to exploit the strength of SAT-solving procedures in order to increase the class of systems amenable to verification via the traditional symbolic methods. We consider modifications of two standard algorithms—forward and backward reachability analysis—where formulas are used to characterize sets of reachable states [Bje99]. In these algorithms we replace BDDs by satisfiability checkers such as the PROVER implementation of Stålmarck's method [Stå] or SATO [Zha97]. We also use a data structure which we call *Reduced Boolean Circuits* (RBCs) to represent formulas. RBCs avoid unnecessarily large representations through the reuse of subformulas, and allow for efficient storage and manipulation of formulas. The only operation of the reachability algorithms that does not carry over straightforwardly to this representation is quantification over propositional variables. Therefore, we provide a simple procedure for the removal of quantifiers, which gives adequate performance for the examples we have tried so far.

We have implemented a tool FIXIT [Eén99] based on our approach, and carried out a number of experiments. The performance of the tool indicates that even though we use simple techniques, our method can perform well in comparison to existing ones.

Related Work. *Bounded Model Checking* (BMC) [BCC+99,BCCZ99,BCRZ99] is the first approach in the literature to perform model checking using SAT-solvers. To check reachability, the BMC procedure searches for counterexamples (paths to undesirable states) by "unrolling" the transition relation k steps. The unrolling is described by a (quantifier-free) formula which characterizes the set

of feasible paths through the transition relation with lengths smaller than or equal to k. The search can be terminated when the value of k is equal to the *diameter* of the system—the maximum length of all shortest path between states in the system. Although the diameter can be specified by a logical formula, its satisfiability is usually hard to check, making BMC incomplete in practice. Furthermore, for "deep" transition systems, formulas characterizing the set of reachable states may be much smaller than those characterizing witness paths. Since our method is based on encodings of sets of states, it may in some cases cope with systems which BMC fails to analyze as it generates formulas that are too large.

Our representation of formulas is closely related to *Binary Expression Diagrams* (BEDs) [AH97,HWA97]. In fact there are straightforward linear space translations back and forth between the representations. Consequently, RBCs share the good properties of BEDs, such as being exponentially more succinct than BDDs [AH97]. The main difference between our approach and the use of BEDs is the way in which satisfiability checking and existential quantification is handled. In [AH97], satisfiability of BEDs is checked through a translation to equivalent BDDs. Although many simplifications are performed at the BED level, converting to BDDs during a fixpoint iteration could cause degeneration into a standard BDD-based fixpoint iteration. In contrast, we check satisfiability by mapping RBCs back to formulas which are then fed to external SAT-solvers. In fact, the use of SAT-solvers can also be applied to BEDs, but this does not seem to have been explored so far. Furthermore, in the BED approach, existential quantification is either handled by introducing explicit quantification vertices, or by a special transformation that rewrites the representation into a form where naive expansion can be applied. We use a similar algorithm that also applies an extra inlining rule. The inlining rule is particularly effective in the case of backward reachability analysis, as it is always applicable to the generated formulas. To our knowledge, no results have been reported in the literature on application of BEDs in symbolic model checking. We would like to emphasize that we view RBCs as a relatively simple representation of formulas, and not as a major contribution of this work.

2 Preliminaries

We verify systems described as synchronous circuits constructed from elementary combinational gates and unit delays—a simple, yet popular, model of computation. The unit delays are controlled by a global clock, and we place no restriction on the inputs to a circuit. The environment is free to behave in any fashion.

We define the *state-holding elements* of a circuit to be the primary inputs and the contents of the delays, and define a *valuation* to be an assignment of boolean values to the state-holding elements. The behaviour of a circuit is modelled as a state-transition graph where (1) each valuation is a state; (2) the initial states comprise all states that agree with the initial values of the delays; and (3) there

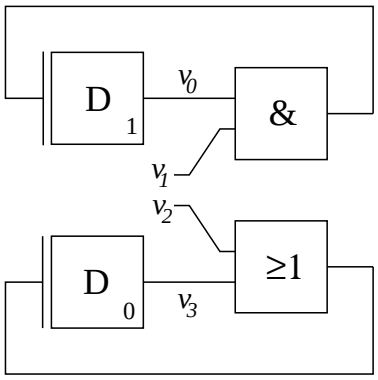


Fig. 1. A simple circuit built from combinational gates and delays.

is a transition between two states if the circuit can move between the source state and the destination state in one clock cycle.

We construct a symbolic encoding of the transition graph in the standard manner. We assign every state-holding element a propositional state variable v_i, and make two copies of the set of state variables, $s = \{v_0, v_1, \ldots, v_k\}$ and $s' = \{v'_0, v'_1, \ldots, v'_k\}$. Given a circuit we can now generate two *characteristic formulas*. The first of the characteristic formulas, $Init(s) = \bigwedge_i v_i \leftrightarrow \phi_i$, defines the initial values of the state-holding elements. The second characteristic formula, $Tr(s, s') = \bigwedge_i v'_i \leftrightarrow \psi_i(s)$, defines the next-state values of state-holding elements in terms of the current-state values.

Example 1. The following formulas characterize the circuit in Figure 1:

$$\begin{aligned} Init &= (v_0 \leftrightarrow \top) \wedge (v_3 \leftrightarrow \bot) \\ Tr &= (v'_0 \leftrightarrow (v_0 \wedge v_1)) \ \wedge \ (v'_3 \leftrightarrow (v_2 \vee v_3)) \end{aligned}$$

We investigate the underlying state-transition graph by applying operations at the formula level. In doing so we make use of the following three facts. First, the relation between any points in a given circuit can be expressed as a propositional formula over the state-holding variables. Second, we can represent any set S of transition-graph states by a formula that is satisfied exactly by the states in S. Third, we can lift all standard set-level operations to operations on formulas (for example, set inclusion corresponds to formula-level implication and set nonemptiness checking to satisfiability solving, respectively).

3 Reachability Analysis

Given the characteristic formulas of a circuit and a formula $Bad(s)$, we define the *reachability problem* as that of checking whether it is possible to reach a state that satisfies $Bad(s)$ from an initial state. As an example, in the case of

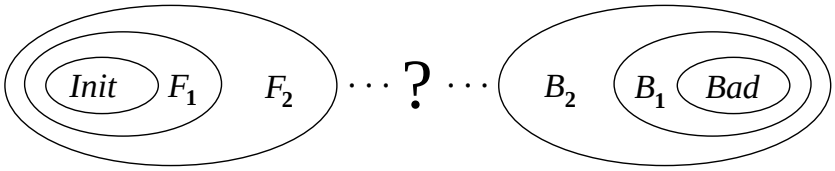


Fig. 2. The intuition behind the reachability algorithms.

the circuit in Figure 1, we might be interested in whether the circuit could reach a state where the two delay elements output the same value (or equivalently, where the formula $v_0 \leftrightarrow v_3$ is satisfiable). We adapt two standard algorithms for performing reachability analysis. In *forward reachability* we compute a sequence of formulas $F_i(s)$ that characterize the set of states that the initial states can reach in i steps:

$$F_0(s) = Init$$
$$F_{i+1}(s') = toProp(\exists s.\ Tr(s, s') \wedge F_i(s)))$$

Each computation of F_{i+1} gives rise to a *Quantified Boolean Formula* (QBF), which we translate back to a pure propositional formula using an operation *toProp* (defined in in Section 5). We terminate the sequence generation if either (1) $F_n(s) \wedge Bad(s)$ is satisfiable: this means that a bad state is reachable; hence we answer the reachability problem positively; or (2) $\bigvee_{k=0}^{n} F_k(s) \rightarrow \bigvee_{k=0}^{n-1} F_k(s)$ holds: this implies that we have reached a fixpoint without encountering a bad state; consequently the answer to the reachability question is negative.

In *backward reachability* we instead compute a sequence of formulas $B_i(s)$ that characterize the set of states that can reach a bad state in i steps:

$$B_0(s) = Bad$$
$$B_{i+1}(s) = toProp(\exists s'.\ Tr(s, s') \wedge B_i(s')))$$

In a similar manner to forward reachability, we terminate the sequence generation if either (1) $B_n(s) \wedge Init(s)$ is satisfiable, or (2) $\bigvee_{k=0}^{n} B_k(s) \rightarrow \bigvee_{k=0}^{n-1} B_k(s)$ holds.

Figure 2 shows the intuition behind the algorithms. We remark that the two reachability methods can be combined by alternating between the computation of F_{i+1} and B_{i+1}. The generation can be terminated when either a fixpoint is reached in some direction, or when F_n and B_n intersect. However, we do not make use of hybrid analyses in this paper.

We need to address three nontrivial issues in an implementation of the adapted reachability algorithms. First, we must avoid the generation of unnecessarily large formula characterizations of the sets F_i and B_i—formulas are not a canonical representation. Second, we must define the operation *toProp* in such a way that it translates quantified boolean formulas to propositional logic without

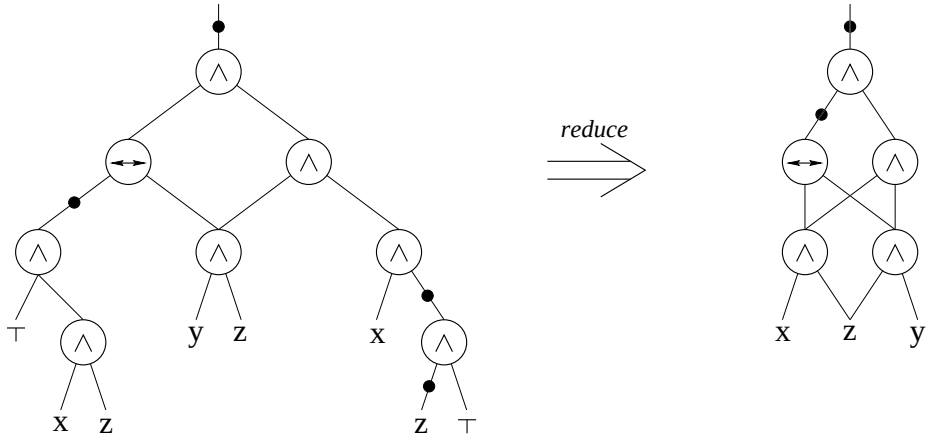


Fig. 3. A non-reduced *Boolean Circuit* and its reduced form.

needlessly generating exponential results. Third, we must interface efficiently to external satisfiability solvers. The remainder of the paper explains our solutions, and evaluates the resulting reachability checker.

4 Representation of Formulas

Let **Bool** denote the set of booleans, **Vars** denote the set of propositional variables, including a special variable $\top$ for the constant *true*, and **Op** denote the set $\{\leftrightarrow, \wedge\}$.

We introduce the representation *Boolean Circuit* (BC) for propositional formulas. A BC is a directed acyclic graph, $(\mathbf{V}, \mathbf{E})$. The vertices $\mathbf{V}$ are partitioned into internal nodes, $\mathbf{V_I}$, and leaves, $\mathbf{V_L}$. The vertices and edges are given attributes as follows:

- Each internal vertex $v \in \mathbf{V_I}$ has three attributes: A binary operator $op(v) \in \mathbf{Op}$, and two edges $left(v), right(v) \in \mathbf{E}$.
- Each leaf $v \in \mathbf{V_L}$ has one attribute: $var(v) \in \mathbf{Vars}$.
- Each edge $e \in \mathbf{E}$ has two attributes: $sign(e) \in \mathbf{Bool}$ and $target(e) \in \mathbf{V}$.

We observe that negation is coded into the edges of the graph, by the *sign* attribute. Furthermore, we identify *edges* with *subformulas*. In particular, the whole formula is identified with a special top-edge having no source vertex. The interpretation of an edge as a formula is given by the standard semantics of $\wedge, \leftrightarrow$ and $\neg$ by viewing the graph as a parse tree (with some common sub-expressions shared). Although $\wedge$ and $\neg$ are functionally complete, we choose to include $\leftrightarrow$ in the representation as it would otherwise require three binary connectives to express. Figure 3 shows an example of a BC.

A *Reduced Boolean Circuit* (RBC) is a BC satisfying the following properties:

```
reduce(AND, left ∈ RBC, right ∈ RBC)
    if    (left = right)   return left
    elif  (left = ¬right)  return ⊥
    elif  (left = ⊤)       return right
    elif  (right = ⊤)      return left
    elif  (left = ⊥)       return ⊥
    elif  (right = ⊥)      return ⊥
    else                   return NIL

reduce(EQUIV, left ∈ RBC, right ∈ RBC)
    if    (left = right)   return ⊤
    elif  (left = ¬right)  return ⊥
    elif  (left = ⊤)       return right
    elif  (left = ⊥)       return ¬right
    elif  (right = ⊤)      return left
    elif  (right = ⊥)      return ¬left
    else                   return NIL

mk_Comp(op ∈ Op, left ∈ RBC, right ∈ RBC, sign ∈ Bool)
    result := reduce(op, left, right)
    if (result ≠ NIL)
        return id(result, sign)    – id returns result or ¬result depending on sign

    if (right < left)
        (left, right) := (right, left)    – Swap the values of left and right

    if (op = EQUIV)
        sign  := sign xor sign(left) xor sign(right)
        left  := unsigned(left)
        right := unsigned(right)

    result := lookup(RBC_env, (op, left, right))    – Look for vertex in environment
    if (result = NIL)
        result := insert(RBC_env, (op, left, right))
    return id(result, sign)
```

Fig. 4. Pseudo-code for creating a composite RBC from two existing RBCs.

1. All common subformulas are shared so that no two vertices have identical attributes.
2. The constant $\top$ never occurs in an RBC, except for the single-vertex RBCs representing *true* or *false*.
3. The children of an internal vertex are syntactically distinct, $\mathit{left}(v) \neq \mathit{right}(v)$.
4. If $\mathit{op}(v) = \leftrightarrow$ then the edges to the children of v are unsigned.
5. For all vertices v, $\mathit{left}(v) < \mathit{right}(v)$, for some total order $<$ on BCs.

The purpose of these constraints is to identify as many equivalent formulas as possible, and thereby increase the amount of subformula sharing. For this reason we allow only one representation of $\neg(\phi \leftrightarrow \psi) \iff (\neg\phi \leftrightarrow \psi)$ (in 4 above), and $(\phi \wedge \psi) \iff (\psi \wedge \phi)$ (in 5 above).

The RBCs are created in an implicit environment, where all existing subformulas are tabulated. We use the environment to assure property (1). Figure 4 shows the only non-trivial constructor for RBCs, *mk_Comp*, which creates a composite RBC from two existing RBCs (we use $x \in \mathrm{Vars}(\phi)$ to denote that x is a variable occurring in the formula ϕ). It should be noted that the above properties only takes constant time to maintain in *mk_Comp*.

5 Quantification

In the reachability algorithms we make use of the operation *toProp* to translate QBF formulas into equivalent propositional formulas. We reduce the translation of a set of existential quantifiers to the iterated removal of a single quantifier after we have chosen a quantification order. In the current implementation an arbitrary order is used, but we are evaluating more refined approaches.

Figure 5 presents the quantification algorithm of our implementation. By definition we have:

$$\exists x \,.\, \phi(x) \iff \phi(\bot) \vee \phi(\top) \qquad (*)$$

The definition can be used to naively resolve the quantifiers, but this may yield an exponential blowup in representation size. To try to avoid this, we use the following well-known identities (applied from left to right) whenever possible:

Inlining:

$$\exists x \,.\, (x \leftrightarrow \psi) \wedge \phi(x) \iff \phi(\psi) \qquad (\text{where } x \notin \text{Vars}(\psi))$$

Scope Reduction:

$$\exists x \,.\, \phi(x) \wedge \psi \iff (\exists x.\phi(x)) \wedge \psi \qquad (\text{where } x \notin \text{Vars}(\psi))$$
$$\exists x \,.\, \phi(x) \vee \psi(x) \iff (\exists x.\phi(x)) \vee (\exists x.\psi(x))$$

When applicable, *inlining* is an effective method of resolving quantifiers as it immediately removes all occurrences of the quantified variable x. The applicability of the transformation relies on the fact that the formulas occurring in reachability often have a structure that matches the rule. This is particularly true for backward reachability as the transition relation is a conjunction of next state variables defined in terms of current state variables $\bigwedge_i v_i' \leftrightarrow \psi_i(s)$.

The first step of the inlining algorithm temporarily changes the representation of the top-level conjunction. From the binary encoding of the RBC, we extract an equivalent set representation $\bigwedge\{\phi_0, \phi_1, \ldots, \phi_n\}$. If the set contains one or more elements of the form $x \leftrightarrow \psi$, the smallest such element is removed from the set and its right-hand side ψ is substituted for x in the remaining elements. The set is then re-encoded as an RBC.

If inlining is not applicable to the formula (and variable) at hand, the translator tries to apply the *scope reduction* rules as far as possible. This may result in a quantifier being pushed through an OR (represented as negated AND), in which case inlining may again be possible.

For subformulas where the scope can no longer be reduced, and where inlining is not applicable, we resort to *naive quantification* (*). Reducing the scope as much as possible before doing this will help prevent blowups. Sometimes the quantifiers can be pushed all the way to the leaves of the RBC, where they can be eliminated.

Throughout the quantification procedure, we may encounter the same subproblem more than once due to shared subformulas. For this reason we keep a table of the results obtained from all previously processed subformulas.

```
– Global variable processed tabulates the results of the performed quantifications.

quant_naive(φ ∈ RBC, x ∈ Vars)
    result = subst(φ, x, ⊥) ∨ subst(φ, x, ⊤)
    insert(processed, φ, x, result)
    return result

quant_reduceScope(φ ∈ RBC, x ∈ Vars)
    if (x ∉ Vars(φ)) return φ
    if (φ = x)       return ⊤

    result := lookup(processed, φ, x)
    if (result ≠ NIL)
        return result

    – In the following φ must be composite and contain x:
    if (φ_op = EQUIV)
        result := quant_naive(φ, x)
    elif (not φ_sign)    – Operator AND, unsigned
        if    (x ∉ Vars(φ_left))  result := φ_left ∧ quant_reduceScope(φ_right, x)
        elif  (x ∉ Vars(φ_right)) result := quant_reduceScope(φ_left, x) ∧ φ_right
        else                      result := quant_naive(φ, x)
    else    – Operator AND, signed ("OR")
        result := quant_inline(¬φ_left, x) ∨ quant_inline(¬φ_right, x)

    insert(processed, φ, x, result)
    return result

quant_inline(φ ∈ RBC, x ∈ Vars)    – "Main"
    C := collectConjuncts(φ)    – Merge all binary ANDs at the top of φ into a
                                  "big" conceptual conjunction (returned as a set).
    ψ := findDef(C, x)          – Return the smallest formula ψ such that (x ↔ ψ)
                                  is a member of C.
    if (ψ ≠ NIL)
        C' := C \ (x ↔ ψ)                    – Remove definition from C.
        return subst(makeConj(C'), x, ψ)     – makeConj builds an RBC.
    else
        return quant_reduceScope(φ, x)
```

Fig. 5. Pseudo-code for performing existential quantification over one variable. By ϕ_{left} we denote $left(target(\phi))$ etc. We use $\wedge$, $\vee$ as abbreviations for calls to *mk_Comp*.

6 Satisfiability

Given an RBC, we want to decide whether there exists a satisfying assignment for the corresponding formula by applying an external SAT-solver. The naive translation—unfold the graph to a tree and encode the tree as a formula—has the drawback of removing sharing. We therefore use a mapping where each internal node in the representation is allocated a fresh variable. This variable

is used in place of the subformula that corresponds to the internal node. The generated formula is the conjunction of all the definitions of internal nodes and the literal that defines the top node.

Example 2. The right-hand RBC in Figure 3 is mapped to the following formula in which the i_k variables define internal RBC nodes:

$$\begin{aligned}
&(i_0 \leftrightarrow \neg i_1 \wedge\ i_2) \\
\wedge\ &(i_1 \leftrightarrow i_3 \leftrightarrow i_4) \\
\wedge\ &(i_2 \leftrightarrow i_3 \wedge i_4) \\
\wedge\ &(i_3 \leftrightarrow x \wedge z) \\
\wedge\ &(i_4 \leftrightarrow z \wedge y) \\
\wedge\ &\neg i_0
\end{aligned}$$

A formula resulting from the outlined translation is *not* equivalent to the original formula without sharing, but it will be satisfiable if and only if the original formula is satisfiable. Models for the original formula are obtained by discarding the values of internal variables.

7 Experimental Results

We have implemented a tool FIXIT [Eén99] for performing symbolic reachability analysis based on the ideas presented in this paper. The tool has a *fixpoint mode* in which it can perform both forward and backward reachability analysis, and an *unroll mode* where it searches for counterexamples in a similar manner to the BMC procedure. We have carried out preliminary experiments on three benchmarks: a *multiplier* and a *barrel shifter* (both from the BMC distribution), and a *swapper* (defined by the authors). The first two benchmarks are known to be hard for BDD-based methods.

We present only time consumption. Memory consumption is much smaller than for BDD-based systems. Garbage collection has not yet been implemented in FIXIT, but the amount of simultaneously referenced memory peaks at about 1-2 MB in our experiments. We also know that the memory requirements of PROVER are relatively low (worst case quadratic in the formula size). In all the experiments, PROVER outperforms SATO, so we only present the measurements for PROVER. The test results for FIXIT are compared against results obtained from VIS release 1.3, BMC version 1.0f and CADENCE SMV release 09-01-99.

The Multiplier. The example models a standard 16×16 bit shift-and-add multiplier, with an output result of 32 bits. Each output bit is individually verified against the C6288 combinational multiplier of the ISCAS'85 benchmarks by checking that we cannot reach a state where the computation of the shift-and-add multiplier is completed, but where the selected result bit is not consistent with the corresponding output bit of the combinational circuit.

Bit	FixIt Fwd sec	FixIt Bwd sec	FixIt Unroll sec	BMC sec	VIS sec	SMV sec
0	0.8	2.0	0.7	1.0	5.3	41.41
1	0.9	2.3	0.7	1.4	5.4	41.29
2	1.1	3.0	0.8	2.0	5.3	42.46
3	1.8	3.9	0.9	4.0	5.5	42.57
4	3.0	6.1	1.2	8.2	6.2	[>450 MB]
5	7.2	9.9	1.8	19.9	10.2	–
6	24.3	21.5	3.8	66.7	32.9	–
7	100.0	61.9	11.8	304.6	153.5	–
8	492.8	224.7	45.2	1733.7	[>450 MB]	–
9	2350.6	862.6	197.8	9970.8	–	–
10	11927.5	3271.0	862.8	54096.8	–	–
11	60824.6	13494.3	3838.0	–	–	–
12	–	50000.0	16425.8	–	–	–

Table 1. Experimental results for the multiplier.

Table 1 presents the results for the multiplier. The SAT-based methods outperform VIS on all system sizes. The unroll mode is a constant factor more efficient than the fixpoint mode. However, we were unable to prove the diameter of the system by the diameter formula generated by BMC, which means that the verification performed by the unroll method (and BMC) should be considered partial.

The Barrel Shifter. The barrel shifter rotates the contents of a register file R with one position in each step. The system also contains a fixed register file R_0, related to R in the following way: if two registers from R and R_0 have the same contents, then their neighbours also have the same contents. We constrain the initial states to have this property, and the objective is to prove that it holds throughout the reachable part of the state space. The width of the registers is $\log |R|$ bits, and we let the BMC tool prove that the diameter of the circuit is $|R|$.

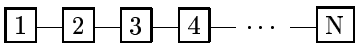
Table 2 presents the results for the barrel shifter. No results are presented for VIS due to difficulties in describing the extra constraint on the initial state in the VIS input format.

The backward reachability mode of FixIt outperforms SMV and BMC on this example. The reason for this is that the set of bad states is closed under the pre-image function, and hence FixIt terminates after only one iteration. SMV is unable to build the BDDs characterising the circuits for larger problem instances. The BMC tool has to unfold the system all the way up to the diameter, producing very large formulas; in fact, the version of BMC that we used could not generate formulas for larger instances than size 17 (a size 17 formula is 2.2 MB large). The oscillating timing data for the SAT-based tools reflects the heuristic nature of the underlying SAT-solver.

$\|R\|$	FixIt Fwd sec	FixIt Bwd sec	FixIt Unroll sec	BMC sec	Diam sec	SMV sec
2	1.7	0.1	0.1	0.0	0.0	0.0
3	2.3	0.1	0.1	0.0	0.0	0.1
4	3.0	0.1	0.2	0.0	0.0	0.1
5	42.4	0.2	0.3	0.1	0.1	44.15
6	848.9	0.2	0.5	0.3	0.1	[>450 MB]
7	5506.6	0.4	0.5	0.4	0.2	–
8	[>3 h]	0.5	1.0	1.2	0.3	–
9	–	0.8	1.6	2.4	0.6	–
10	–	1.1	2.3	8.6	0.8	–
11	–	1.5	2.3	3.3	1.1	–
12	–	2.3	4.1	25.6	1.5	–
13	–	2.6	3.9	7.1	2.0	–
14	–	3.2	7.8	80.1	2.6	–
15	–	3.7	8.6	75.1	3.5	–
16	–	4.3	12.1	150.0	4.4	–
17	–	6.7	11.0	34.6	7.9	–
18	–	8.7	30.5	?	?	–
19	–	9.2	15.6	?	?	–
20	–	13.5	49.1	?	?	–
...						
30	–	51.4	452.1	?	?	–
...						
40	–	230.5	2294.7	?	?	–
...						
50	–	501.5	8763.3	?	?	–

Table 2. Experimental results for the barrel shifter.

The Swapper. N nodes, each capable of storing a single bit, are connected linearly:



At each clock-cycle (at most) one pair of adjacent nodes may swap their values. From this setting we ask whether the single final state in which exactly the first $\lfloor N/2 \rfloor$ nodes are set to 1 is reachable from the single initial state in which exactly the last $\lfloor N/2 \rfloor$ nodes are set to 1. Table 3 shows the result of verifying this property.

Both VIS and SMV handle the example easily. FixIt can handle sizes up to 15, but does not scale up as well as VIS and SMV, as the representations get too large. This illustrates the importance of maintaining a compact representation during deep reachability problems; something that is currently not done by FixIt. However, BMC does even worse, even though the problem is a strict search for an existing counterexample—something BMC is generally good at. This shows that fixpoint methods can be superior both for proving unreachability and detecting counterexamples for certain classes of systems.

N	FixIt Fwd sec	FixIt Bwd sec	FixIt Unroll sec	BMC sec	VIS sec	SMV sec
3	0.2	0.2	0.18	0.01	0.3	0.03
4	0.3	0.3	0.17	0.01	0.3	0.05
5	0.6	0.5	0.27	0.08	0.3	0.04
6	0.9	1.5	1.76	7.24	0.4	0.06
7	1.7	3.7	131.19	989.51	0.4	0.06
8	3.8	10.4	[>2 h]	[>2 h]	0.4	0.08
9	9.7	58.9	–	–	0.4	0.11
10	27.7	187.1	–	–	0.4	0.11
11	74.1	779.2	–	–	0.5	0.18
12	238.8	4643.2	–	–	0.6	0.23
13	726.8	[>2 h]	–	–	0.7	0.30
14	2685.7	–	–	–	0.7	0.44
15	[>2 h]	–	–	–	0.7	0.61
...						
20	–	–	–	–	1.6	7.88
...						
25	–	–	–	–	3.3	52.97
...						
30	–	–	–	–	15.1	263.08
...						
35	–	–	–	–	39.1	929.57
...						
40	–	–	–	–	89.9	2944.26

Table 3. Experimental results for the swapper.

8 Conclusions and Future Work

We have described an alternative approach to standard BDD-based symbolic model checking which we think can serve as a useful complement to existing techniques. We view our main contribution as showing that with relatively simple means it is possible to modify traditional algorithms for symbolic reachability analysis so that they work with SAT-procedures instead of BDDs. The resulting method gives surprisingly good results on some known hard problems.

SAT-solvers have several properties which make us believe that SAT-based model checking will become an interesting complement to BDD-based techniques. For example, in a proof system like Stålmarck's method, formula size does not play a decisive role in the hardness of satisfiability checking. This is particularly interesting since industrial applications often give rise to formulas which are extremely large in size, but not necessarily hard to prove.

There are several directions for future work. We are currently surveying simplification methods that can be used to maintain compact representations. One promising approach [AH97] is to improve the local reduction rules to span over multiple levels of the RBC graphs. We are also interested in exploiting the structure of big conjunctions and disjunctions, and in simplifying formulas using algorithms based on Stålmarck's notion of formula saturation [Bje99]. As for

the representation itself, we are considering adding *if-then-else* and substitution nodes [HWA97]. Other ongoing work includes experiments with heuristics for choosing good quantification orderings.

In the longer term, we will continue to work on conversions of BDD-based algorithms. For example, we have already implemented a prototype model checker for general (fair) CTL formulas. Also, employing traditional BDD-based model checking techniques such as front simplification and approximate analysis are very likely to improve the efficiency of SAT-based model checking significantly.

Many important questions related to SAT-based model checking remain to be answered. For example, how should the user choose between bounded and fixpoint-based model checking? How can SAT-based approaches be combined with standard approaches to model checking?

Acknowledgements

The implementation of FIXIT was done as a Master's thesis at Prover Technology, Stockholm. Thanks to Purushothaman Iyer, Bengt Jonsson, Gordon Pace, Mary Sheeran and Gunnar Stålmarck for giving valuable feedback on earlier drafts.

This research was partially supported by TFR, the ASTEC competence center for advanced software technology, and the ARTES network for real-time research and graduate education in Sweden.

References

[AH97] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *Proc. 12^{th} IEEE Int. Symp. on Logic in Computer Science*, pages 88–98, 1997.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, 1999.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS '98, 8^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[BCMD92] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[BCRZ99] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs. In *Proc. 11^{th} Int. Conf. on Computer Aided Verification*, 1999.

[Bje99] P. Bjesse. Symbolic model checking with sets of states represented as formulas. Technical Report CS-1999-100, Department of Computer Science, Chalmers technical university, March 1999.

[Bor97] A. Borälv. The industrial success of verification tools based on Stålmarck's method. In *Proc. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10, 1997.

[Bor98] A. Borälv. Case study: Formal verification of a computerized railway interlocking. *Formal Aspects of Computing*, 10(4):338–360, 1998.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[Eén99] N. Eén. Symbolic reachability analysis based on SAT-solvers. Master's thesis, Dept. of Computer Systems, Uppsala university, 1999.

[GvVK95] J.F. Groote, S.F.M. van Vlijmen, and J.W.C. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *COMPASS'95*, 1995.

[HWA97] H. Hulgaard, P.F. Williams, and H.R. Andersen. Combinational logic-level verification using boolean expression diagrams. In *3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1997.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Pap94] C. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *5th International Symposium on Programming, Turin*, volume 137 of *Lecture Notes in Computer Science*, pages 337–352. Springer Verlag, 1982.

[SS90] G. Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In *SAFECOMP'90*, pages 31–36. Pergamon Press, 1990.

[SS00] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's method of propositional proof. *Formal Methods In System Design*, 16(1), 2000.

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

[Zha97] H. Zhang. SATO: an efficient propositional prover. In *Proc. Int. Conference om Automated Deduction (CADE'97)*, volume 1249 of *LNAI*, pages 272–275. Springer Verlag, 1997.

Paper 2

Temporal Induction by Incremental SAT Solving

Niklas Eén, Niklas Sörensson

Synopsis

This paper introduce the reader to induction techniques for safety checking (as proposed by Gunnar Stålmarck). It further shows how to make a modern SAT-solver incremental, and how induction algorithms can be improved by the use of such a SAT-solver. Several improvements to the basic induction scheme are presented, and the effect of each documented by thorough testing. The paper also contains a discussion on the relationship between *bounded model checking* (BMC) and induction.

The paper was presented at the *BMC 2003* workshop, affiliated with *CAV 2003*.

Temporal Induction by Incremental SAT Solving

Niklas Eén, Niklas Sörensson

Chalmers University of Technology, Sweden
{een,nik}@cs.chalmers.se

Abstract. We show how a very modest modification to a typical modern SAT-solver enables it to solve a series of related SAT-instances efficiently. We apply this idea to checking safety properties by means of *temporal induction*, a technique strongly related to *bounded model checking*. We further give a more efficient way of constraining the extended induction hypothesis to so called *loop-free* paths. We have also performed the first comprehensive experimental evaluation of induction methods for safety-checking.

1 Introduction

In recent years, SAT-based methods for hardware verification have become an important complement to traditional BDD-based model checking. Several methods have proven their usefulness on a number of industrial applications, in particular *bounded model checking* (BMC) [BCCZ99,BCRZ99,CFF+01]. In this paper we will focus our attention on how SAT-based verification procedures can be implemented more efficiently by a tighter integration with the underlying SAT-solver.

There are three main contributions of the paper. Firstly, we show how a number of similar SAT-instances can be solved incrementally by a very modest modification of a modern Chaff-like SAT-solver [MZ01]. The technique we propose is simpler than previous attempts [WKS01], while still obtaining a performance increase of the same magnitude. Secondly, we demonstrate the incremental technique on *temporal induction* [SSS00], a method of checking safety properties on *finite state machines* (FSM). We show the impact of the incremental approach experimentally, both for proving correctness and for finding counter-examples. Thirdly, we refine the method of ensuring completeness for temporal induction. The standard method works by requiring all states in the induction hypothesis to be *unique*. By a simple analysis of the FSM, we are able to exclude some state-variables from the uniqueness constraints, resulting in stronger requirements. This may exponentially reduce the induction depth needed. We prove that this strengthening is sound. Additionally, we demonstrate a speed-up by adding the unique states requirement dynamically for only those pairs of states where it is needed.

The experiments we have performed with our prototype tool TIP show that many properties can be proven at speeds comparable to mature BDD-based tools such as CADENCE SMV and CMU SMV.

2 Preliminaries

In this paper, we consider *safety properties* on *finite state machines* (FSM). The states of the FSM are vectors of booleans, defining the values of the *state variables*. We assume the FSM to have a set of legal *initial states*, and the safety property to be specified as a propositional formula over the state variables. By *reachable state space* we mean all states of the FSM reachable from the initial states. Our task is to prove that the property holds for each state in the reachable state space.

In a standard manner, we will assume the transitions of the FSM to be represented by a propositional formula $\mathbf{T}(\boldsymbol{s}, \boldsymbol{s'})$, the set of initial states by a formula $\mathbf{I}(\boldsymbol{s})$, and further denote the safety property by $\mathbf{P}(\boldsymbol{s})$. We will use $\boldsymbol{s_n}$ to denote the state variables of time step n and introduce the shorthand notation $\mathbf{I}_n$, $\mathbf{P}_n$, and $\mathbf{T}_n$ for $\mathbf{I}(\boldsymbol{s_n})$, $\mathbf{P}(\boldsymbol{s_n})$, and $\mathbf{T}(\boldsymbol{s_n}, \boldsymbol{s_{n+1}})$.

2.1 The SAT problem

Let *Bool* denote the *boolean* domain $\{0, 1\}$, and $\mathit{Vars} := \{x_0, x_1, x_2, \ldots\}$ be a finite set of boolean variables. A *literal* is a boolean variable x_i or a negated boolean variable $\overline{x_i}$. A *clause* is a set of literals, implicitly disjoined. A *SAT instance* is a set of clauses, implicitly conjoined. A *valuation* is a function $\mathit{Vars} \to \mathit{Bool}$. A literal x_i is said to be satisfied by a valuation if its variable is mapped to 1; a literal $\overline{x_i}$ if its variable is mapped to 0. A clause is said to be satisfied if at least one of its literals is satisfied. A *model* (satisfying assignment) for a SAT instance is a valuation where all clauses are satisfied. The *SAT problem* is to find a model for a given set of clauses.

Converting formulas to SAT. There are several ways of translating a propositional formula into clauses, in such a way that satisfiability is preserved. This is typically done by introducing auxiliary variables giving names to some or all subformulas, then generating clauses that establish a definitional relation between the introduced variables and the truth-values of their respective subformulas. Any model for the translated problem (which contains more variables) has the property that its restriction to the original set of variables yields a model for the original formula. We assume the existence of such a translation technique and introduce the following notation:

> **Definition.** By $[\varphi]^p$ we denote a set of clauses defining φ such that p is the literal representing the truth-value of the whole formula. We call p the *definition literal* of φ. Further, we write $[\varphi]$ as a short hand for $[\varphi]^p \cup \{p\}$.

For example $[x \wedge y]^p$ may be translated into the clauses $\{ \{\overline{p}, x\}, \{\overline{p}, y\}, \{p, \overline{x}, \overline{y}\} \}$.

2.2 Temporal Induction

This section briefly summarizes the verification technique *temporal induction* presented in [SSS00].[1] The word "temporal" suggests that the induction is car-

[1] The authors use only the word "induction" in this presentation, but have later adopted the term "temporal induction" and used it in other contexts.

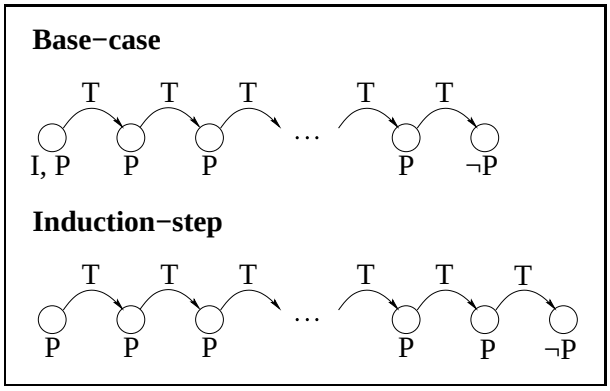


Fig. 1. If the n-th *base-case* is unsatisfiable, it should be read as "There exists no n-step path to a state violating the property, assuming the property holds the first $n-1$ steps." If the n-th *induction-step* is unsatisfiable, it should be read as "Following an n-step trace where the property holds, there exists no next state where it fails".

ried out over the time steps of the FSM. Like a standard induction proof, a temporal induction proof consists of two parts: the base-case and the induction-step. In its simplest form, the base-case states that the property should hold in the initial states; and the induction-step states that the property should be preserved by the transitions of the FSM. Expressing the two parts of the induction proof as SAT-problems is straight-forward—still, the resulting method is already an interesting complement to BDD-based verification methods, especially for systems where the transition relation has no succinct BDD-representation. However, the method is not complete, since the induction-step might not be provable even though the property is true.

To make the method complete, the induction-step is strengthened in two ways. Firstly, the property is assumed to hold for a path of n successive states, rather than just one. This means that a longer base-case must be proven. Secondly, the states of the path are assumed to be unique. It follows immediately from finiteness that the second strengthening makes the method complete in the sense that there is always a length for which the induction-step is provable. Soundness is treated in detail in section 4. Let us formalize the strengthened induction by defining the following formulas:

$$\mathbf{Base}_n := \mathbf{I}_0 \wedge \Big((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \ldots \wedge (\mathbf{P}_{n-1} \wedge \mathbf{T}_{n-1})\Big) \wedge \overline{\mathbf{P}_n}$$

$$\mathbf{Step}_n := \Big((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \ldots \wedge (\mathbf{P}_n \wedge \mathbf{T}_n)\Big) \wedge \overline{\mathbf{P}_{n+1}}$$

$$\mathbf{Unique}_n := \bigwedge_{i \le j \le n} (\boldsymbol{s_i} \neq \boldsymbol{s_{j+1}}) = \bigwedge_{i \le j \le n} \bigvee_k \neg(s_{i,k} \leftrightarrow s_{j,k})$$

An interpretation of these formulas is depicted in *Fig. 1.* Note that when proving correctness we show that the formulas are *unsatisfiable.* In the base-case we assume that all shorter base-cases have been proved already, and add the property

to each state as this tends to make the resulting SAT-problem easier. With these definitions, we can now state an algorithm that intertwines looking for bugs of longer and longer lengths, and trying to prove the property by deeper and deeper induction-steps:

Algorithm 1. "Temporal Induction".

for n $\in 0..\infty$ **do**
  if ($satisfiable([\mathbf{Base}_n])$)
    return PROPERTY FAILS
  if ($\neg satisfiable([\mathbf{Step}_n] \cup [\mathbf{Unique}_n])$)
    return PROPERTY HOLDS

Variations of this algorithm are also meaningful. For instance, checking only the base-case gives a pure bug-hunting algorithm, which delivers counter-examples more quickly. By altering the formula of the base-case slightly, it is possible to start at a higher n and taking bigger leaps than 1. Checking every size of n may be unnecessarily costly. If the bug or proof is deep, taking bigger leaps means solving fewer SAT-problems. However, if there is a bug, *Algorithm 1* (as stated) will always find a shortest counter-example. This may be important. In the remainder of the article, we will show how the cost of incrementing n by only 1 can be greatly reduced by solving the SAT-problems incrementally.

3 Incremental SAT

A typical stand-alone SAT-solver accepts a problem instance as input, solves it, and outputs a model or an "Unsatisfiable" statement as result. This can be inadequate if you wish to solve many similar SAT-instances. The most obvious overhead is re-parsing the (almost) same clause set over and over again. But more importantly, the same, often expensive, inferences may be carried out over and over again. Equipping the SAT-solver with an interface that allows the next SAT-instance to be specified incrementally from the current (solved) instance will certainly remove the parsing problem, but may reduce the number of inferences too.

We focus on the type of solver introduced by [MS99], based on *conflict analysis* and *clause recording*.[2] Such a solver implements a DPLL-style backtracking search procedure [DLL62]. The idea behind augmenting the basic procedure with conflict analysis is that for every conflict detected during the search, some effort is spent on finding a *reason* for the conflict that can be encoded as a clause and added to the clause set. The *recorded* clauses will serve as a cache for the same type of conflicts in later parts of the search-space. For example, if assuming x and y to be true led to a conflict, the clause $\{\overline{x}, \overline{y}\}$ may be recorded. Assuming either x or y to be true in some later part of the search-tree, will immediately give the implied value to the other variable, avoiding repetition of the possibly

[2] This includes SAT-solvers such as: GRASP, SATO, ZCHAFF, LIMMAT, BERKMIN, and the authors' own solvers SATNIK and SATZOO.

lengthy derivation. The effectiveness of this idea has been empirically established by many authors. A motivation for incremental SAT is that the recorded clauses may not only be useful in later parts of the search-tree of the *same* SAT-instance, but also in a later *similar* SAT-instance.

To describe the different design issues encountered when implementing an incremental SAT-system, we adopt an object-oriented view, using a *solver object* which stores the *problem clauses* (the current SAT-instance) as well as the *learnt clauses* (the recorded clauses). The solver has methods for modifying and solving the current SAT-instance. The simplest imaginable interface would contain the following methods:

addClause (**Clause** c) – *will add a clause to the clause database.*
solve – *will solve the current instance.*

Using this interface, the user is allowed to add clauses until he has specified the first SAT-problem. He can then use *solve* to check if the problem is satisfiable or not. If it is, he may add more clauses to constrain the problem further and re-run *solve*. This procedure can be repeated until all SAT instances of interest have been solved. Typically the last instance is unsatisfiable, from which point no extension can be satisfiable.

This approach to incremental SAT, introduced in [Hok93], is limited as the user can never remove anything added. Many interesting incremental SAT-problems requires some form of clause removal. Therefore [WKS01] suggested the following interface to the solver:

addClause (**Clause** c)
removeClause (**Clause** c) – *will remove an existing clause from the clause database.*
solve

By this interface, any set of related problems can be solved incrementally. However, the ability to remove clauses clashes with conflict clause recording. The conflict analysis is guaranteed to produce clauses that are implied by the problem clause set; thus adding these clauses can never cause unsoundness. But removing problem clauses may suddenly render recorded clauses invalid. A detailed dependency analysis must therefore be carried out to remove the invalid clauses, which in turn may require extra book-keeping during the actual solving process. For a longer treatment of this approach see [WKS01].

In contrast, we propose the following interface which only enables the removal of unit clauses. The motivation is that it is *very* simple to implement (5 lines of code in our solver), while being expressive enough to encompass several interesting incremental SAT-problems not expressible by the original interface:

addClause (**Clause** c)
solve (**list**⟨**Literal**⟩ assumptions)

The extra list of literals passed to *solve* should be viewed as unit clauses to be added during this particular solving, then removed upon return from the solver. The reason that this approach is simpler is that *all* learned clauses are safe to

keep, and thus no extra book-keeping is needed. To see why it is safe, note that the extra unit clauses can be seen (and implemented) as internal assumptions by the search procedure, and that it is an inherent property of conflict clauses that they are independent of the assumptions under which they occur. [3]

4 Incremental Induction

In section 2.2 we saw a straight-forward algorithm for proving or disproving safety properties by induction. We break this algorithm into two parts, the *base-case* ("bug-finder") and the *induction-step* ("upper-bound prover"), and show how they can be implemented incrementally using the SAT-interface of section 3.

Algorithm 2 "Extending base".

addClauses($[\mathbf{I}_0]$)
for n $\in$ 0..∞ **do**
  addClauses($[\mathbf{P}_n]^{p_n}$)
  solve($\{\overline{p_n}\}$)
  if (SATISFIABLE)
    return PROPERTY FAILS
  addClause($\{p_n\}$)
  addClauses($[\mathbf{T}_n]$)

Algorithm 3 "Extending step".

addClauses($[\overline{\mathbf{P}_0}]$)
for n $\in$ -1..$-\infty$ **do**
  solve($\{\}$)
  if (UNSATISFIABLE)
    return IND. STEP HOLDS
  addClauses($[\mathbf{T}_n]$)
  addClauses($[\mathbf{P}_n]$)
  for i $\in$ 0..n+1 **do**
    addClauses($[\boldsymbol{s_i} \neq \boldsymbol{s_n}]$)

A first observation on these algorithms is that they build the trace of states related by the transition relation in different directions (n is decremented in the step). Growing the trace forwards in the base-case allows us to keep the often strong formula $\mathbf{I}_0$ fixed in the SAT-solver. Building the trace in the opposite direction would force us to put the initial state constraints as an assumption literal to "*solve*", which will have the undesirable effect of making any recorded conflict clause depending on the initial state ineffective in successive iterations. Similarly in the step, growing the trace backwards makes it unnecessary to use any assumption literal at all, which again promotes reuse of recorded clauses between iterations.

Different top-level strategies for how to combine the two algorithms to a safety-checking procedure are possible. To emulate *Algorithm 1* of section 2.2, the algorithms could be run in parallel, each with its own solver instance. As soon as the induction-step succeeds for a particular length, an unsatisfiable base-case of that length will constitute a proof of the safety property. However, it is also possible to mix the two algorithms into one. We will then have to break the natural direction of building the trace for either the base-case or the induction-step. We arbitrarily chose to sacrifice the induction-step.

[3] In fact, the more general interface can be simulated to a large extent. By inserting the clause $\{x\} \cup C$, and passing $\overline{x}$ as an assumption literal, we achieve the same effect as inserting C. Asserting x to be true afterwards will make the clause true forever, and it will be removed from the clause database by the top-level simplification procedure of the solver.

Algorithm 4 "Zig-zag".

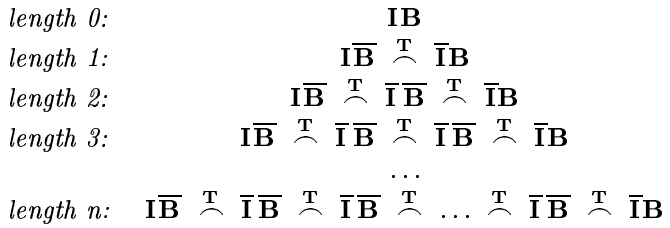
```
addClauses([I_0]^z)                    – z is the definition literal for I_0
for n ∈ 0..∞ do
    addClauses([P_n]^{p_n})            – p_n is the definition literal for P_n
    solve({¬p_n})                      – step: do not include I_0
    if (UNSATISFIABLE)                 – P_n must hold!
        return PROPERTY HOLDS
    solve({z, ¬p_n})                   – base-case: include I_0
    if (SATISFIABLE)                   – counter-example found!
        return PROPERTY FAILS
    addClause({p_n})                   – assert P_n from now on
    addClauses([T_n])                  – assert transition from s_n to s_{n+1}
    for i ∈ 0..n-1 do                  – add uniqueness constraints
        addClauses([s_i ≠ s_n])
```

The reason for stating this algorithm is partly to show that there is many possible ways of encoding the safety-checking procedure incrementally. With this algorithm, the SAT-solver is allowed to share conflict clauses between the base-case and the induction-step, which may be beneficial. We include the algorithm in our benchmark section.

4.1 Discussion

We will now try to draw a map over possible induction based safety-checking algorithms. Let us use the term *bad state* for a state were the safety property does not hold. It is generally observed that checking safety properties is symmetric with respect to the initial states and the bad states. Everything presented up to this point could have been carried out backwards, with the roles of initial states and bad states exchanged, and the transition relation inverted. We are going to adopt this symmetrical view from now on.

In this view, we regard the induction-step as a method of finding an upper bound on the length of a shortest counter-example, and the base-case as a way of producing the counter-example. Now, what must a shortest counter-example look like? It has to start in an initial state, it has to end up in a bad state, and the states in between must not be either initial or bad (otherwise it could not be a shortest counter-example). Using $\mathbf{B}$ (bad) for $\overline{\mathbf{P}}$ we can view the set of possible shortest counter-examples pictorially:



Each line depicting a (shortest) counter-example corresponds to a conjunction of constraints ($\mathbf{I}_0 \wedge \mathbf{T}_0 \wedge \overline{\mathbf{B}_1} \wedge \overline{\mathbf{I}_1} \wedge \mathbf{T}_1 \wedge \ldots$). There is a lot of sharing between the counter-examples of different lengths, and indeed if we remove either the initial $\mathbf{I}$ or the final $\mathbf{B}$ from the n-th counter example, i.e.:

$$(1) \qquad \overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B}$$

$$\textit{or} \quad (2) \qquad \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}$$

then any counter-example of length n or *longer* will include all the constraints of (1) and (2). This means that if either the constraints of (1) or (2), or any *subset* of these, yields an unsatisfiable problem, then so will *all* possible shortest counter-examples of longer lengths. Thus we have found an upper bound on the shortest counter-example.

The picture above does not contain all constraints derivable from the fact that we are considering a *shortest* counter-example. We can further conclude:

1. Between no two states is there a shorter path.

or weaker 2. Between no two non-neighbors is there a transition (and the last state is unique).

or weaker 3. No two states are the same.

Any of these facts can be used when proving an upper bound. As long as we keep adding constraints that must be fulfilled by shortest counter-examples, any contradiction reached means we have established an upper bound. The reason for stating weaker versions of the shortest-path requirement is that these versions can be implemented more efficiently. Furthermore, we have already noted that the third condition is enough to make the procedure complete. In the next section we describe how the implementation of this condition can be improved.

Taking this subset-of-counter-example view, the induction-step we have used in our algorithms can now be viewed as selecting the subset of (1) not containing any $\overline{\mathbf{I}}$:s but including the uniqueness constraints dictated by condition 3. [4] Through experiments we found that this choice worked well in practice.

Finding a counter-example. If the user knows or has reason to believe that the property is false, he may want to run just the base-case to quickly produce a counter-example. In this case, it is less clear if any extra constraints should be added to the trace. In *Algorithm 1* and *2* we chose to add $\mathbf{P}$. More constraints mean more clauses in the solver, which leads to slower propagation, but also to a smaller search-tree. Which of the two effects is predominant in a particular case is hard to judge. In general, adding weak constraints is seldom a good idea.

Present BMC tools can optionally produce a SAT-problem stating that the property fails among the first n steps rather than after exactly n steps. Care must be taken before adding extra constraints to such formulations. For instance, one can no longer require the states to be unique. One must also assume (or modify)

[4] The *recurrence diameter* introduced in [BCCZ99] can similarly be viewed as the subset containing only the $\mathbf{T}$:s together with uniqueness constraints.

the transition relation to always have a next state; or risk getting an unsatisfiable problem due to deadlock, even in the presence of a bug. A comparison between this "one-shot" method and the incremental base-case is included in our experiments.

4.2 Improving the Unique States Requirement

The uniqueness constraints described in section 2.2 and used in *Algorithm 1, 3* and *4* require each pair of states to be different. These requirements are *statically* added, and their number will grow quadratically in the length of the induction-step. For problems requiring high induction length, there is a risk of adding numerous possibly superfluous constraints that will tax the SAT-solver heavily. We propose a *dynamic* approach where the models returned by the solver in the induction-step are examined, and only if two states are actually equal, a constraint stating that they should be different is added. The solver must then be run again, which may possibly cost more than adding superfluous constraints, but hopefully the incrementality of the approach means that any re-run is very quick. We verified experimentally that the method indeed seems to perform better in general.

A question that has not been treated sufficiently in earlier presentations on induction is what variables should be included in the uniqueness constraints. It is not unusual to describe the FSM in the form of a sequential circuit. The standard interpretation of a circuit is to consider both the latches (the state holding elements) and the inputs as *state variables* of the FSM. However, it is fairly clear that there is no need to include inputs in the uniqueness constraints. If two states are equal except for the inputs, whatever value the inputs assume in the second state, they could have assumed in the first. It is therefore safe to require only the latch-variables do be different—a much stronger condition. In fact, this is often what is implemented [CS00]. Note that failing to remove the superfluous state variables from the uniqueness constraints gives an ineffective induction algorithm, as each extra state variable has the potential of doubling the depth needed to prove the step.

If on the other hand the FSM is given as two propositional formulas $\mathbf{I}$ and $\mathbf{T}$ it is less clear what variables can be excluded.[5] We propose the following solution:

1. Include only variables occurring *both* in the current and the next state of the transition relation.
2. Do not add uniqueness constraints including the first or the last state of the trace.

We refer to uniqueness constraints over this reduced set of state variables as *strong uniqueness.*

[5] The result of parsing an SMV file often leaves you with just this.

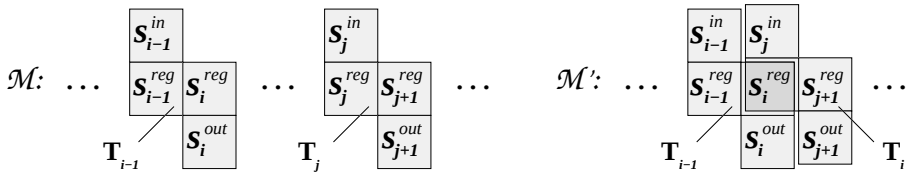


Fig. 2. The picture shows the contraction of the counter-example $\mathcal{M}$ to $\mathcal{M}'$. The state variables constrained by the transition relations at the point of "gluing" are printed in the boxes; the remaining trace is represented by the "...".

Correctness. We will now prove that temporal induction with strong uniqueness is sound. Recall that the induction-step can be strengthened by anything that holds for a shortest counter-example. It then suffices to show that a counter-example that is not strongly unique cannot be shortest. Let us introduce the following notation:

$$\begin{array}{ll} \boldsymbol{s_i}^{left} := vars(\mathbf{T}_i) \cap \boldsymbol{s_i} & \boldsymbol{s_i}^{in} := \boldsymbol{s_i}^{left} \setminus \boldsymbol{s_i}^{right} \\ \boldsymbol{s_i}^{right} := vars(\mathbf{T}_{i-1}) \cap \boldsymbol{s_i} & \boldsymbol{s_i}^{out} := \boldsymbol{s_i}^{right} \setminus \boldsymbol{s_i}^{left} \\ & \boldsymbol{s_i}^{reg} := \boldsymbol{s_i}^{left} \cap \boldsymbol{s_i}^{right} \end{array}$$

Let $\mathcal{M}$ be the model of a formula encoding a counter-example of depth n:

$$\mathcal{M} \models \mathbf{I}_0 \wedge \mathbf{T}_0 \wedge \mathbf{T}_1 \wedge \ldots \wedge \mathbf{T}_{n-1} \wedge \mathbf{B}_n.$$

We now show by construction that if $\mathcal{M} \models (\boldsymbol{s_i}^{reg} = \boldsymbol{s_j}^{reg})$ for some $0 < i < j < n$ ($\mathcal{M}$ is not strongly unique) then there is a shorter counter-example. Define $\mathcal{M}'$ over $\{\boldsymbol{s_0}, \ldots, \boldsymbol{s_{n-(j-i)}}\}$ as follows:

$$\begin{array}{lll} \mathcal{M}'(\boldsymbol{s_k}) & = \mathcal{M}(\boldsymbol{s_k}) & ,k < i \\ \mathcal{M}'(\boldsymbol{s_k}) & = \mathcal{M}(\boldsymbol{s_{k+(j-i)}}) & ,k > i \\ \mathcal{M}'(\boldsymbol{s_i}^{in}) & = \mathcal{M}(\boldsymbol{s_j}^{in}) & \\ \mathcal{M}'(\boldsymbol{s_i}^{out}) & = \mathcal{M}(\boldsymbol{s_i}^{out}) & \\ \mathcal{M}'(\boldsymbol{s_i}^{reg}) & = \mathcal{M}(\boldsymbol{s_i}^{reg}) & \end{array}$$

$\mathcal{M}'$ now constitutes a counter-example of depth $n-(j-i)$. We have contracted the counter-example by simply removing all states between i and j (depicted in *Fig. 2*). The only potential problem lies in the "gluing" of the head and the tail at state i. However, the only constraints containing $\boldsymbol{s_i}$ are $\mathbf{T}_{i-1}$ and $\mathbf{T}_i$. But $\mathbf{T}_{i-1}$ does not contain any variables from $\boldsymbol{s_i}^{in}$, so letting $\mathcal{M}(\boldsymbol{s_i}^{in}) \neq \mathcal{M}'(\boldsymbol{s_i}^{in})$ cannot make $\mathbf{T}_{i-1}$ false in $\mathcal{M}'$. Similarly for $\mathbf{T}_i$ which does not contain any variables from $\boldsymbol{s_i}^{out}$. Finally $\mathcal{M}(\boldsymbol{s_i}^{reg}) = \mathcal{M}(\boldsymbol{s_j}^{reg})$, so indeed $\mathcal{M}'$ must be a model for the constraints $\mathbf{T}_{i-1}$ and $\mathbf{T}_i$. □

The proof can easily be extended to establish that the exclusion of the first and the last state is superfluous if all variables of $\mathbf{I}$ occur in the next state of $\mathbf{T}$ and all variables of $\mathbf{B}$ occur in the current state of $\mathbf{T}$.

5 Experimental Results

The ideas presented in this paper were implemented in the prototype tool TIP[6] which was integrated with the SAT-solver SATZOO. All benchmarks were performed on a 2 GHz Pentium 4 with 512 MB of memory running Linux. We set the time-out for all launches to 10 minutes, and the memory limit to 400 MB. The benchmarks were collected from several sources. In the tables, each benchmark name is tagged with the source of the problem:

cadence	–	Example files from the CADENCE SMV distribution.
cmu	–	Example files from the CMU SMV distribution.
ken	–	SMV case studies from Ken McMillan's web-page.
nusmv	–	Example files from the NUSMV distribution.
vis	–	Example files from the VIS distribution.
texas	–	The *Texas 97 benchmarks* available from Berkeley University.
eijk	–	ISCAS'89 sequential equivalence checking from [Eijk98].
irst	–	Problems from the Model Checking Group at IRST.

All problems were converted to flat SMV-format with only boolean variables and no sub-modules. For each problem, the safety properties were extracted. In this process, CTL formulas "EF" were changed into "AG¬" and all fairness constraints were removed. Different properties for the same system are indicated by a subscript after the system name.

Counting each property as a separate instance, a total of 185 problem instances were collected. As our first experiment, we ran TIP, CADENCE SMV, CMU SMV, and NUSMV on each of these instances. All tools were run with a default set of options, providing no problem specific variable ordering:

`Tip` *filename*
`CadSMV` *filename*
`CmuSMV -reorder` *filename*
`NuSMV -AG -dynamic -coi` *filename*

Instances solved in less than 1 second by all tools were considered trivial and removed, leaving 158 instances.

Comparison with BDD-tools. The result of the comparative experiment is presented in *Table 1.* The default strategy of TIP runs the base-case and the induction-step presented in *Algorithm 2* and *3* in parallel, each with its own solver instance. The two algorithms are given equal amount of CPU time, until the point where either the base-case fails, and a counter-example is found, or the induction-step is proven, and the remaining base-cases (if any) are proved with 100% CPU.

The purpose of the experiment was to relate the performance of induction to industrially applied methods, and to show the (lack of) correlation between hardness for BDD-based methods and hardness for induction-based methods.

[6] The tool TIP, the SAT-solver SATZOO and all benchmarks used in this article can be downloaded from *http://www.cs.chalmers.se/~een/*

TIP was able to solve 6 instances where BDD-based verification failed, showing that induction may be a valuable complementary method.[7]

Effect of incrementality. The second experiment we performed was a comparison of *Algorithm 2* and *3* using the incremental interface of SATZOO and using SATZOO as an external solver. In this experiment, we used only problem instances where the property held. The result is presented in *Table 2.*

The experiment establishes a substantial speed-up by the incremental approach. Unsurprisingly, the gain was larger for instances where a long induction-step was needed to prove the property.

From the table we can also see that the induction-step usually takes longer to prove than the base-case. We observed the same behavior for instances where the property failed (although not presented here). This is the reason the default strategy of TIP does not increase the lengths of the step and base evenly, but instead devotes the same amount of CPU to each. Otherwise, bugs may not be found due to hard (and futile) induction-steps.

One solver instance or two. The third experiment compared *Algorithm 4* ("Zig-Zag") using one solver instance to running the induction-step and the base-case in separate solver instances. ("Dual"). In this experiment, the step and the base were incremented evenly so that both methods would solve only the minimal number of SAT-instances. We also include the standard implementation of (complete) induction as presented in [SSS00]. The results are also in *Table 2.*

The experiment suggests that separate solver instances for the base and the step is favorable. From the table we can also see that the incremental implementation of induction clearly outperforms the standard implementation.

BMC Comparison. In the fourth experiment, we compared incremental search for counter-example to the "one-shot" approach described in section 4.1. The result is presented in *Table 3.* The experiment shows that often you must know the exact length of a shortest counter-example for the one-shot method to be advantageous.

Uniqueness constraints. In the final experiment, we studied the effect of adding uniqueness constraints dynamically and statically, including both instances where the constraints must be added, and instances which are provable without uniqueness constraints. The result is presented in *Table 4.*

The effect of sharpening the constraints by removing variables are not presented, as it is clearly advantageous. A study of the "*eijk*" equivalence checking problems, where 9 out of 13 need uniqueness constraints, showed that *none* of these could be solved within the time-bound without using the sharpening.

[7] These problems were all "TCAS II" problems from the NUSMV distribution, originally used in "Model Checking Large Software Specifications" [CAB98].

Tool	Solved (of 158)	Alone in solving
CADENCE SMV	131	5
TIP	92	6
CMU-SMV	90	0
NUSMV	73	0

Table 1. *Tool comparison.* The left column shows the total number of solved instances within 10 minutes. The right column show how many of these instances no other tool could solve. CADENCE SMV excelled by proving 22 instances that neither of the two other SMVs could prove, and 39 more instances than TIP. Still only 5 instances were unique, as TIP solved many of the problems where NUSMV and CMU-SMV failed, plus 6 that CADENCE SMV did not solve.

Name	Len	Step^{inc}	Step^{ext}	Base^{inc}	Base^{ext}	Dual	ZigZag	StdInd
cmu:periodic	97	70.7	[>600]	10.7	141.8	80.9	[>600]	[>600]
eijk:S208c	259	448.0	[>600]	[>600]	[>600]	[>600]	[>600]	[>600]
eijk:S208o	258	483.2	[>600]	[>600]	[>600]	[>600]	564.2	[>600]
eijk:S208	259	436.7	[>600]	[>600]	[>600]	[>600]	503.7	[>600]
eijk:S298	59	27.7	[>600]	34.9	96.2	62.9	316.1	[>600]
eijk:S510	11	5.2	8.0	0.5	0.9	5.9	7.4	10.1
eijk:S820	12	6.1	22.9	6.4	12.5	12.6	20.2	30.1
eijk:S832	12	7.6	28.2	5.8	12.9	13.4	25.1	35.2
eijk:S953	8	1.7	4.2	0.1	0.2	1.9	4.2	4.4
ken:oop$_1$	30	39.4	[>600]	0.3	7.4	39.9	492.0	254.0
nusmv:guidance$_1$	11	2.8	10.2	0.8	3.4	3.5	3.9	11.1
nusmv:guidance$_7$	28	120.3	[>600]	315.0	[>600]	438.9	[>600]	[>600]
nusmv:tcas$_2$	7	1.3	3.1	0.2	0.3	1.5	1.9	4.3
nusmv:tcas$_3$	6	1.3	3.3	0.0	0.1	1.3	1.8	3.2
texas:parsesys$_2$	4	12.2	13.5	0.2	0.2	14.7	12.5	7.8
vis:prodcell$_{12}$	30	256.6	[>600]	112.8	445.5	367.3	[>600]	[>600]
vis:prodcell$_{13}$	9	4.6	12.4	0.1	0.6	4.8	3.7	14.7
vis:prodcell$_{14}$	17	31.3	185.1	7.3	14.2	38.7	52.3	219.9
vis:prodcell$_{15}$	24	109.3	[>600]	23.0	80.1	132.4	216.7	[>600]
vis:prodcell$_{16}$	6	2.1	4.1	0.0	0.1	2.1	1.2	4.7
vis:prodcell$_{17}$	28	211.3	[>600]	52.4	277.5	265.0	[>600]	[>600]
vis:prodcell$_{18}$	14	21.4	117.9	0.4	3.2	21.8	28.6	128.9
vis:prodcell$_{19}$	23	61.6	457.0	23.4	86.0	85.0	178.5	[>600]
vis:prodcell$_{24}$	38	391.9	[>600]	[>600]	[>600]	[>600]	[>600]	[>600]

Table 2. *Experimental results for the effect of incremental SAT vs. external SAT.* All times are in seconds. The experiment includes all instances where the property was proved to hold in in the first experiment. Launches where all methods took less than 3 seconds have been left out. "Dual" stands for running one iteration of *Alg.2* and *Alg.3* interchangeably; "ZigZag" refers to *Alg.4*; "StdInd" stands for standard induction with all uniqueness constraints statically added and using an external SAT-solver.

Name	Length	Incremental BMC	Perfect Guess	25%-off Guess
nusmv:tcas$_1$	11	3.6	3.7	5.0
nusmv:tcas$_4$	15	9.7	9.7	18.2
nusmv:tcas$_5$	24	48.7	40.1	125.2
nusmv:tcas$_6$	17	13.6	13.5	38.2
texas:parsesys$_1$	10	9.3	0.8	1.1
texas:parsesys$_3$	9	3.3	0.7	0.9
texas:two-proc$_2$	16	4.7	1.0	2.9
texas:two-proc$_4$	20	20.9	1.8	9.1
vis:eisenberg	20	20.7	18.1	79.1

Table 3. *Experimental result for incremental BMC vs. SAT-instances of fixed length.* All times are in seconds. "Perfect Guess" means the SAT-instance encode "there is a bug of length $\leq k$" where k is the length of the shortest counter-example. "25%-off" means k is multiplied by 1.25. Launches where all methods took less than 3 seconds have been left out.

Name	Len	Time^d	Time^s	Ban^d	Ban^s	Clau^d	Clau^s	Conf^d	Conf^s
cmu:periodic	97	70.7	120.4	0	4656	455k	908k	15k	14k
eijk:S208	259	436.7	[>600]	258	[>20000]	186k	-	76k	-
eijk:S298	59	27.7	66.6	114	1653	69k	296k	24k	25k
ken:oop$_1$	30	39.4	50.4	113	406	67k	101k	32k	30k
nusmv:guidance$_7$	28	120.3	66.9	0	378	151k	276k	56k	28k
vis:prodcell$_{12}$	30	256.6	252.7	0	406	346k	439k	48k	43k
vis:prodcell$_{14}$	17	31.3	41.7	0	120	189k	217k	11k	13k
vis:prodcell$_{15}$	24	109.3	134.3	0	253	273k	330k	29k	29k
vis:prodcell$_{17}$	28	211.3	253.6	0	351	322k	400k	45k	46k
vis:prodcell$_{18}$	14	21.4	25.5	0	78	153k	171k	10k	10k
vis:prodcell$_{19}$	23	61.6	71.9	0	231	260k	311k	18k	18k
vis:prodcell$_{24}$	38	391.9	490.1	0	666	440k	588k	60k	61k

Table 4. *Experimental results for dynamic vs. static uniqueness constraints in the induction-step.* All times are in seconds. Launches taking less than 10 seconds or having shorter length than 5 has been left out. A superscript "d" means dynamic (on demand) adding of uniqueness constraints. A superscript "s" means static adding of uniqueness constraints between all pairs of states. "Ban" is the number of constraints added (banning two states from being equal). "Clau" is the final number of clauses in the solver. "Conf" is the total number of conflicts in the search-tree of the solver. Only three problems actually needed uniqueness constraints to be provable, and in almost all other cases it incurred a cost to add them. For the three cases where the constraints were necessary, adding them dynamically lead to a speed-up. Without uniqueness constraints these three problem are not provable by induction. The dynamic method thus saves the user from guessing for each problem if uniqueness constraints should be used or not without incurring any extra cost.

6 Related Work

Incremental BMC was independently introduced by Ofer Strichman in [Stri01] and Sakallah et. al. in [WKS01]. Our approach differs from previous attempts in that we keep all clauses from previous iterations (including conflict clauses). Moreover, we complete the method with incremental temporal induction. Strichman's work further includes several techniques to enhance the SAT-solving of BMC problems, including *internal constraints replication* for copying invariant conflict clauses between the time steps of the trace, and BMC specific variable decision strategies [Stri00].

Related techniques for proving upper bounds for BMC are presented in [KS03] (computing the recurrence diameter) and [BKA02] (approximating the diameter by structural analysis). In particular, the authors of [KS03] suggest another solution to the quadratic blow-up of uniqueness constraints by adding a sorting network for the state variables to the SAT-problem.

7 Conclusions

Temporal induction has been used before to prove upper bounds for BMC [SSS00]. In these efforts, the authors established it too costly to gradually increase the depth of the induction proof using an external SAT-solver. We have shown that integrating the SAT-solver and the induction procedure overcomes this cost. Furthermore, we sharpened the unique-states constraints by a syntactic analysis on the transition relation; an improvement that was absolutely necessary for many of our benchmarks to go through.

By extensive testing we further reinforced the view that induction is an important complement to BDD-based methods for safety-checking. The combination of techniques presented in this paper results in what the authors believe to be the first efficient and complete induction based checker produced by academia. Enabled by the incremental SAT-interface, we explored an online method of adding uniqueness constraints on demand. To a large extent the method saves the user from deciding manually whether or not to add these constraints, making temporal induction a more push-button technique.

As a side-effect of implementing temporal induction incrementally, we got an incremental BMC for safety properties. The efforts on incremental BMC by [Stri01,WKS01] was based on extensive adaptation of the underlaying SAT-solver. We have shown that results of the same magnitude can be achieved by a much smaller modification of the solver. A standard way of applying BMC is to generate a single SAT-problem encoding the presence of a bug within k time steps. We have compared this method to iterating up to k incrementally and found that the incremental approach was faster in most cases, even if k was specified as close as 25% above the length of a shortest counter-example.

8 Future Work

The single most significant factor for the success of temporal induction is the induction depth needed. We therefore believe the most important direction of

research is towards methods of automatically strengthening the induction-step in order to reduce this depth. A successful method achieving this was presented in [Eijk98,BC00]. It works by finding invariant equivalences or implications between the state variables and internal points. Casting this method into our incremental system looks very promising. Stronger constraints on the shape of a shortest counter-example were suggested in [SSS00], but have not yet been successfully applied. We would like to investigate if a dynamic approach similar to that we used for uniqueness constraints might be helpful.

Finally, there are many possible ways of tuning the SAT-solver to incremental temporal induction. In particular, we wish to explore native uniqueness constraints, as well as the methods presented in [Stri00,Stri01] for specialized variable orderings and constraint replication.

Acknowledgments

We would like to thank Per Bjesse and Mary Sheeran for their careful reading and valuable criticism of the manuscript for this paper.

References

[BC00] P. Bjesse, K. Claessen. **"SAT-based Verification without State Space Traversal"** in *FMCAD 2000*, LNCS:1954, Springer-Verlag.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. **"Symbolic model checking without BDDs"** in *TACAS 1999*, LNCS:1579, Springer.

[BCRZ99] A. Biere, E.M. Clarke, R. Raimi, and Y. Zhu. **"Verifying safety properties of a PowerPC[tm] microprocessor using symbolic model checking without BDDs"** in *CAV 1999*, LNCS:1633.

[BKA02] J. Baumgartner, A. Kuehlmann, J. Abraham **"Property Checking via Structural Analysis"** in *CAV 2002*, LNCS:2404, Springer-Verlag.

[Bry86] R.E. Bryant. **"Graph-based algorithms for boolean function manipulation"** in *IEEE Trans. on Computers*, C-35(8), Aug. 1986.

[CAB98] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J.D. Reese **"Model Checking Large Software Specifications"** in *IEEE Tran. on Software Engineering* 24(7), Jul. 1998

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M.Y. Vardi. **"Benefits of bounded model checking at an industrial setting"** in *CAV 2001*, LNCS:2102.

[CS00] K. Claessen, M. Sheeran. **"A Tutorial on Lava: A Hardware Description and Verification System"** at *http://www.cs.chalmers .se/~koen/Lava*, 2000

[DLL62] M. Davis, M. Logman, D. Loveland. **"A machine program for theorem proving"** in *Communications of the ACM*, vol 5, 1962.

[Eijk98] C.A.J. van Eijk. **"Sequential equivalence checking without state space traversal"** in *Proc. Conf. on Design, Automation and Test in Europe*, 1998.

[Hok93] J.N. Hooker **"Solving the Incremental Satisfiability Problem"** in *Journal of Logic Programming*, vol 15, 1993.

[KS03] D. Kroening, O. Strichman **"Efficient Computation of Recurrence Diameters"** in *VMCAI 2003* LNCS:2575, Springer-Verlag 2003.

[MS99] J.P. Marques-Silva, K.A. Sakallah. **"GRASP: A Search Algorithm for Propositional Satisfiability"** in *IEEE Transactions on Computers*, vol 48, 1999.

[MZ01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik **"Chaff: Engineering an Efficient SAT Solver"** in *DAC 2001*.

[SSS00] M. Sheeran, S. Singh, G. Stålmarck. **"Checking safety properties using induction and a SAT-solver"** in *FMCAD 2000*, LNCS:1954.

[Stri00] O. Strichman **"Tuning SAT checkers for Bounded Model Checking"** in *CAV 2000*, LNCS:1855, Springer-Verlag.

[Stri01] O. Strichman **"Pruning techniques for the SAT-based Bounded Model Checking Problem"** in *Proc. 11th Advanced Research Working Conf. on Correct Hardware Design and Verification Methods*, 2001.

[WKS01] J. Whittemore, J. Kim, K. Sakallah. **"SATIRE: A New Incremental Satisfiability Engine"** in *DAC 2001*, ACM Press.

Paper 3

An Extensible SAT-solver

Niklas Eén, Niklas Sörensson

Synopsis

This paper shows you how to code a state-of-the-art SAT-solver (as of 2003).

The paper was published in the *SAT 2003* proceedings. The material is based on the authors' SAT-solvers SATZOO and SATNIK. The solver SATZOO (by Niklas Eén) won two of the six categories in the *SAT competition 2003*. For more information, see *http://www.lri.fr/~simon/contest03/results/*.

An Extensible SAT-solver

Niklas Eén, Niklas Sörensson

Chalmers University of Technology, Sweden
{een,nik}@cs.chalmers.se

Abstract. In this article, we present a small, complete, and efficient SAT-solver in the style of conflict-driven learning, as exemplified by CHAFF. We aim to give sufficient details about implementation to enable the reader to construct his or her own solver in a very short time. This will allow *users* of SAT-solvers to make domain specific extensions or adaptions of current state-of-the-art SAT-techniques, to meet the needs of a particular application area. The presented solver is designed with this in mind, and includes among other things a mechanism for adding arbitrary boolean constraints. It also supports solving a series of related SAT-problems efficiently by an incremental SAT-interface.

1 Introduction

The use of SAT-solvers in various applications is on the march. As insight on how to efficiently encode problems into SAT is increasing, a growing number of problem domains are successfully being tackled by SAT-solvers. This is particularly true for the *electronic design automation* (EDA) industry [BCCFZ99,Lar92]. The success is further magnified by current state-of-the-art solvers being extended and adapted to meet the specific characteristics of these problem domains [ARMS02,ES03].

However, modifying an existing solver, even with a thorough understanding of both the problem domain and of modern SAT-techniques, can become a time consuming and bewildering journey into the mysterious inner workings of a ten-thousand-line software package. Likewise, writing a solver from scratch can also be a daunting task, as there are numerous pitfalls hidden in the intricate details of a correct and efficient solver. The problem is that although the *techniques* used in a modern SAT-solver are well documented, the details necessary for an *implementation* have not been adequately presented before.

In the fall of 2002, the authors implemented the solvers SATZOO and SATNIK. In order to sufficiently understand the implementation tricks needed for a modern SAT-solver, it was necessary to consult the source-code of previous implementations.[1] We find that the material contained therein can be made more accessible, which is desirable for the SAT-community. Thus, the principal goal of this article is to bridge the gap between existing descriptions of SAT-techniques and their actual implementation.

We will do this by presenting the code of a minimal SAT-solver MINISAT, based on the ideas for conflict-driven backtracking [MS96], together with watched literals and dynamic variable ordering [MZ01]. The original C++ source code

[1] LIMMAT at `http://www.inf.ethz.ch/personal/biere/projects/limmat/`
ZCHAFF at `http://www.ee.princeton.edu/~chaff/zchaff`

(downloadable from `http://www.cs.chalmers.se/~een`) for MINISAT is under 600 lines (not counting comments), and is the result of rethinking and simplifying the designs of SATZOO and SATNIK without sacrificing efficiency. We will present all the relevant parts of the code in a manner that should be accessible to anyone acquainted with either C++ or Java.

The presented code includes an incremental SAT-interface, which allows for a series of related problems to be solved with potentially huge efficiency gains [ES03]. We also generalize the expressiveness of the SAT-problem formulation by providing a mechanism for arbitrary *constraints* over boolean variables to be defined. Paragraphs discussing implementation alternatives are marked "[Disc]" and can be skipped on a first reading.

From the documentation in this paper we hope it is possible for *you* to implement a fresh SAT-solver in your favorite language, or to grab the C++ version of MINISAT from the net and start modifying it to include new and interesting ideas.

2 Application Programming Interface

We start by presenting MINISAT's external interface, with which a user application can specify and solve SAT-problems. A basic knowledge about SAT is assumed (see for instance [MS96]). The types ***var***, ***lit***, and ***Vec*** for variables, literals, and vectors respectively are explained in detail in section 4.

class *Solver* – *Public interface*

var	*newVar*	()
bool	*addClause*	(***Vec***⟨***lit***⟩ literals)
bool	*add...*	(...)
bool	*simplifyDB*	()
bool	*solve*	(***Vec***⟨***lit***⟩ assumptions)
Vec⟨***bool***⟩	model	– *If found, this vector has the model.*

The "*add...*" method should be understood as a place-holder for additional constraints implemented in an extension of MINISAT.

For a standard SAT-problem, the interface is used in the following way: Variables are introduced by calling *newVar()*. From these variables, clauses are built and added by *addClause()*. Trivial conflicts, such as two unit clauses $\{x\}$ and $\{\overline{x}\}$ being added, can be detected by *addClause()*, in which case it returns FALSE. From this point on, the solver state is undefined and must not be used further. If no such trivial conflict is detected during the clause insertion phase, *solve()* is called with an empty list of assumptions. It returns FALSE if the problem is *unsatisfiable*, and TRUE if it is *satisfiable*, in which case the model can be read from the public vector "model".

The *simplifyDB()* method can be used before calling *solve()* to simplify the set of problem constraints (often called the *constraint database*). In our implementation, *simplifyDB()* will first propagate all unit information, then remove all satisfied constraints. As for *addClause()*, the simplifier can sometimes detect a

conflict, in which case FALSE is returned and the solver state is, again, undefined and must not be used further.

If the solver returns *satisfiable*, new constraints can be added repeatedly to the existing database and *solve()* run again. However, more interesting sequences of SAT-problems can be solved by the use of *unit assumptions*. When passing a non-empty list of assumptions to *solve()*, the solver temporarily assumes the literals to be true. After finding a model or a contradiction, these assumptions are undone, and the solver is returned to a usable state, even when *solve()* return FALSE, which now should be interpreted as *unsatisfiable under assumptions*.

For this to work, calling *simplifyDB()* before *solve()* is no longer optional. It is the mechanism for detecting conflicts independent of the assumptions – referred to as a *top-level* conflict from now on – which puts the solver in an undefined state. We wish to remark that the ability to pass unit assumptions to *solve()* is more powerful than it might appear at first. For an example of its use, see [ES03].

An alternative interface would be for *solve()* to return one of three values: *satisfiable, unsatisfiable,* or *unsatisfiable under assumptions*. This is indeed a less error-prone interface as there is no longer a pre-condition on the use of *solve()*. The current interface, however, represents the smallest modification of a non-incremental SAT-solver. The early non-incremental version of SATZOO was made compliant to the above interface by adding just 5 lines of code. [Disc]

3 Overview of the SAT-solver

This article will treat the popular style of SAT-solvers based on the DPLL algorithm [DLL62], backtracking by conflict analysis and clause recording (also referred to as *learning*) [MS96], and boolean constraint propagation (BCP) using *watched literals* [MZ01].[2] We will refer to this style of solver as a *conflict-driven SAT-solver*.

The components of such a solver, and indeed a more general constraint solver, can be conceptually divided into three categories:

- **Representation.** Somehow the SAT-instance must be represented by internal data structures, as must any derived information.
- **Inference**. Brute force search is seldom good enough on its own. A solver also needs some mechanism for computing and propagating the direct implications of the current state of information.
- **Search**. Inference is almost always combined with search to make the solver complete. The search can be viewed as another way of deriving information.

A standard conflict-driven SAT-solver can represent *clauses* (with two literals or more) and *assignments*. Although the assignments can be viewed as unit-clauses, they are treated specially in many ways, and are best viewed as a separate type of information.

The only inference mechanism used by a standard solver is *unit propagation*. As soon as a clause becomes *unit* under the current assignment (all literals except

[2] This includes SAT-solvers such as: ZCHAFF, LIMMAT, BERKMIN.

one are false), the remaining unbound literal is set to true, possibly making more clauses unit. The process is continued until no more information can be propagated.

The search procedure of a modern solver is the most complex part. Heuristically, variables are picked and assigned values (*assumptions* are made), until the propagation detects a *conflict* (all literals of a clause have become false). At that point, a so called *conflict clause* is constructed and added to the SAT problem. Assumptions are then canceled by backtracking until the conflict clause becomes unit, from which point this unit clause is propagated and the search process continues.

MINISAT is extensible with arbitrary boolean constraints. This will affect the *representation*, which must be able to store these constraints; the *inference*, which must be able to derive unit information from these constraints; and the *search*, which must be able to analyze and generate conflict clauses from the constraints. The mechanism we suggest for managing general constraints is very lightweight, and by making the dependencies between the SAT-algorithm and the constraints implementation explicit, we feel it rather adds to the clarity of the solver than obscures it.

Propagation. The propagation procedure of MINISAT is largely inspired by that of CHAFF [MZ01]. For each literal, a list of constraints is kept. These are the constraints that *may* propagate unit information (variable assignments) if the literal becomes TRUE. For clauses, no unit information can be propagated until all literals except one have become FALSE. Two unbound literals p and q of the clause are therefore selected, and references to the clause are added to the lists of $\overline{p}$ and $\overline{q}$ respectively. The literals are said to be *watched* and the lists of constraints are referred to as *watcher lists*. As soon as a watched literal becomes TRUE, the constraint is invoked to see if information may be propagated, or to select new unbound literals to be watched.

A feature of the watcher system for clauses is that on backtracking, no adjustment to the watcher lists need to be done. Backtracking is therefore very cheap. However, for other constraint types, this is not necessarily a good approach. MINISAT therefore supports the optional use of *undo lists* for those constraints; storing what constraints need to be updated when a variable becomes unbound by backtracking.

Learning. The learning procedure of MINISAT follows the ideas of Marques-Silva and Sakallah in [MS96]. The process starts when a constraint becomes conflicting (impossible to satisfy) under the current assignment. The conflicting constraint is then asked for a set of variable assignments that make it contradictory. For a clause, this would be all the literals of the clause (which are FALSE under a conflict). Each of the variable assignments returned must be either an *assumption* of the search procedure, or the result of some *propagation* of a constraint. The propagating constraints are in turn asked for the set of variable assignments that forced the propagation to occur, continuing the analysis backwards. The procedure is repeated until some termination condition is fulfilled, resulting in a set of variable assignments that implies the conflict. A clause prohibiting that particular assignment is added to the clause database. This *learnt*

clause must always, by construction, be implied by the original problem constraints.

Learnt clauses serve two purposes: they drive the backtracking (as we shall see) and they speed up future conflicts by "caching" the reason for the conflict. Each clause will prevent only a constant number of inferences, but as the recorded clauses start to build on each other and participate in the unit propagation, the accumulated effect of learning can be massive. However, as the set of learnt clauses increase, propagation is slowed down. Therefore, the number of learnt clauses is periodically reduced, keeping only the clauses that seem useful by some heuristic.

Search. The search procedure of a conflict-driven SAT-solver is somewhat implicit. Although a recursive definition of the procedure might be more elegant, it is typically described (and implemented) iteratively. The procedure will start by selecting an unassigned variable x (called the *decision variable*) and assume a value for it, say TRUE. The consequences of $x = \text{TRUE}$ will then be propagated, possibly resulting in more variable assignments. All variables assigned as a consequence of x is said to be from the same *decision level*, counting from 1 for the first assumption made and so forth. Assignments made before the first assumption (decision level 0) are called *top-level*.

All assignments will be stored on a stack in the order they were made; from now on referred to as the *trail*. The trail is divided into decision levels and is used to undo information during backtracking.

The decision phase will continue until either all variables have been assigned, in which case we have a model, or a conflict has occurred. On conflicts, the learning procedure will be invoked and a conflict clause produced. The trail will be used to undo decisions, one level at a time, until precisely one of the literals of the learnt clause becomes unbound (they are all FALSE at the point of conflict). By construction, the conflict clause cannot go directly from conflicting to a clause with two or more unbound literals. If the clause remains unit for several decision levels, it is advantageous to chose the lowest level (referred to as *backjumping* or *non-chronological backtracking* [MS96]).

```
loop
    propagate()    – propagate unit clauses
    if not conflict then
        if all variables assigned then
            return SATISFIABLE
        else
            decide()    – pick a new variable and assign it
    else
        analyze()    – analyze conflict and add a conflict clause
        if top-level conflict found then
            return UNSATISFIABLE
        else
            backtrack()    – undo assignments until conflict clause is unit
```

An important part of the procedure is the heuristic for *decide()*. Like CHAFF, MINISAT uses a dynamic variable order that gives priority to variables involved in recent conflicts.

[Disc] Although this is a good default order, domain specific heuristics have successfully been used in various areas to improve the performance [Stri00]. Variable ordering is a traditional target for improving SAT-solvers.

Activity heuristics. One important technique introduced by CHAFF [MZ01] is a dynamic variable ordering based on activity (referred to as the VSIDS heuristic). The original heuristic imposes an order on *literals*, but borrowing from SATZOO, we make no distinction between p and $\overline{p}$ in MINISAT.

Each variable has an *activity* attached to it. Every time a variable occurs in a recorded conflict clause, its activity is increased. We will refer to this as *bumping*. After recording the conflict, the activity of all the variables in the system are multiplied by a constant less than 1, thus *decaying* the activity of variables over time. Recent increments count more than old. The current sum determines the activity of a variable.

In MINISAT we use a similar idea for clauses. When a learnt clause is used in the analysis process of a conflict, its activity is bumped. Inactive clauses are periodically removed.

Constraint removal. The constraint database is divided into two parts: the *problem constraints* and the *learnt clauses*. As we have noted, the set of learnt clauses can be periodically reduced to increase the performance of propagation. Learnt clauses are used to crop future branches in the search tree, so we risk getting a bigger search space instead. The balance between the two forces is delicate, and there are SAT-instances for which a big learnt clause set is advantageous, and others where a small set is better. MINISAT's default heuristic starts with a small set and gradually increases the size.

Problem constraints can also be removed if they are satisfied at the top-level. The API method *simplifyDB()* is responsible for this. The procedure is particularly important for incremental SAT-problems, where techniques for clause removal build on this feature.

Top-level solver. Although the pseudo-code for the search procedure presented above suffices for a simple conflict-driven SAT-solver, a solver *strategy* can improve the performance. A typical strategy applied by modern conflict-driven SAT-solvers is the use of *restarts* to escape from futile parts of the search tree. In MINISAT we also vary the number of learnt clauses kept at a given time. Furthermore, the *solve()* method of the API supports incremental assumptions, not handled by the above pseudo-code.

4 Implementation

The following conventions are used in the code. Atomic types start with a lowercase letter and are passed by value. Composite types start with a capital letter and are passed by reference. Blocks are marked only by indentation level. The

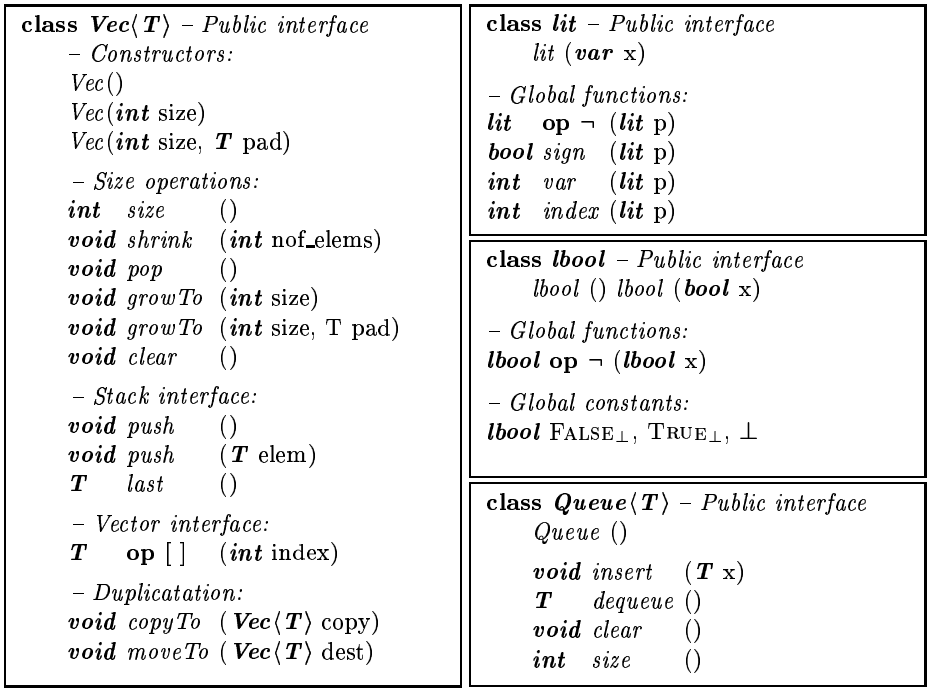


Fig. 1. *Basic abstract data types used throughout the code.* The vector data type can push a default constructed element by the *push*() method with no argument. The *moveTo*() method will move the contents of a vector to another vector in constant time, clearing the source vector. The literal data type has an *index*() method which converts the literal to a "small" integer suitable for array indexing. The *var*() method returns the underlying variable of the literal, and the *sign*() method if the literal is signed (FALSE for x and TRUE for $\overline{x}$).

bottom symbol $\bot$ will always mean *undefined*; the symbol FALSE will be used to denote the boolean false.

We will use, but not specify an implementation of, the following abstract data types: ***Vec***$\langle \boldsymbol{T} \rangle$ an extensible vector of type $\boldsymbol{T}$; ***lit*** the type of literals containing a special literal $\bot_{lit}$; ***lbool*** for the lifted boolean domain containing elements $\text{TRUE}_\bot$, $\text{FALSE}_\bot$, and $\bot$; ***Queue***$\langle \boldsymbol{T} \rangle$ a queue of type $\boldsymbol{T}$. We also use ***var*** as a type synonym for ***int*** (for implicit documentation) with the special constant $\bot_{var}$. The interfaces of the abstract data types are presented in *Figure 1.*

4.1 The solver state

A number of things need to be stored in the solver state. *Figure 2* shows the complete set of member variables of the solver type of MINISAT. Together with the state variables we define some short helper methods in *Figure 3*, as well as the interface of *VarOrder* (*Figure 4*), explained in section 4.6.

The state does *not* contain a boolean "conflict" to remember if a top-level conflict has been reached. Instead we impose as an invariant that the solver must never be in a conflicting state. As a consequence, any method that puts the solver [Disc]

in a conflicting state must communicate this. Using the solver object after this point is illegal. The invariant makes the interface slightly more cumbersome to use, but simplifies the implementation, which is important when extending and experimenting with new techniques.

4.2 Constraints

MINISAT can handle arbitrary constraints over boolean variables through the abstraction presented in *Figure 5*. Each constraint type needs to implement methods for constructing, removing, propagating and calculating reasons. In addition, methods for simplifying the constraint and updating the constraint on backtrack can be specified. We explain the meaning and responsibilities of these methods in detail:

Constructor. The constructor may only be called at the top-level. It must create and add the constraint to appropriate watcher lists after enqueuing any unit information derivable under the current top-level assignment. Should a conflict arise, this must be communicated to the caller.

Remove. The remove method supplants the destructor by receiving the solver state as a parameter. It should dispose the constraint and remove it from the watcher lists.

Propagate. The propagate method is called if the constraint is found in a watcher list during propagation of unit information p. The constraint is removed from the list and is required to insert itself into a new or the same watcher list. Any unit information derivable as a consequence of p should be enqueued. If successful, TRUE is returned; if a conflict is detected, FALSE is returned. The constraint may add itself to the undo list of $var(p)$ if it needs to be updated when p becomes unbound.

Simplify. At the top-level, a constraint may be given the opportunity to simplify its representation (returns TRUE) or state that the constraint is satisfied under the current assignment (returns FALSE). A constraint must *not* be simplifiable to produce unit information or to be conflicting; in that case the propagation has not been correctly defined.

Undo. During backtracking, this method is called if the constraint added itself to the undo list of $var(p)$ in *propagate()*. The current variable assignments are guaranteed to be identical to that of the moment before *propagate()* was called.

Calculate Reason. This method is given a literal p and an empty vector. The constraint is the *reason* for p being true, that is, during propagation, the current constraint enqueued p. The received vector is extended to include a set of assignments (represented as literals) implying p. The current variable assignments are guaranteed to be identical to that of the moment before the constraint propagated p. The literal p is also allowed to be the special constant $\bot_{lit}$ in which case the reason for the clause being *conflicting* should be returned through the vector.

```
class Solver
  – Constraint database
  Vec⟨Constr⟩ constrs     – List of problem constraints.
  Vec⟨Clause⟩ learnts     – List of learnt clauses.
  double      cla_inc     – Clause activity increment – amount to bump with.
  double      cla_decay   – Decay factor for clause activity.

  – Variable order
  Vec⟨double⟩ activity    – Heuristic measurement of the activity of a variable.
  double      var_inc     – Variable activity increment – amount to bump with.
  double      var_decay   – Decay factor for variable activity.
  VarOrder    order       – Keeps track of the dynamic variable order.

  – Propagation
  Vec⟨Vec⟨Constr⟩⟩ watches – For each literal 'p', a list of constraints watching 'p'.
                             A constraint will be inspected when 'p' becomes true.
  Vec⟨Vec⟨Constr⟩⟩ undos   – For each variable 'x', a list of constraints that need to
                             update when 'x' becomes unbound by backtracking.
  Queue⟨lit⟩  propQ       – Propagation queue.

  – Assignments
  Vec⟨lbool⟩  assigns     – The current assignments indexed on variables.
  Vec⟨lit⟩    trail       – List of assignments in chronological order.
  Vec⟨int⟩    trail_lim   – Separator indices for different decision levels in 'trail'.
  Vec⟨Constr⟩ reason      – For each variable, the constraint that implied its value.
  Vec⟨int⟩    level       – For each variable, the decision level it was assigned.
  int         root_level  – Separates incremental and search assumptions.
```

Fig. 2. Internal state of the solver.

```
int   Solver.nVars()          return assigns.size()
int   Solver.nAssigns()       return trail.size()
int   Solver.nConstraints()   return constrs.size()
int   Solver.nLearnts()       return learnts.size()
lbool Solver.value(var x)     return assigns[x]
lbool Solver.value(lit p)     return sign(p) ? ¬assigns[var(p)] : assigns[var(p)]
int   Solver.decisionLevel()  return trail_lim.size()
```

Fig. 3. Small helper methods. For instance, *nLearnts()* returns the number of learnt clauses.

```
class VarOrder – Public interface
  VarOrder ( Vec⟨lbool⟩ ref_to_assigns, Vec⟨double⟩ ref_to_activity)

  void newVar()      – Called when a new variable is created.
  void update(var x) – Called when variable has increased in activity.
  void updateAll()   – Called when all variables have been assigned new activities.
  void undo(var x)   – Called when variable is unbound (may be selected again).
  var  select()      – Called to select a new, unassigned variable.
```

Fig. 4. Assisting ADT for the dynamic variable ordering of the solver. The constructor takes references to the assignment vector and the activity vector of the solver. The method *select()* will return the unassigned variable with the highest activity.

```
class Constr
  virtual void remove     (Solver S)                                  - must be defined
  virtual bool propagate  (Solver S, lit p)                           - must be defined
  virtual bool simplify   (Solver S)                                  - defaults to return false
  virtual void undo       (Solver S, lit p)                           - defaults to do nothing
  virtual void calcReason (Solver S, lit p, Vec<lit> out_reason)      - must be defined
```

Fig. 5. Abstract base class for constraints.

The code for the *Clause* constraint is presented in *Figure 7*. It is also used for learnt clauses, which are unique in that they can be added to the clause database while the solver is not at top-level. This makes the constructor code a bit more complicated than it would be for a normal constraint.

Implementing the *addClause()* method of the solver API is just a matter of calling *Clause_new()* and pushing the new constraint on the "constrs" vector, storing the list of problem constraints. For completeness, we also display the code for creating variables in the solver (*Figure 6*).

```
var Solver.newVar()
    int index
    index = nVars()
    watches .push()
    watches .push()
    undos   .push()
    reason  .push(NULL)
    assigns .push(⊥)
    level   .push(-1)
    activity .push(0)
    order   .newVar()
    return index
```

Fig. 6. Creates a new SAT variable in the solver.

[Disc] There are a number of tricks for smart-coding that can be used in a C++ implementation of *Clause*. In particularly the "lits" vector can be implemented as an zero-sized array placed last in the class, and then extra memory allocated for the clause to contain the data. We observed a 20% speedup for this trick. Furthermore, memory can be saved by not storing activity for problem clauses.

[Disc] Of the methods defining a constraint, *propagate()* should be the primary target for efficient implementation. The SAT-solver spends about 80% of the time propagating, so the method will be called frequently. In SATZOO a performance gain was achieved by remembering the position of the last watched literal and start looking for a new literal to watch from that position. Further speedups may be achieved by specializing the code for small clause sizes.

4.3 Propagation

Given the mechanism for adding constraints, we now move on to describe the propagation of unit information on these constraints.

The propagation routine keeps a set of literals (unit information) that is to be propagated. We call this the *propagation queue*. When a literal is inserted into the queue, the corresponding variable is immediately assigned. For each literal in the queue, the watcher list of that literal determines the constraints that may be affected by the assignment. Through the interface described in the previous section, each constraint is asked by a call to its *propagate()* method if more unit information can be inferred, which will then be enqueued. The process continues until either the queue is empty or a conflict is found.

```
class Clause : public Constr
  bool      learnt
  float     activity
  Vec⟨lit⟩ lits

  – Constructor – creates a new clause and adds it to watcher lists:
  static bool Clause_new(Solver S, Vec⟨lit⟩ ps, bool learnt, Clause out_clause)
      "Implementation in Figure 8"

  – Learnt clauses only:
  bool locked(Solver S)
      return S.reason[var(lits[0])] == this

  – Constraint interface:
  void remove(Solver S)
      removeElem(this, S.watches[index(¬lits[0])])
      removeElem(this, S.watches[index(¬lits[1])])
      delete this

  bool simplify(Solver S)          – only called at top-level with empty prop. queue
      int j = 0
      for (int i = 0; i < lits.size(); i++)
          if (S.value(lits[i]) == TRUE⊥)
              return TRUE
          else if (S.value(lits[i]) == ⊥)
              lits[j++] = lits[i] – false literals are not copied (only occur for i ≥ 2)
      lits.shrink(lits.size() − j)
      return FALSE

  bool propagate(Solver S, lit p)
      – Make sure the false literal is lits[1]:
      if (lits[0] == ¬p)
          lits[0] = lits[1], lits[1] = ¬p

      – If 0th watch is true, then clause is already satisfied.
      if (S.value(lits[0]) == TRUE⊥)
          S.watches[index(p)].push(this)          – re-insert clause into watcher list
          return TRUE

      – Look for a new literal to watch:
      for (int i = 2; i < size(); i++)
          if (S.value(lits[i]) != FALSE⊥)
              lits[1] = lits[i], lits[i] = ¬p
              S.watches[index(¬lits[1])].push(this) – insert clause into watcher list
              return TRUE

      – Clause is unit under assignment:
      S.watches[index(p)].push(this)
      return S.enqueue(lits[0], this)                    – enqueue for propagation

  void calcReason(Solver S, lit p, vec⟨lit⟩ out_reason)
      – invariant: (p == ⊥) or (p == lits[0])
      for (int i = ((p == ⊥) ? 0 : 1); i < size(); i++)
          out_reason.push(¬lits[i])          – invariant: S.value(lits[i]) == FALSE⊥
      if (learnt) S.claBumpActivity(this)
```

Fig. 7. Implementation of the *Clause* constraint.

```
bool Clause_new(Solver S, Vec⟨lit⟩ ps, bool learnt, Clause out_clause)
  out_clause = NULL

  – Normalize clause:
  if (!learnt)
      if ("any literal in ps is true")      return TRUE
      if ("both p and ¬p occurs in ps") return TRUE
      "remove all false literals from ps"
      "remove all duplicates from ps"

  if (ps.size() == 0)
      return FALSE
  else if (ps.size() == 1)
      return S.enqueue(ps[0])                          – unit facts are enqueued
  else
      – Allocate clause:
      Clause c = new Clause
      ps.moveTo(c.lits)
      c.learnt   = learnt
      c.activity = 0                                   – only relevant for learnt clauses

      if (learnt)
          – Pick a second literal to watch:
          "Let max_i be the index of the literal with highest decision level"
          c.lits[1] = ps[max_i], c.lits[max_i] = ps[1]

          – Bumping:
          S.claBumpActivity(c)  – newly learnt clauses should be considered active
          for (int i = 0; i < ps.size(); i++)
              S.varBumpActivity(ps[i])  – variables in conflict clauses are bumped

      – Add clause to watcher lists:
      S.watches[index(¬c.lits[0])].push(c)
      S.watches[index(¬c.lits[1])].push(c)
      out_clause = c

      return TRUE
```

Fig. 8. Constructor function for clauses. Returns FALSE if top-level conflict is detected. 'out_clause' may be set to NULL if the new clause is already satisfied under the current top-level assignment. **Post-condition:** 'ps' is cleared. For learnt clauses, all literals will be false except 'lits[0]' (this by design of the *analyze*() method). For the propagation to work, the second watch must be put on the literal which will first be unbound by backtracking. (Note that none of the learnt-clause specific things needs to be done for a user defined constraint type.)

An implementation of this procedure is displayed in *Figure 9*. It starts by dequeuing a literal and clearing the watcher list for that literal by moving it to "tmp". The propagate method is then called for each constraint of "tmp". This will re-insert watches into new lists. Should a conflict be detected during the traversal of "tmp", the remaining watches will be copied back to the original watcher list, and the propagation queue cleared.

The method for enqueuing unit information is relatively straightforward. Note that the same fact can be enqueued several times, as it may be propagated from different constraints, but it will only be put on the propagation queue once.

It may be that later enqueuings have a "better" reason (determined heuristically) and a small performance gain was achieved in SATZOO by changing reason if the new reason was smaller than the previously stored. The changing affects the conflict clause generation described in the next section. [Disc]

4.4 Learning

This section describes the conflict-driven clause learning. It was first described in [MS96] and is one of the major advances of SAT-technology in the last decade.

We describe the basic conflict-analysis algorithm by an example. Assume the database contains the clause $\{x, y, z\}$ which just became unsatisfied during propagation. This is our conflict. We call $\overline{x} \wedge \overline{y} \wedge \overline{z}$ the reason set of the conflict. Now x is false because $\overline{x}$ was propagated from some constraint. We ask that constraint to give us the reason for propagating $\overline{x}$ (the *calcReason()* method). It will respond with another conjunction of literals, say $u \wedge v$. These were the variable assignment that implied $\overline{x}$. The constraint may in fact have been the clause $\{\overline{u}, \overline{v}, \overline{x}\}$. From this little analysis we know that $u \wedge v \wedge \overline{y} \wedge \overline{z}$ must also lead to a conflict. We may prohibit this conflict by adding the clause $\{\overline{u}, \overline{v}, y, z\}$ to the clause database. This would be an example of a *learnt* conflict clause.

In the example, we picked only one literal and analyzed it one step. The process of expanding literals with their reason sets can be continued, in the extreme case until all the literals of the conflict set are decision variables (which were not propagated by any constraints). Different learning schemes based on this process have been proposed. Experimentally the "First Unique Implication Point" (First UIP) heuristic has been shown effective [ZM01]. We will not give the definition of UIPs here, but just state the algorithm: In a breadth-first manner, continue to expand literals of the current decision level, until there is just one left.

In the code for *analyze()*, displayed in *Figure 10*, we make use of the fact that a breadth-first traversal can be achieved by inspecting the trail backwards. Especially, the variables of the reason set of p is always before p in the trail. Furthermore, in the algorithm we initialize p to $\bot_{lit}$, which will make *calcReason()* return the reason for the conflict.

Assuming x to be the unit information that causes the conflict, an alternative implementation would be to calculate the reason for $\overline{x}$ and just add x to that set. The code would be slightly more cumbersome but the contract for *calcReason()* would be simpler, as we no longer need the special case for $\bot_{lit}$. [Disc]

```
Constr Solver.propagate()
  while (propQ.size() > 0)
      lit p = propQ.dequeue()          – 'p' is now the enqueued fact to propagate
      Vec⟨Constr⟩ tmp                  – 'tmp' will contain the watcher list for 'p'
      watches[index(p)].moveTo(tmp)

      for (int i = 0; i < tmp.size(); i++)
          if (!tmp[i].propagate(this, p))
              – Constraint is conflicting; copy remaining watches to 'watches[p]'
              – and return constraint:
              for (int j = i+1; j < tmp.size(); j++)
                watches[index(p)].push(tmp[j])
              propQ.clear()
              return tmp[i]
  return NULL
```

```
bool Solver.enqueue(lit p, Constr from = NULL)
  if (value(p) != ⊥)
      if (value(p) == FALSE⊥)
          – Conflicting enqueued assignment
          return FALSE
      else
          – Existing consistent assignment – don't enqueue
          return TRUE
  else
      – New fact, store it
      assigns [var(p)] = lbool(!sign(p))
      level   [var(p)] = decisionLevel()
      reason  [var(p)] = from
      trail.push(p)
      propQ.insert(p)
      return TRUE
```

Fig. 9. ***propagate():*** Propagates all enqueued facts. If a conflict arises, the *conflicting* clause is returned, otherwise NULL. ***enqueue():*** Puts a new fact on the propagation queue, as well as immediately updating the variable's value in the assignment vector. If a conflict arises, FALSE is returned and the propagation queue is cleared. The parameter 'from' contains a reference to the constraint from which 'p' was propagated (defaults to NULL if omitted).

Finally, the analysis not only returns a conflict clause, but also the backtracking level. This is the lowest decision level for which the conflict clause is unit. It is advantageous to backtrack as far as possible [MS96], and is referred to as *back-jumping* or *non-chronological backtracking* in the literature.

4.5 Search

The search method in *Figure 13* works basically as described in section 3 but with the following additions:

Restarts. The first argument of the search method is "nof_conflicts". The search for a model or a contradiction will only be conducted for this many

```
void Solver.analyze(Constr confl, Vec⟨lit⟩ out_learnt, Int out_btlevel)
  Vec⟨bool⟩ seen(nVars(), FALSE)
  int       counter  = 0
  lit       p        = ⊥_lit
  Vec⟨lit⟩  p_reason

  out_learnt.push()                              – leave room for the asserting literal
  out_btlevel = 0
  do
      p_reason.clear()
      confl.calcReason(this, p, p_reason)        – invariant here: confl != NULL

      – TRACE REASON FOR P:
      for (int j = 0; j < p_reason.size(); j++)
          lit q = p_reason[j]
          if (!seen[var(q)])
              seen[var(q)] = TRUE
              if (level[var(q)] == decisionLevel())
                  counter++
              else if (level[var(q)] > 0)  – exclude variables from decision level 0
                  out_learnt.push(¬q)
                  out_btlevel = max(out_btlevel, level[var(q)])

      – SELECT NEXT LITERAL TO LOOK AT:
      do
          p     = trail.last()
          confl = reason[var(p)]
          undoOne()
      while (!seen[var(p)])
      counter−−
  while (counter > 0)
  out_learnt[0] = ¬p
```

Fig. 10. Analyze a conflict and produce a reason clause. **Pre-conditions:** (1) 'out_learnt' is assumed to be cleared. (2) Current decision level must be greater than root level. **Post-conditions:** (1) 'out_learnt[0]' is the asserting literal at level 'out_btlevel'. **Effect:** Will undo part of the trail, but not beyond last decision level.

```
void Solver.record( Vec⟨lit⟩ clause)
  Clause c              – will be set to created clause, or NULL if 'clause[]' is unit
  Clause_new(this, clause, TRUE, c)                    – cannot fail at this point
  enqueue(clause[0], c)                                – cannot fail at this point
  if (c != NULL) learnts.push(c)
```

Fig. 11. Record a clause and drive backtracking. Pre-condition: 'clause[0]' must contain the asserting literal. In particular, 'clause[]' must not be empty.

conflicts. If failing to solve the SAT-problem within the bound, all assumptions will be canceled and $\perp$ returned. The surrounding solver strategy will then restart the search, possibly with a new set of parameters.

Reduce. The second argument, "nof_learnts", sets an upper limit on the number of learnt clauses that are kept. Once this number is reached, *reduceDB()* is called. Clauses that are currently the reason for a variable assignment are said to be *locked* and cannot be removed by *reduceDB()*. For this reason, the limit is extended by the number of assigned variables, which approximates the number of locked clauses.

Parameters. The third argument to the search method groups some tuning constants. In the current version of MINISAT, it only contains the decay factors for variables and clauses.

Root-level. To support incremental SAT, the concept of a *root-level* is introduced. The root-level acts a bit as a new top-level. Above the root-level are the incremental assumptions passed to *solve()* (if any). The search procedure is not allowed to backtrack above the root-level, as this would change the incremental assumptions. If we reach a conflict at root-level, the search will return FALSE.

A problem with the approach presented here is conflict clauses that are unit. For these, *analyze()* will always return a backtrack level of 0 (top-level). As unit clauses are treated specially, they are never added to the clause database. Instead they are enqueued as facts to be propagated (see the code of *Clause_new()*). There would be no problem if this was done at top-level. However, the search procedure will only undo until root-level, which means that the unit fact will be enqueued there. Once *search()* has solved the current SAT-problem, the surrounding solver strategy will undo any incremental assumption and put the solver back at the top-level. By this the unit clause will be forgotten, and the next incremental SAT problem will have to infer it again.

A solution to this is to store the learnt unit clauses in a vector and re-insert them at top-level before the next call to *solve()*. The reason for omitting this in MINISAT is that we have not seen any performance gain by this extra handling in our applications [ES03,CS03]. Simplicity thus dictates that we leave it out of the presentation.

Simplify. Provided the root-level is 0 (no assumptions were passed to *solve()*) the search will return to the top-level every time a unit clause is learnt. At that point it is legal to call *simplifyDB()* to simplify the problem constraints according to the top-level assignment. If a stronger simplifier than presented here is implemented, a contradiction may be found, in which case the search should be aborted. As our simplifier is not stronger than normal propagation, it can never reach a contradiction, so we ignore the return value of *simplify()*.

```
void Solver.undoOne()
    lit   p = trail.last()
    var   x = var(p)
    assigns [x] = ⊥
    reason  [x] = NULL
    level   [x] = -1
    order.undo(x)
    trail.pop()
    while (undos[x].size() > 0)
        undos[x].last().undo(this, p)
        undos[x].pop()
```

```
bool Solver.assume(lit p)
    trail_lim.push(trail.size())
    return enqueue(p)
```

```
void Solver.cancel()
    int c = trail.size() − trail_lim.last()
    for (; c != 0; c−−)
        undoOne()
    trail_lim.pop()
```

```
void Solver.cancelUntil(int level)
    while (decisionLevel() > level)
        cancel()
```

Fig. 12. ***assume():*** returns FALSE if immediate conflict. **Pre-condition:** propagation queue is empty. ***undoOne():*** unbinds the last variable on the trail. ***cancel():*** reverts to the state before last *push*(). **Pre-condition:** propagation queue is empty. ***cancelUntil():*** cancels several levels of assumptions.

4.6 Activity heuristics

The implementation of activity is shown in *Figure 14*. Instead of actually multiplying all variables by a decay factor after each conflict, we bump variables with larger and larger numbers. Only relative values matter. Eventually we will reach the limit of what is representable by a floating point number. At that point, all activities are scaled down.

In the *VarOrder* data type of MINISAT, the list of variables is kept sorted on activity at all time. The backtracking will always accurately choose the most active variable. The original suggestion for the VSIDS dynamic variable ordering was to sort periodically.

The polarity of a literal is ignored in MINISAT. However, storing the latest polarity of a variable might improve the search when restarts are used, but it remains to be empirically supported. Furthermore, the interface of *VarOrder* can be used for other variable heuristics. In SATZOO, an initial static variable order computed from the clause structure was particularly successful on many problems. [Disc]

4.7 Constraint removal

The methods for reducing the set of learnt clauses as well as the top-level simplification procedure can be found in *Figure 15*.

When removing learnt clauses, it is important not to remove so called *locked* clauses. Locked clauses are those participating in the current backtracking branch by being the reason (through propagation) for a variable assignment. The reduce procedure keeps half of the learnt clauses, except for those which have decayed below a threshold limit. Such clauses can occur if the set of active constraints is very small.

Top-level simplification can be seen as a special case of propagation. Since [Disc]

```
lbool Solver.search(int nof_conflicts, int nof_learnts, SearchParams params)
  int conflictC = 0
  var_decay = 1 / params.var_decay
  cla_decay = 1 / params.cla_decay
  model.clear()

  loop
      Constr confl = propagate()
      if (confl != NULL)
          - CONFLICT

          conflictC++
          Vec<lit> learnt_clause
          int backtrack_level
          if (decisionLevel() == root_level)
              return FALSE⊥
          analyze(confl, learnt_clause, backtrack_level)
          cancelUntil(max(backtrack_level, root_level))
          record(learnt_clause)
          decayActivities()
      else
          - NO CONFLICT

          if (decisionLevel() == 0)
              - Simplify the set of problem clauses:
              simplifyDB()                    - our simplifier cannot return false here

          if (learnts.size()-nAssigns() ≥ nof_learnts)
              - Reduce the set of learnt clauses:
              reduceDB()

          if (nAssigns() == nVars())
              - Model found:
              model.growTo(nVars())
              for (int i = 0; i < nVars(); i++)
                 model[i] = (value(i) == TRUE⊥)
              cancelUntil(root_level)
              return TRUE⊥
          else if (conflictC ≥ nof_conflicts)
              - Reached bound on number of conflicts:
              cancelUntil(root_level)                              - force a restart
              return ⊥
          else
              - New variable decision:
              lit p = lit(order.select())       - may have heuristic for polarity here
              assume(p)                                            - cannot return false
```

Fig. 13. Search method. Assumes and propagates until a conflict is found, from which a conflict clause is learnt and backtracking performed until search can continue. **Precondition:** root_level == *decisionLevel*().

```
void Solver.varBumpActivity(var x)
  if ((activity[x] += var_inc) > 1e100)
      varRescaleActivity()
  order.update(x)

void Solver.varDecayActivity()
  var_inc *= var_decay

void Solver.varRescaleActivity()
  for (int i = 0; i < nVars(); i++)
      activity[i] *= 1e-100
  var_inc *= 1e-100
```

```
void Solver.claBumpActivity(Clause c)
void Solver.claDecayActivity()
void Solver.claRescaleActivity()
    - Similarly implemented.

void Solver.decayActivities()
    varDecayActivity()
    claDecayActivity()
```

Fig. 14. Bumping of variable and clause activities.

it is performed under no assumption, anything learnt can be kept forever. The freedom of not having to store derived information separately, with the ability to undo it later, makes it easier to implement stronger propagation.

4.8 Top-level solver

The method implementing MINISAT's top-level strategy can be found in *Figure 16*. It is responsible for making the incremental assumptions and setting the root level. Furthermore, it completes the simple backtracking search with restarts, which are performed less and less frequently. After each restart, the number of allowed learnt clauses is increased.

The code contains a number of hand-tuned constants that have shown to perform reasonable on our applications [ES03,CS03]. The top-level strategy, however, is a productive target for improvements (possibly application dependent). In SATZOO, the top-level strategy contains an initial phase where a static variable ordering is used.

5 Conclusions and Related Work

By this paper, we have provided a minimal reference implementation of a modern conflict-driven SAT-solver. Despite the abstraction layer for boolean constraints, and the lack of more sophisticated heuristics, the performance of MINISAT is comparable to state-of-the-art SAT-solvers. We have tested MINISAT against ZCHAFF and BERKMIN 5.61 on 177 SAT-instances. These instances were used to tune SATZOO for the *SAT 2003 Competition*. As SATZOO solved more instances and series of problems, ranging over all three categories (*industrial, handmade,* and *random*), than any other solver in the competition, we feel that this is a good test-set for the overall performance. No extra tuning was done in MINISAT; it was just run once with the constants presented in the code. At a time-out of 10 minutes, MINISAT solved 158 instances, while ZCHAFF solved 147 instances and BERKMIN 157 instances.

Another approach to incremental SAT and non-clausal constraints was presented by Aloul, Ramani, Markov, and Sakallah in their work on SATIRE and PBS [WKS01,ARMS02]. Our implementation differs in that it has a simpler

```
void Solver.reduceDB()
  int     i, j
  double lim = cla_inc / learnts.size()

  sortOnActivity(learnts)
  for (i=j=0; i < learnts.size()/2; i++)
      if (!learnts[i].locked(this))
          learnts[i].remove(this)
      else
          learnts[j++] = learnts[i]
  for (; i < learnts.size(); i++)
      if (!learnts[i].locked(this)
      && learnts[i].activity() < lim)
          learnts[i].remove(this)
      else
          learnts[j++] = learnts[i]
  learnts.shrink(i − j)
```

```
bool Solver.simplifyDB()
  if (propagate() != NULL)
      return FALSE

  for (int type = 0; type < 2; type++)
      Vec⟨Constr⟩ cs = type ?
        (Vec⟨Constr⟩)learnts : constrs
      int j = 0
      for (int i = 0; i < cs.size(); i++)
          if (cs[i].simplify(this))
              cs[i].remove(this)
          else
              cs[j++] = cs[i]
      cs.shrink(cs.size()−j)
  return TRUE
```

Fig. 15. ***reduceDB():*** Remove half of the learnt clauses minus some locked clauses. A locked clause is a clauses that is reason to a current assignment. Clauses below a certain lower bound activity are also be removed. ***simplifyDB():*** Top-level simplify of constraint database. Will remove any satisfied constraint and simplify remaining constraints under current (partial) assignment. If a top-level conflict is found, FALSE is returned. **Pre-condition:** Decision level must be zero. **Post-condition:** Propagation queue is empty.

```
bool Solver.solve(Vec⟨lit⟩ assumps)
    SearchParams params(0.95, 0.999)
    double nof_conflicts = 100
    double nof_learnts   = nConstraints()/3
    lbool  status        = ⊥

    – PUSH INCREMENTAL ASSUMPTIONS:
    for (int i = 0; i < assumps.size(); i++)
        if (!assume(assumps[i]) || propagate() != NULL)
            cancelUntil(0)
            return FALSE
    root_level = decisionLevel()

    – SOLVE:
    while (status == ⊥)
        status = search((int)nof_conflicts, (int)nof_learnts, params)
        nof_conflicts *= 1.5
        nof_learnts   *= 1.1

    cancelUntil(0)
    return status == TRUE⊥
```

Fig. 16. Main solve method. **Pre-condition:** If assumptions are used, *simplifyDB()* must be called right before using this method. If not, a top-level conflict (resulting in a non-usable internal state) cannot be distinguished from a conflict under assumptions.

notion of incrementality, and that it contains a well documented interface for non-clausal constraints.

Finally, a set of reference implementations of modern SAT-techniques is present in the OpenSAT project.[3] However, the project aim for completeness rather than minimal exposition, as we have chosen in this paper.

6 Exercises

1. Write the code for an *AtMost* constraint. The constraint is satisfied if at most n out of m specified literals are true.
2. Implement a generator for (generalized) pigeon-hole formulas using the new constraints. The generator should take three arguments: number of pigeons, number of holes, and hole capacity. Each pigeon must reside in some pigeon-hole. No hole may contain more pigeons than its capacity.
3. Make an incremental version that adds one pigeon to the problem at a time until the problem becomes unsatisfiable.

References

[ARMS02] F. Aloul, A. Ramani, I. Markov, K. Sakallah. **"Generic ILP vs. Specialized 0-1 ILP: an Update"** in *International Conference on Computer Aided Design (ICCAD)*, 2002.

[BCCFZ99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu. **"Symbolic Model Checking using SAT procedures instead of BDDs"** in *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[CS03] K. Claessen, N. Sörensson. **"New Techniques that Improve MACE-style Finite Model Finding"** in *CADE-19, Workshop W4. Model Computation – Principles, Algorithms, Applications*, 2003.

[DLL62] M. Davis, G. Logemann, D. Loveland. **"A machine program for theorem proving"** in *Communications of the ACM*, vol 5, 1962.

[ES03] N. Eén, N. Sörensson. **"Temporal Induction by Incremental SAT Solving"** in *Proc. of First International Workshop on Bounded Model Checking*, 2003. ENTCS issue 4 volume 89.

[Lar92] T. Larrabee. **"Test Pattern Generation Using Boolean Satisfiability"** in *IEEE Transactions on Computer-Aided Design*, vol. 11-1, 1992.

[MS96] J.P. Marques-Silva, K.A. Sakallah. **"GRASP – A New Search Algorithm for Satisfiability"** in *ICCAD. IEEE Computer Society Press*, 1996

[MZ01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. **"Chaff: Engineering an Efficient SAT Solver"** in *Proc. of the 38th Design Automation Conference*, 2001.

[Stri00] O. Strichman **"Tuning SAT checkers for Bounded Model Checking"** in *Proc. of 12th Intl. Conf. on Computer Aided Verification*, LNCS:1855, Springer-Verlag 2000

[WKS01] J. Whittemore, J. Kim, K. Sakallah. **"SATIRE: A New Incremental Satisfiability Engine"** in *Proc. 38th Conf. on Design Automation*, ACM Press 2001.

[ZM01] L. Zhang, C.F. Madigan, M.W. Moskewicz, S. Malik. **"Efficient Conflict Driven Learning in Boolean Satisfiability Solver"** in *Proc. of the International Conference on Computer Aided Design (ICCAD)*, 2001.

[3] http://www.opensat.org

Appendix – What is missing from Satzoo?

In order to reduce the size of MINISAT to a minimum, all non-essential parts of SATZOO/SATNIK were left out. Since SATZOO won two categories of the *SAT 2003 Competition*, we chose to present the missing parts here for completeness.

Initial strategies:

- *Burst of random variable orders.* Before anything else, SATZOO runs several passes of about 10-100 conflicts each with the variable order initiated to random. For satisfiable problems, SATZOO can sometimes stumble upon the solution by this strategy. For hard (typically unsatisfiable) problems, important clauses can be learnt in this phase that is outside the "local optimum" that the activity driven variable heuristic will later get stuck in.
- *Static variable ordering.* The second phase of SATZOO is to compute a static variable ordering taking into account how the variables of different clauses relates to each other (see *Figure 17*). Variables often occurring together in clauses will be put close in the variable order. SATZOO uses this static ordering for at least 5000 conflicts and does not stop until progress is halted severely. The static ordering often counters the effect of "shuffling" the problem (changing the order of clauses). The authors believe this phase to be the most important feature left out of MINISAT, and an important part of the success of SATZOO in the competition.[4]

Extra variable decision heuristics:

- *Variable of recent importance.* Inspired by the SAT-solver BERKMIN, occasionally variables from recent (unsatisfied) recorded clauses are picked.
- *Random.* About 1% of the time, a random variable is selected for branching. This simple strategy seems to crack some extra problems without incurring any substantial overhead for other problems. Give it a try!

Other:

- *Equivalent variable substitution.* The binary clauses are checked for cyclic implications. If a cycle is found, a representative is selected and all other variables in the cycle is replaced by this representative in the clause database. This yields a smaller database and fewer variables. The simplification is done periodically, but is most important in the initial phase (some problems can be very redundant).
- *Garbage collection.* SATZOO implements its own memory management which allows clauses to be stored more compactly.
- *0-1-programming.* Pseudo-boolean constraints are supported by SATZOO. This can of course easily be added to MINISAT through the constraint interface.

[4] The provided code currently has no further motivation beyond the authors' intuition. Indeed it was added as a quick hack two days before the competition.

```
void Solver.staticVarOrder()

  - Clear activity:
  for (int i = 0; i < nVars(); i++) activity[i] = 0

  - Do simple variable activity heuristic:
  for (int i = 0; i < clauses.size(); i++)
      Clause c   = clauses[i]
      double add = pow2(-size(c))
      for (int j = 0; j < size(c); j++) activity[var(c[j])] += add

  - Calculate the initial "heat" of all clauses:
  Vec<Vec<int>>          occurs(2*nVars())  - Map literal to list of clause indices
  Vec<Pair<double,int>> heat(clauses.size())    - Pairs of heat and clause index
  for (int i = 0; i < clauses.size(); i++)
      Clause c   = clauses[i]
      double sum = 0
      for (int j = 0; j < size(c); j++)
          occurs[index(c[j])].push(i)
          sum += activity[var(c[j])]
      heat[i] = Pair_new(sum, i)

  - Bump heat for clauses whose variables occur in other hot clauses:
  double iter_size = 0
  for (int i = 0; i < clauses.size(); i++)
      Clause c = clauses[i]
      for (int j = 0; j < size(c); j++) iter_size += occurs[index(c[j])].size()

  int    iterations = min((int)(((double)literals / iter_size) * 100), 10)
  double disapation = 1.0 / iterations
  for (int c = 0; c < iterations; c++)
      for (int i = 0; i < clauses.size(); i++)
          Clause c = clauses[i]
          for (int j = 0; j < size(c); j++)
              Vec<int> os = occurs[index(c[j])]
              for (int k = 0; k < os.size(); k++)
                  heat[i].fst += heat[os[k]].fst * disapation

  - Set activity according to hot clauses:
  sort(heat)
  for (int i = 0; i < nVars(); i++) activity[i] = 0

  double extra = 1e200
  for (int i = 0; i < heat.size(); i++)
      Clause& c = clauses[heat[i].snd]
      for (int j = 0; j < size(c); j++)
          if (activity[var(c[j])] == 0)
              activity[var(c[j])] = extra
              extra *= 0.995

  order.updateAll()
  var_inc = 1
```

Fig. 17. The static variable ordering of SATZOO. The code is defined only for clauses, not for arbitrary constraints. It must be adapted before it can be used in MINISAT.

Paper 4

Improved Subsumption Techniques for Variable Elimination in SAT

Niklas Eén, Armin Biere

Synopsis

A modern, clause based SAT solver uses DPLL style search to guide the creation of a resolution proof of unsatisfiability. The resolution steps take place in a *conflict analysis* phase, performed each time the search procedure has reached a conflicting assignment. However, DPLL search is not the only type of guidance that can be used when searching for a resolution proof. In this paper, a method is presented that applies resolution for the purpose of simplifying the current set of clauses ("the CNF"). By reducing the number of clauses and variables in the CNF by a greedy resolution guidance, the hope is that a DPLL search applied afterwards will complete the proof faster than it would on the original, unsimplified CNF.

Although the procedure is only evaluated as a pre-processor in the paper, it can be applied at any time during the solving. But regarded as a pre-processor, the simplification scheme can be viewed as an attempt to solve the important problem of generating good CNF encodings. The work was particularly motivated by the poor performance of the standard translation of *netlists* ("gate level circuits") into CNF by means of introducing an auxiliary variable for each internal point in the net. Typically, many variables can be eliminated in the translation, with a strict, syntactical gain in the resulting CNF size. The cost of this is that a more complicated CNF generator has to be tailored for the particular domain of interest. The resolution procedure presented in this paper achieves a similar gain, by more general methods, when applied to CNFs produced by an unoptimized CNF generator.

Improved Subsumption Techniques for Variable Elimination in SAT

Niklas Eén
Chalmers University of Technology
Göteborg, Sweden
een@cs.chalmers.se

Armin Biere
Swiss Federal Institute of Technology
Zürich, Switzerland
biere@inf.ethz.ch

Abstract. Preprocessing SAT instances can reduce the size of SAT problems considerably. We combine variable elimination with subsumption and self-subsuming resolution and present a new technique that avoids full clause distribution while eliminating functionally dependent variables. These techniques not only shrink the formula but also decrease run-time of SAT solvers substantially. We discuss critical implementation details that make the reduction procedure fast enough, such that it can also be applied dynamically during SAT solving. In comparison with related approaches we show in extensive experiments that our improvement produces smaller formulas, which are solved faster.

1 Introduction

SAT solvers are widely used in design verification, test pattern generation, and synthesis. Typically the solution to a problem in an application domain is obtained by translating it into a satisfiability problem and running a SAT solver on the generated propositional formula in conjunctive normal form (CNF).

The size of the generated formulas is often very large, particularly in the context of formal verification. In theory, a very large formula may be easy to solve and a small formula hard. However, in practice, it is often observed, that the run-time of a SAT solver is very much related to the size of the input formula, at least when the formulas stem from the same set of problems. This paper presents new techniques which reduce the size of a propositional formula by preprocessing in order to speed up overall SAT solving time. Our experiments show large speedups, particularly in the context of circuit verification.

2 Related Work

Modern SAT solvers use unit propagation and the pure literal rule in a preprocessing phase, as already described in the original DPLL algorithm [DLL62]. More sophisticated techniques [Stål89,Braf04,BW03,LS03] focus on deriving units, implications and equivalent literals. Similar techniques have been used in the context of ATPG.

Recently, the rule of *elimination of atomic formulas* from [DP60], which eliminates variables from a CNF by *clause distribution*, has been reconsidered as a basis for symbolic DPLL in the ZDD based SAT solver ZRES [CS00], as a way

to eliminate variables in the QBF solver QUANTOR [Bier04], and, independently in the preprocessor NIVER [SP04].

Compared to [SP04], we have added subsumption, self-subsuming resolution, and variable elimination by substitution, which together allow us to achieve much higher reduction rates and faster SAT solving. The description of the implementation of NIVER [SP04] stays on a very high level. We describe two implementation techniques for speeding up the process, based on (i) restricting the set of variables considered for elimination to *touched* variables and (ii) using *signatures* for fast subsumption checks. The latter has has also been used in [Bier04], but the focus of [Bier04] is QBF, and no experimental results for preprocessing SAT instances are given.

As is the case with [Bier04,SP04], our variable elimination techniques are orthogonal to those discussed in [Stål89,Braf04,BW03,LS03]. None of the latter considers subsumption.

Generally, our simplification techniques can be applied in three different ways: (i) during preprocessing (ii) during SAT-solving, e.g. at restart, at conflict clause generation, or (iii) between two incremental SAT problems. We will focus on applying simplification as a preprocessor, although a small study is included of an application in an incremental SAT problem.

3 Preliminaries

We assume that the formula is given in CNF. A CNF consists of a set of *clauses*, where each clause C is a set of literals. A literal is a boolean variable x or its negation $\overline{x}$.

Given two clauses $C_1 = \{x, a_1, \ldots, a_n\}$ and $C_2 = \{\overline{x}, b_1, \ldots, b_m\}$ the implied clause $C = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ is called the *resolvent* of the two original clauses by performing *resolution* on the variable x. We write $C = C_1 \otimes C_2$. This notion can be lifted to sets of clauses. Let S_1 be a set of clauses which all contain x, S_2 a set of clauses which all contain $\overline{x}$. Then $S_1 \otimes S_2$ is defined as

$$S_1 \otimes S_2 = \{C_1 \otimes C_2 \mid C_1 \in S_1,\ C_2 \in S_2\}$$

The basic simplification technique in this paper, and also in NIVER [SP04], follows [DP60] and simply eliminates variables. In a given CNF, let S_x be the set of clauses in which x occurs, $S_{\overline{x}}$ be the set of clauses in which $\overline{x}$ occurs and define $S = S_x \cup S_{\overline{x}}$.

The elimination of a variable x in the whole CNF can be computed by pairwise resolving each clause in S_x with every clause in $S_{\overline{x}}$. The produced resolvents $S' = S_x \otimes S_{\overline{x}}$ replace the original clauses S containing x or $\overline{x}$, resulting in a satisfiability equivalent problem. We refer to this procedure as elimination by *clause distribution*, and count only non-trivial clauses as part of the result. A clause is trivial if it contains a variable and its negation.

4 New Simplifications

In early experiments and also in the context of QBF [Bier04] we observed that clause distribution produces many subsumed clauses. A clause C_1 is said to

(syntactically) *subsume* C_2 if $C_1 \subseteq C_2$. A subsumed clause is redundant and can be discarded from the SAT-problem. Particularly, a subsumed clause never needs to be part of a resolution proof of unsatisfiability.

We also observed that often similar clauses of a particular kind occur: one clause C_2 almost subsumes a clause C_1, except for one literal $\overline{x}$, which, occurs with the opposite sign in C_2. For instance, let $C_1 = \{x, a, b\}$, and $C_2 = \{\overline{x}, a\}$, then resolving on x will produce $C_1' = \{a, b\}$, which subsumes C_1. Thus after adding C_1' to the CNF, we can remove C_1, in essence eliminating one literal. In this case, we say that C_1 is *strengthened* by *self-subsumption* using C_2. This simplification rule is called *self-subsuming resolution.*

As we will show in the experimental section, adding these subsumption techniques to variable elimination through clause distribution gives huge benefits compared to [SP04].

If a circuit is encoded in CNF, typically using the Tseitin transformation [Tsei68], then many variables are actually *functionally dependent* on other variables, particularly those introduced for gate outputs. In previous work, this information has been used to restrict the set of decision variables in a SAT solver to functionally *independent* variables [GMT02,OGMS02]. We use the information to simplify the CNF, essentially extracting gates as in [OGMS02]. Output variables of gates are functionally dependent on input variables. If the following three clauses

$$\ldots \{x, \overline{a}, \overline{b}\}, \{\overline{x}, a\}, \{\overline{x}, b\} \ldots \tag{1}$$

are part of a CNF then the AND gate $x = (a \wedge b)$ can be extracted, showing that x is functionally dependent. We also call this boolean equation a *definition* of x.

If x has a definition and is eliminated by clause distribution, many redundant resolvents are generated. By using the definition these clauses can be removed easily. Let G be the set of clauses used for extracting a *gate* with output x. Further recall that S_x is the set of clauses of S in which x occurs and similarly define G_x, and $G_{\overline{x}}$. Then the set S of all clauses with x or $\overline{x}$ can be partitioned into $S = G \cup R$, with $R \equiv S \backslash G$ the set of *remaining* clauses not used for extracting the gate. From $S = (G_x \cup R_x) \cup (G_{\overline{x}} \cup R_{\overline{x}})$ it follows that the set S' of all resolvents can be partitioned into $S' = S'' \cup G' \cup T'$ with

$$S'' = (R_x \otimes G_{\overline{x}}) \cup (G_x \otimes R_{\overline{x}}),$$
$$G' = G_x \otimes G_{\overline{x}}, \text{ and } R' = R_x \otimes R_{\overline{x}}.$$

Furthermore, we have the following Theorem, which shows that S'' implies G' and R', allowing S' to be replaced by S''.

Theorem. $S'' \models G' \cup R'$

The proof follows by first noticing, as in [GOMS04], that G' contains only trivial clauses. All the resolvents in R' can be obtained through several resolution steps (linear in the width of the gate or by just one hyper resolution step [BW03]) from clauses in S''. Another view is to substitute in R all occurrences of x by its definition (x by $a \wedge b$ and $\overline{x}$ by $\overline{a \wedge b}$ in the example) and then apply the distributivity law to obtain a flat CNF.

As a result, in the elimination of a functional dependent variable the clauses in G' and R' do not have to be added, which always reduces the number of added resolvents. We call this simplification rule *variable elimination by substitution*. To continue the example in Eqn. (1) , let S be

$$\underbrace{\overset{1}{\{x,c\}},\overset{2}{\{x,\overline{d}\}}}_{R_x},\underbrace{\overset{3}{\{x,\overline{a},\overline{b}\}}}_{G_x},\underbrace{\overset{4}{\{\overline{x},a\}},\overset{5}{\{\overline{x},b\}}}_{G_{\overline{x}}},\underbrace{\overset{6}{\{\overline{x},\overline{e},f\}}}_{R_{\overline{x}}}$$

The resolvents are

$$\overset{1\otimes4}{\{c,a\}},\overset{1\otimes5}{\{c,b\}},\overset{2\otimes4}{\{d,a\}},\overset{2\otimes5}{\{\overline{d},b\}},\overset{3\otimes6}{\{\overline{a},\overline{b},\overline{e},f\}}\ (S'')$$

$$\overset{3\otimes4}{\{\overline{a},\overline{b},a\}},\overset{3\otimes5}{\{\overline{a},\overline{b},b\}}\ (G') \qquad \overset{1\otimes6}{\{c,\overline{e},f\}},\overset{2\otimes6}{\{\overline{d},\overline{e},f\}}\ (R')$$

G' has only trivial clauses. Since trivial clauses are not counted, we have $|S'| = 7 > 5 = |S''|$. Replacing S with S'' results in a decrease of the number of clauses from 6 to 5, while the full clause distribution actually results in an increase from 6 to 7. Also note that the redundant clauses in R' can be obtained from S'' through two resolution steps each (actually by one hyper resolution step [BW03]): $1\otimes6 = (1\otimes4)\otimes((1\otimes5)\otimes(3\otimes6))$ and $2\otimes6 = (2\otimes4)\otimes((2\otimes5)\otimes(3\otimes6))$.

We also realized that subsumption sometimes removes clauses which could be used to extract a gate. For instance if the clause $C = \{\overline{a},\overline{b}\}$ is added to the CNF in Eqn. (1), then the clause $\{x,\overline{a},\overline{b}\}$ is removed and no AND gate can be extracted anymore. However, by one hyper resolution step, or two ordinary resolution steps, of C with the original two binary clauses the unit x can be derived, which, of course, simplifies the CNF even further. For all clauses C, we try to find binary clauses, that, if resolved with C in one hyper resolution step produce a unit. We call this simplification rule *hyper-unary-resolution*, similar to hyper-binary-resolution of [BW03].

5 Implementation

The techniques in this paper aim at simplifying a SAT problem by reducing its size. Although the simplification techniques can be used both on the original clauses of the SAT problem and on *learned* clauses [Söre04], we restrict the application to the original clauses. The set of learned clauses changes rapidly, and maintaining the indexing structure needed for efficient subsumption and variable elimination becomes costly.

Variable elimination is applied until no more improvement can be made to the clause database by a single elimination. Different notions of "improvement" can be used, and previous work [SP04] is focused on minimizing the number of *literal occurrences*. In our implementation we minimize the number of *clauses*. The rationale behind this is that propagation in a SAT solver is roughly proportional to the number of clauses, independent of their size.

5.1 Touched-lists

Subsumption and variable elimination interact, such that strengthening or removing a clause by (self-) subsumption can turn the elimination of a variable

into an improvement, and eliminating a variable, which produces new clauses, might give new opportunities for subsumption.

In our implementation, subsumption and elimination are alternated until a fixed-point is reached. To make this efficient, it is important not to loop repeatedly over all clauses. Therefore, three sets are maintained, storing information about the modifications made to the clause database:

Touched (set of variables). A variable is added to this set if it occurs in a clause being added, removed, or strengthened.

Added (set of clauses). When a clause is added to the SAT problem, it is also added to this set.

Strengthened (set of clauses). When a clause is strengthened (one literal is removed), it is added to this set.

These sets are *cleared* during the simplification procedure described in Sect. 5.3, then populated again as new clauses are produced during variable elimination, and while existing clauses are removed or strengthened by self-subsumption.

5.2 Subsumption

The efficiency of subsumption is most important and is achieved by two implementation techniques. First, for each clause a 64-bit *signature* is stored [Bier04]. The signature abstracts the set of literals of a clause in the following way: A hash function h maps literals to numbers 0..63, and the signature of a clause C is calculated as the bitwise OR of $2^{h(p)}$ over its literals $p \in C$. Then for each literal an *occur* list is maintained, pointing to all the clauses in which the literal occurs.

Now, *backward* subsumption, that is checking if a clause *subsumes* (as opposed to being *subsumed by*) some other clause in the database, can be implemented as follows:[1]

```
findSubsumed(Clause C)
  pick the literal p in C with the shortest occur list
  for each C' ∈ occur(p) do
      if (C ≠ C' && size(C) ≤ size(C') && subset(C, C'))
          add C' to result
  return result

subset(Clause C, Clause C')
  if (sig(C) & ~sig(C') ≠ 0) return FALSE
  else return result of iterating over C and C' in a
                complete (expensive) subset test
```

This algorithm is very fast and allows backward subsumption to be applied *eagerly* to each added or strengthened clause. We rely on this fact in Sect. 5.3. Given a procedure for finding subsumed clauses, we can now define a method for using a clause C to strengthen other clauses by self-subsumption:

[1] && denotes logical AND, & bitwise AND, and ~ bitwise negation.

```
selfSubsume(Clause C)
  for each p ∈ C do
      for each C' subsumed by C[p := p̄] do
          strengthen(C', p̄)                    – remove p̄ from C'
```

For the clause $\{a, b, c\}$ this method would call *findSubsumed*() for $\{\overline{a}, b, c\}$, $\{a, \overline{b}, c\}$, $\{a, b, \overline{c}\}$, and strengthen any result returned. It should be noted that the order of strengthening matters, but is not optimized in our implementation.

5.3 The top-level simplification method

```
simplify()
  do
      – SUBSUMPTION:
      S_0 = {set of clauses containing a literal occurring in
             some clause in Added}
      do
          S_1 = {set of clauses containing a literal occurring
                 negatively in some clause in Added}
                ∪ Added ∪ Strengthened
          clear Added and Strengthened
          for each C ∈ S_1 do selfSubsume(C)
          propagateToplevel()              – may strengthen clauses
      while (Strengthened ≠ ∅)
      for each C ∈ S_0 not deleted do subsume(C)
      – VARIABLE ELIMINATION:
      do
          S = Touched ; clear Touched
          for each x ∈ S do maybeEliminate(x)
                  – eliminated variables will touch other variables
      while (Touched ≠ ∅)
  while (Added ≠ ∅)
```

The method *subsume*(C) removes any clause subsumed by C, and similarly *selfSubsume*(C) removes a literal from any clause that may be strengthened using C. The method *maybeEliminate*(x) removes x by clause distribution or substitution if the number of clauses is reduced. Finally, *propagateToplevel*() removes any satisfied clause or false literal permanently from the clause database, assigning variables and repeating the process if unit clauses are produced.

In the subsumption phase, two sets are computed: S_0 for standard subsumption, and S_1 for self-subsumption. Self-subsumption is applied first as it may render more (standard) subsumptions possible.

Because backward subsumption is eagerly applied to all added or strengthened clauses, the only candidates for being *subsumed* are the clauses of *Added*. Strengthened clauses cannot be subsumed as they now have fewer literals and

were not subsumed before strengthening. A necessary condition for C to subsume C' is that C has at least one literal in common with C'. This motivates the definition of S_0.

Let "original clause" denote a clause not in *Added* or *Strengthened*. For self-subsumption (the set S_1) any added or strengthened clause can be used to remove literals from an original clause. Original clauses may self-subsume added clauses. This does not apply to strengthened clauses, since they have already been checked while still containing more literals. It remains to add to S_1 the original clauses that may strengthen a clause in *Added*. All the candidate clauses have to contain one literal $\overline{p}$ for some p in the added clauses.

5.4 Variable elimination

The variable elimination procedure relies on three readily implemented methods, which we state here without pseudo-code:

maybeClauseDistribute(x) eliminates x by clause distribution if the result has fewer clauses than the original (after removing trivially satisfied clauses).

findDefinition(x) returns either $x \leftrightarrow p_1 \vee p_2 \vee \ldots \vee p_n$ or $x \leftrightarrow p_1 \wedge p_2 \wedge \ldots \wedge p_n$ or NoDef. Unit information is also detected by hyper-unary-resolution and returned as $x \leftrightarrow$ True or $x \leftrightarrow$ False. Note that in general there may be many definitions. We use the shortest one and do not extract further information from this.

maybeSubstitute(*def*) takes the definition of a functionally dependent variable and substitutes each occurrence of the variable by its definition, provided this results in fewer clauses. Substituting a literal by a disjunction is unproblematic; substituting by a conjunction requires duplicating the destination clause for each literal of the conjunction, as explained in Sect. 4.

```
maybeEliminate( Var x)
  if (x assigned or has zero occurrences) return
  if (#occurs of x and ¬x are both > 10) return
  def = findDefinition(x)
  if (def ≠ NoDef) maybeSubstitute(def)
  else             maybeClauseDistribute(x)
  if (x was eliminated)
      propagateToplevel()
      remove learned clauses with x          – for incremental SAT
```

It was observed in an early implementation of the simplification procedure that on some problems the majority of time was spent on failed attempts to eliminate variables occurring frequently in both polarities. This is why these variables are heuristically excluded. The last line of the pseudo-code is only relevant in an incremental context; if variable elimination is applied during preprocessing, no learned clauses will exist.

Name	Original			NiVER				SatELite as NiVER				Full SatELite			
	#var (k)	#cla (k)	#lit (k)	#var (k)	#cla (k)	#lit (k)	time (sec)	#var (k)	#cla (k)	#lit (k)	time (sec)	#var (k)	#cla (k)	#lit (k)	time (sec)
6pipe	16	395	1157	15	393	1155	4.4	15	393	1155	2.2	**12**	**323**	**1018**	53.0
abp1-1-k31	15	48	124	8	34	98	0.6	8	33	94	0.3	**3**	**18**	**63**	1.2
barrel9	9	37	102	4	21	66	0.5	4	20	**65**	0.5	**2**	**16**	86	3.3
cache_10	227	880	2192	130	606	1680	20.6	92	417	1146	7.9	**29**	**178**	**748**	58.6
comb2	32	112	274	20	89	231	1.6	20	89	231	0.8	**3**	**18**	**63**	4.4
f2clk_40	28	80	186	10	44	125	1.4	7	32	90	0.5	**4**	**25**	**81**	1.2
fifo8_400	260	708	1602	69	301	859	13.6	42	164	451	6.5	**23**	**129**	**446**	11.2
guid-1-k56	99	307	758	45	193	553	3.9	44	189	540	3.1	**23**	**130**	**443**	8.0
ibm-03_k80	89	375	973	56	308	887	5.5	44	230	661	1.9	**28**	**190**	**629**	5.8
ibm-20_k45	91	373	945	46	281	832	6.7	41	250	725	2.1	**20**	**156**	**546**	7.0
ip50	66	215	513	34	148	398	5.1	12	50	**134**	1.6	**8**	**43**	139	4.2
longmult15	8	24	59	4	16	46	0.3	3	14	39	0.1	**1**	**8**	**27**	0.4
w08_14	120	425	1038	69	324	859	7.2	69	324	856	3.7	**34**	**220**	**688**	15.7

Table 1. *Size-reduction comparison with* NiVER. "#var", "#cla", "#lit" denote the number of variables, clauses, and literals in thousands respectivly. Times are in seconds as provied by the Unix command "time", and include parsing and outputing the result file. "SatELite as NiVER" uses no subsumption and has the same heuristic as NiVER for variable elimination (enforce fewer literals). It shows that SatELite can mimic the NiVER well, and that our techniques is a strict improvement. The last column shows SatELite with all reductions on.

5.5 Variable elimination related issues

The elimination of variables results in a partial model if the problem is satisfiable. Therefore, one must store the clauses removed during variable elimination and use them to complete the model. If the full model is not needed, removed clauses can simply be discarded.

Variable elimination also causes problems for the incremental SAT interface. Later extensions of the SAT instance might re-introduce eliminated variables, rendering the elimination unsound. Bringing back the removed clauses will solve the problem, but a simpler solution is to extend the solver interface to let the user explicitly prevent the elimination of selected variables.

6 Experimental results

The presented techniques were implemented in a version of MiniSat [ES03'] with an improved conflict clause analysis [Söre04]. The resulting tool was named SatELite, and is downloadable together with the benchmarks and all the result files produced during our testing.[2] The benchmarks were selected to be relevant for circuit verification.

MiniSat provides an incremental interface, and accordingly SatELite applies simplification techniques on each incremental SAT problem. For pure SAT benchmarks, simplification is only applied once as a preprocessing step.

***Study 1 – Comparing reduction rates with* NiVER.** This study shows that SatELite is an improvement over earlier work. We use the same problem set as presented in the NiVER paper [SP04]. The results are shown in *Table 1.*

[2] www.cs.chalmers.se/~een/SatELite

Study 2 – Measuring runtime effect of different techniques. To evaluate the benefit of different optimizations, we run two benchmark sets of SAT problems in CNF with SATELITE:

[**IBM**]. The first set is a subset of the huge BMC benchmark set made available by Emmanuel Zarpas at IBM.[3] We arbitrarily picked some of the directories resulting in 355 problems.

[**Mix**]. The second set contains a mix of hardware verification problems, obtained as follows: The available industrial problems of the SAT-2004 Competition were downloaded. Problems concerning graph coloring, set covering and planning problems were removed. We also removed *Miroslav Velev's* problems because SATELITE ran out of memory on all of them. These problems are huge but often easy, and represent a corner case we currently cannot handle. We added 18 satisfiable and 18 unsatisfiable BMC problems used in [ES03], mainly from the *Texas'97 benchmarks*;[4] 18 unsatisfiable BMC problems generated from the *PicoJava* design[5] and 13 liveness problems from *SatLib*.[6] The result contains 115 CNF files.

Figure 1 displays the effect of the different levels of reduction techniques in SATELITE. All benchmarks were run with 5 different settings of the preprocessor, using SATELITE's internal MINISAT based solver. In *Figure 2* we plotted a comparison of SATELITE with and without any preprocessing. We see a very consistent gain in using simplification. Finally, in *Table 2* the effects of preprocessing with ZCHAFF (version II) and BERKMIN as back-ends are shown. The results for BERKMIN are hard to interpret as it is a black-box solver.

Study 3 – Incremental BMC and k-induction. In *Table 3* and *Figure 3*, a study of applying simplification in an incremental context is presented. It is worth noting the somewhat erratic behavior of the procedure, which is not unusual for SAT based techniques.

7 Conclusion

New simplification techniques were presented together with important implementation details. On a large representative set of benchmarks it was shown, that they speed up SAT solvers considerably. We also believe that preprocessing techniques partially provide a solution to the important problem of generating *good* CNFs in various application domains. As future work, it would be interesting to combine our arsenal of simplification techniques with orthogonal approaches mentioned in the introduction.

Acknowledgements

Niklas Eén wants to thank Niklas Sörensson for setting him of in the direction of using self-subsumption in SAT.

[3] www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html
[4] www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/
[5] www.sun.com/microelectronics/communitysource/picojava/download.html
[6] www.cs.ubc.ca/ hoos/SATLIB/Benchmarks/SAT/BMC/bmc.tar.gz

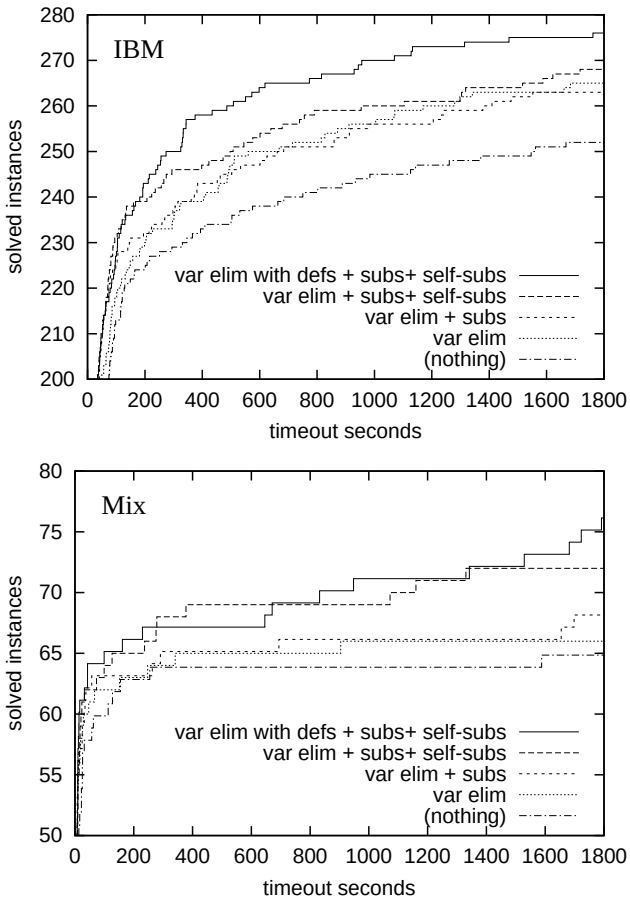


Fig. 1. *Comparing different simplification options using* SATELITE. Time includes everything; parsing, simplifying, and solving.

	IBM			Mix		
	SATELITE	ZCHAFF	BERKMIN	SATELITE	ZCHAFF	BERKMIN
plain	252	233	221	65	69	76
ve	265	237	253	66	69	78
ve s	263	253	252	68	73	79
ve s ss	268	257	253	72	74	82
ve s ss ds	276	242	251	76	76	82

Table 2. *Total number of solved problems.* The table displays the number of solved problems after 1800 seconds using different solvers as back-end. The abbreviations are: "ve" variable elimination, "s" subsumption, "ss" self-subsumption, "ds" definitional substitution.

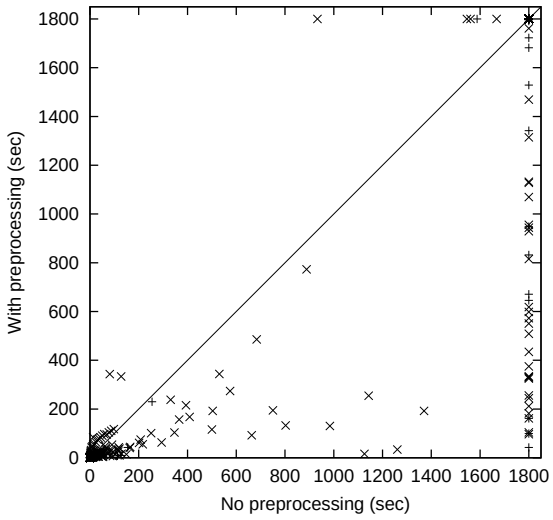


Fig. 2. *Heads-up comparison, with and without preprocessing.* The graph shows solving times for IBM problems (×) and the industrial mix (+) using SATELITE with and without preprocessing. Marks below the diagonal means faster solving times using preprocessing. A timeout of 1800 seconds was used, and a mark along the edge represents a test which timed out for one of the two executions.

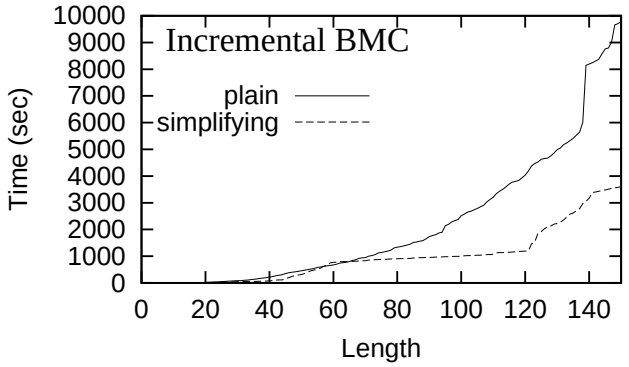


Fig. 3. *Case study on incremental BMC.* We ran TIP's incremental BMC algorithm on a number of problems produced from the Pico Java designs. A representative example is plotted her. Occasionally the SAT solver would get stuck on a seemingly random step and spend a lot of time there, then recover and continue at a faster pace again. In this particular experiment, the size of each incremental SAT instance was about 3.5 bigger in the non-simplifying case.

Name	Depth	Plain	Simplifying
vis.prodcell_12	29	134.4	29.4
vis.prodcell_14	16	21.2	5.8
vis.prodcell_15	23	65.4	17.1
vis.prodcell_17	27	117.8	37.6
vis.prodcell_18	13	14.7	4.0
vis.prodcell_19	22	39.7	13.6
vis.prodcell_23	13	20.9	4.9
vis.prodcell_24	37	235.2	77.0

Table 3. *Study on k-induction.* We modified TIP [ES03] to use SATELITE as a back-end and ran the *zigzag* incremental induction algorithm on the "prodcell" problem distributed with VIS. In the rightmost column, simplification was invoked between each incremental step.

References

[Bier04] A. Biere. Resolve and expand. In *Prel. Proc. SAT'04.*

[Braf04] R. I. Brafman. **A simplifier for propositional formulas with many binary clauses.** *IEEE Trans. on Systems, Man, and Cybernetics*, 34(1), 2004.

[BW03] F. Bacchus and J. Winter. **Effective preprocessing with hyper-resolution and equality reduction.** In *Proc. SAT'03.*

[CS00] P. Chatalic and L. Simon. **ZRes: The old Davis-Putnam procedure meets ZBDDs.** In *Proc. CADE'00*, #1831 in LNAI.

[DLL62] M. Davis, G. Logemann, and D. Loveland. **A machine program for theorem proving.** *Comm. of the ACM*, 5(7), 1962.

[DP60] M. Davis and H. Putnam. **A computing procedure for quantification theory.** *Journal of the ACM*, 7(3), July 1960.

[ES03] N. Eén and N. Sörensson. **Temporal induction by incremental SAT solving.** *ENTCS*, 89(4), 2003.

[ES03'] N. Eén and N. Sörensson. **An extensible SAT solver.** In *Proc. SAT'03*, volume 2919 of *LNCS*. Springer, 2004.

[GMT02] E. Giunchiglia, M. Maratea, and A. Tacchella. **Dependent and independent variables for propositional satisfiability.** In *Proc. JELIA'02*, volume 2424 of *LNCS*. Springer, 2002.

[GOMS04] É. Grégoire, R. Ostrowski, B. Mazure, and L. Saïs. **Automatic extraction of functional dependencies.** In *Prel. Proc. SAT'04.*

[LS03] I. Lynce and J. Marques-Silva. **Probing-based preprocessing techniques for propositional satisfiability.** In *Proc. ICTAI'03.*

[OGMS02] R. Ostrowski, É. Grégoire, B. Mazure, and L. Saïs. **Recovering and exploiting structural knowledge from CNF formulas.** In *Proc. CP'02*, volume 2470 of *LNCS*, 2002.

[SP04] S. Subbarayan and D. K. Pradhan. **NiVER: Non increasing variable elimination resolution for preprocessing SAT instances.** In *Prel. Proc. SAT'04.*

[Stål89] G. Stålmarck. **A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula.** Swedish Patent N° 467 076 (1989).

[Söre04] N. Sörensson paper in preparation, August 2004.

[Tsei68] G. Tseitin. **On the complexity of derivation in propositional calculus.** In *Studies in Constr. Math. and Math. Logic*, 1968.