Thesis for the Degree of Doctor of Philosophy

Effective SAT Solving

Niklas Sörensson

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden

Göteborg, September 2008

Effective SAT Solving
Niklas Sörensson
ISBN 978-91-628-7612-8

Technical report no. 45D
Department of Computer Science and Engineering
Division of Software Engineering and Technology

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31–772 1000

Printed at Chalmers, Göteborg, Sweden, 2008

Abstract

A growing number of problem domains are successfully being tackled by SAT solvers. This thesis contributes to that trend by pushing the state-of-the-art of core SAT algorithms and their implementation, but also in several important application areas. It consists of five papers: the first details the implementation of the SAT solver MiniSat and the other four papers discuss specific issues related to different application domains.

In the first paper, catering to the trend of extending and adapting SAT solvers, we present a detailed description of MiniSat, a SAT solver designed for that particular purpose. The description additionally bridges a gap between theory and practice, serving as a tutorial on modern SAT solving algorithms. Among other things, we describe how to solve a series of related SAT problems efficiently, called incremental SAT solving.

For finding finite first order models the MACE-style method that is based on SAT solving is well-known. In the second paper we improve the basic method with several techniques that can be loosely classified as either transformations that make the reduction to SAT result in fewer clauses or techniques that are designed to speed up the search of the SAT solver. The resulting tool, called Paradox, won the SAT/Models division of the CASC competition in 2003 and has not been beaten since by a single general purpose model finding tool.

In the last decade the interest in methods for safety property verification that are based on SAT solving has been steadily growing. One example of such a method is temporal induction. The method requires a sequence of increasingly stronger induction proofs to be performed. In the third paper we show how this sequence of proofs can be solved efficiently using incremental SAT solving.

The last two papers consider two frequently occurring types of encodings: (1) the problem of encoding circuits into CNF, and (2) encoding 0-1 integer linear programming into CNF and how to use incremental SAT to solve the intended optimization problem.

There are several encoding patterns that occur over and over again in this thesis but also elsewhere. The most noteworthy are: incremental SAT, lazy encoding of constraints, and bit-wise encoding of arithmetic influenced by hardware designs for adders and multipliers.

The general conclusion is: deploying SAT solvers effectively requires implementations that are efficient, yet easily adaptable to specific application needs. Moreover, to get the best results, it is worth spending effort to make sure that one uses the best codings possible for an application. However, it is important to note that this is not absolutely necessary. For some applications naive problem codings work just fine which is indeed part of the appeal of using SAT solving.

Acknowledgments

I'd like to thank everybody that has in some way helped me during my PhD studies. In particular, I would like to thank some without whom this work would never have come to be:

My supervisors Koen Claessen and Reiner Hähnle, for being as responsible and supportive as you could wish supervisors to be, and for mastering the delicate art of applying just the right amount of pressure.

A bit outside the normal responsibilities of a supervisor, Reiner's influence has helped me learn to appreciate Wine.

Koen, for his friendship, enthusiasm[1], and ability to discuss any topic, technical or not.

Mary Sheeran, for fostering the special atmosphere in the Formal Methods group, and stimulated a whole generation of PhD students to be interested in SAT solving.

Niklas Een, for being a great friend and the best co-worker I've had the pleasure to work with. I hope that we will have more chances to collaborate in the future. As a sign of my humble gratitude: I forgive you for forcing me to use (and appreciate) an indentation depth of 4 characters.

My family and friends, who has always encouraged me even when prospects were bleak. In particular, my beloved Cissi, whom have had to endure a lot during my efforts to write this thesis. Examples include, fits of frustration and/or despair, sleepless night, financial problems, missed vacations. The list goes on, but you have stayed with me. For that I'm forever grateful.

[1] "Det låter jätteintressant! Men..."

Contents

Chapter 1

Introduction

In formal logics there is a conflict between expressivness of logics on one end, and the ease with which the decision problems for said logics can be solved. The more expressive a logic is the harder it becomes to automatize. In the lower end of this scale there exist logics with decision procedure that run in polynomial time, but the expressiveness of such logics is usually very limited. At the other end of the scale, most higher-order logics are expressive enough to be the foundation for all of mathematics but they are then necessarily incomplete, in the sense that there are bound to be properties that appear to be true but are impossible to prove for any given (consistent) axiomatization. This remarkable fact is usually referred to as "Gödel's incompleteness theorem".

This thesis is mainly concerned with propositional logic, but it touches on issues in temporal logic and first order logic (FOL) as well. Propositional logic is in the lower end of the expressiveness scale: it allows you to state simple boolean properties constructed using *variables* and basic logical conjunctives such as *and*, *or*, *not*, and *implies*.

Propositional logic is expressive enough to be useful in practice, but there is no known algorithm for its decision problem that executes in polynomial time. In fact, it was the first problem proven to be NP-complete [Coo71] which essentially means that it can be seen as the canonical example of a problem that is "hard to solve". Nonetheless, considerable progress in practical terms was made in solving the satisfiability problem of propositional logic (called simply the "SAT problem" from now on) in the last decade. The SAT problem is at the core of this thesis and is discussed in Section 1.1 below.

Temporal logic increases expressivity compared to propositional logic in that it allows us to talk about time. For example, it is possible to model stateful system that evolve as time progresses. The price for this expressivity is that decision complexity is harder than for propositional logic, typically, PSPACE. The most popular technique for reasoning about temporal logic is called *Model Checking*.

First order logic goes beyond propositional logic in that one can model relations and functions over individual objects and, more importantly, it is possible to quantify over individuals. This capability pushes the first order decision problem into the realm of the undecidable, however, there exist various syntactic restrictions that render FOL decidable again.

It is important to note that first order logic is *semi-decidable*: given a first

order theorem, it is possible to prove it automatically in finite time. This is *not* possible for non-provable statements in general. This has the important consequence that finding counter examples for invalid first order statements is inherently hard. The crux of the matter is the (necessary) existence of statements that can only be refuted with infinitely large counter examples. A natural restriction is the limitation to finite counter examples, or even counter examples with a fixed bound.

It is well known from theoretical logic that logics can be encoded into each other and coding techniques have been used to prove a number of theoretical results. Only in the last years, however, coding techniques came into focus as a means to achieve practically useful algorithms. Thus, coding expressive features in a less expressive logic can been seen as (superficially) increasing the expressiveness of a logic, at the cost of lowering reasoning to potentially unnatural levels. A good example of this is finite arithmetic, which can be coded at the bit-level in propositional logic.

We say more on model checking, first order finite model finding, and encoding of logics in Section 1.2.

1.1 SAT

The satisfiability (SAT) problem is the problem of deciding whether the variables of a propositional formula can be assigned in such a way that the formula evaluates to true. The research area devoted to this problem is today very active as there have recently been an increased general interest in SAT. However, research into SAT algorithms has been going on for long time now. This section gives a short historical background on SAT research, followed by a description of our first contribution to the field, the SAT solver MiniSat.

1.1.1 The Origins of SAT Solving

During the late 50's and early 60's a lot of the pioneering work on automated theorem proving was made possible by both the fact that general purpose computers were recently made readily available, and also, that computability had been anticipated in academia for a long time, and a lot of the theoretical ground-work had already been done. Among those experimenting with the first implementations of theorem provers were Martin Davis and Hilary Putnam, who wrote the paper describing what is believed to be the first program solving the SAT problem[DP60]. Although this paper really tackles first order theorem proving, it is best known and appreciated for the groundwork on SAT. The algorithm is based on quantifier elimination and removes the variables one-by-one from the problem, until either an empty clause is derived, which means the problem is unsatisfiable, or, all variables have been removed, meaning the problem is satisfiable.

The problem with this algorithm is that repeated quantifier elimination is very likely to grow the problem expontentially. Moreover, since computer memory was a particularily scarce resource at the time there was a necessity to trade memory for running time. This was part of the aim of the follow up work by Martin Davis, George Logemann and Donald W. Loveland[DLL62]. The algorithm they proposed for solving the SAT problem has some of the most crucial

aspects of modern SAT algorithms, such as branching/backtracking and unit propagation. This allows the algorithm to run in memory linear to the size of the problem, but may still take exponential time. For a more detailed elaboration, see the chapter *The early history of Automated Deduction* of [RV01].

1.1.2 The DPLL algorithmic framework

The SAT algorithm usually referred to as DPLL should rather be seen as a framework of algorithms. The unique features that characterize DPLL are *branching*, *backtracking*, and *unit propagation*. These elements are tied together in a recursive procedure that systematically investigates the complete boolean state space in the following way: *branching* means that the problem is split into two subproblems, where a certain variable has been assigned true and false respectively. These subproblems can then be solved independently in any order. If there is some clause that is false given the current set of assignments, that part of the search space cannot contain any solutions. It is then necessary to *backtrack*, undoing all variable assignments, until the latest branch point that has not been tried with both values.

Branching and backtracking are in fact enough to form a complete procedure, in the sense that it will always find a solution, but it will be very inefficient. The problem is that, given a certain set of assignments to variables induced by a branch in the search, there may be a number of apparent consequential assignments to variables, and it would be very bad to try both values for them. *Unit propagation* is a procedure that derives such consequences for a very specific definition of "apparent": namely given that all literals but one of a clause has been given the value false, then one can conclude that the remaining literal must have the value true.

Branching Heuristics The most obvious degree of freedom is in which order to branch on variables. Over the years a large number of branching heuristics have been devised, but for a long time they were all quite similar in the sense that they used syntactic properties of the problem as guides. For example, by choosing the variable that satisfies the largest number of unresolved clauses (clauses that still contain unassigned literals), one might expect to quickly reduce the problem and possibly find a satisfying assignment faster. These kind of measures was usually weighted in some way with respect to how easily clauses could be satisfied, where satisfying a "hard" clause was worth more than an "easy" clause. A prominent example of this kind of heuristics is the Jeroslow-Wang heuristic [JW90], which worked relatively well at the time of it's inception. The motivation for why this heuristic worked well was later refuted by Hooker et al [HV95], which serves as a good example of the notorious difficulty in understanding and evaluating heuristics.

Another complication in the investigation of heuristics is the interdependence between different components of the algorithms. For instance, João Marques-Silva[Sil99] comes to the conclusion that the branching heuristic does not matter much in the precence of a modern learning implementation (outlined below), at least considering the heuristics available at the time.

This syntactic type of heuristics were dominant until the end of the nineties, when there was a major break-through with the arrival of the SAT solver CHAFF[MMZ+01b]. The branching heuristics introduced in CHAFF, named

Variable State Independent Decaying Sum (VSIDS), had two unique properties that made it very well adapted to the rest of the DPLL algorithm: (1) As the name implies it is not dependent on the current assignments to variables in a particular branch. This means that no extra work has to be done during unit propagation or backtracking, which makes these key procedures faster. In particular, it is no longer necessary to recognize exactly which clauses are unresolved, a fact that is crucial for allowing the particular implementation of unit propagation outlined below. (2) Instead of depending on syntactic properties of the problem, it is closely tied to the set of learned clauses. Informally, variables that occur often in recently learned clauses are preferred. The intention is that this will steer the search into a direction making the most out of the current set of learned clauses.

Another similar variable heuristic, the Berkmin heuristic [GN02], is based on the observation that the reason that the VSIDS heuristic works so well is that it tends to branch on variables occuring in recent clauses. The Berkmin heuristic achieves the same effect more directly by examining recently learnt clauses for candidate branching variables.

Efficient unit propagation Until recently, unit propagation used to be implemented with straight-forward techniques. However, the invention of clever data structures has changed this. *Head-tail lists* were introduced in SATO [ZS00] in order to reduce the amount of work in unit propagation. Most modern SAT solvers today use a method called *two literal watching*, pioneered in Chaff [MMZ$^+$01b], which uses a similar datastructure to a somewhat better effect [LS05].

Learning With only the mechanisms outlined above it is very likely that a DPLL search procedure will run into conflicts that are essentially the same, in the sense that they depend on the same assignments, over and over again. To cope with this redundancy it is standard practice to deploy a learning procedure [SS96] that for each conflict derives a new learned clause, such that unit propagation alone will avoid the same conflict in the future. This learned clause is derived using *conflict analysis*, which essentially traces the reasons for assignments involved with the conflict. As a side effect the conflict analysis may reveal that there was in fact a number of assumptions not involved in the conflict, that can be skipped during backtracking. The branches for the other polarity of these assumptions can be skipped entirely, and this process is usually referred to as *non-chronological backtracking*.

1.1.3 MiniSat

MiniSat and the accompanying paper that describes its implementation (Chapter 2) was first introduced as a write-up for Satzoo, one of the winning entries of the 2002 SAT competition. However, it was actually re-written from scratch (drawing from both Satzoo and Satnik), and significantly simplified in the process. The aim was to produce a SAT solver that had state-of-the-art performance, and yet was small enough that essentially all the code could be included in full detail in the paper. The authors felt at the time that there was a need to provide more details on what was required to implement a modern SAT solver

— details that newcomers had to either figure out themselves or extract from reading the source-code of freely available solvers such as CHAFF and LIMMAT. Another goal was to make the implementation of MiniSat as easy as possible to pick up by others, and start hacking on it too. This was partly achieved by the simple design, but it was also important to distribute the source-code under an unrestrictive license.

Even though the implementation was minimalistic, we managed to sneak in some features outside of the core algorithms. Firstly, we provided a simple API to extend the solver with new types of boolean constraints, similar to the standard practice in the field of constraint programming. Secondly, we introduced another API for *incremental* SAT solving that was simpler than what had existed before, but still powerful enough to encode both addition and removal of clauses using a bit of extra coding.

While the first version of MINISAT was not quite as efficient as the solvers it was derived from, it was never very far behind. Moreover, it proved to be a very sound foundation to build further work on, as it has since caught-up with and significantly outperformed its predeccessors.

For an overview of the significant impact of MINISAT, see Section 1.3.

1.2 Applications

Apart from the design and implementation of core SAT algorithms, this thesis also consists of several investigations into different application areas where SAT-solving plays a central role. Each of the four chapters after Chapter 2 is concerned with one such application area.

First Order Finite Model Finding As the dual to automated theorem proving for first order logic, *model finding* is a very important tool. In general, there are no complete automated methods for this, but finding models with finite domains is possible to do automatically. Chapter 3 is based on the paper *New Techniques that Improve MACE-style Finite Model Finding*, written together with Koen Claessen in 2003. The paper introduces significant improvements over the conventional way of finding finite models using a SAT solver, which we called MACE-style model finding, after McCune's model finding tool MACE2 [McC94]. We implemented our method in the tool PARADOX, which has consistently won the CASC competition in its division every year since 2003. Alternative methods for finite model finding are based on a direct search, and are implemented in tools such as SEM [JZ96] and MACE4 [McC03]. These methods can outperform SAT-based methods in some problem areas, but on the whole SAT-based model finding is very competitive.

SAT-based Model Checking For circuits with sequential behaviour *model checking* proved to be a highly successful verification technique. It is probably the single most successful formal verification procedure, as is demonstrated by the fact that some of the pioneers[1] in the area of Model Checking were recently given the Turing-award. Chapter 4 is based on the paper *Temporal Induction by Incremental SAT Solving*, written together with Niklas Eén in 2003. This

[1] Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis

paper describes in detail how an *induction based* model checking algorithm can be implemented using incremental SAT solving. The main challenge with this type of algorithm is that for each property the circuit may have to be unrolled an unknown number of times to a certain *depth*, in order to make the induction hypothesis strong enough. Before our paper it was standard practice to make an arbitrary guess and, if the depth was too small to prove the property, one would guess a larger depth and repeat the process. In the paper, we show that it is feasible to try *all* depths one after another using an incremental procedure. This achieves a greater level of automation as the user does not have to make any guesses.

Circuit to CNF encoding SAT solvers usually accept input formulas in Conjunctive Normal Form (CNF). However, it is commonly the case that problem domains require reasoning about circuits, or simply need the flexibility given by general propositional formulas. In these cases it is possible to translate a circuit or a propositional formula to an equivalent formula in CNF, but these translations are not very useful as they will increase the formula size exponentially in general. This problem is possible to get around with translations that are allowed to introduce new fresh variables for internal points in the circuit or formula. The most well-known and often-used such translation is the Tseitin [Tse68] transformation which is very simple yet useful in practice. Chapter 5 is based on the paper *Applying Logic Synthesis for Speeding Up SAT*, co-authored by Niklas Eén and Alan Mishchenko in 2007. In this paper we explore two techniques for Circuit to CNF encoding that are more elaborate than the Tseitin transformation with the intention to produce encodings that are smaller. The first technique, *DAG-Aware minimization*, is used as a pre-processing step and is designed to reduce redundancy in the circuit itself. This technique is not new, but its effect on runtimes of SAT solving is isolated and presented in more detail than what has been done before. The second technique, the actual CNF encoding, is novel, and it puts standard FPGA *technology mapping* algorithms to the use of minimizing the number of clauses in the result. The appeal of this approach is that the heuristics used are of a more *global* flavour compared to other commonly used improvements to the Tseitin transformation.

0-1 Integer Linear Programming The optimization of a linear function, subject to a set of linear constraints, is referred to as *linear programming*. An important special case is when the variables of the objective function and constraints are restricted to be either zero or one. The resulting type of problem is known as *0-1 integer linear programming* (ILP) and it has a certain propositional flavour to it. Therefore, it is natural to investigate the relation between 0-1 ILP and the SAT problem. Chapter refcha:minisatplus is based on the paper *Translating Pseudo-Boolean Constraints into SAT* co-authored by Niklas Eén in 2005. In this paper we investigate several ways of translating 0-1 linear constraints into CNF. We were at first interested in how a translation based approach could compare to methods that were based on extenstions of SAT algorithms and produced a solver called MiniSat+. As the results seemed promising, we were intrigued enough to try to better understand the relative merits of the different encodings we used. One important conclusion was that while the size of the CNF clearly matters, a smaller encoding is not always

better. In such cases, the notion of *arc-consistency* helps explain why a more verbose encoding may be preferable. Essentially, an encoding that preserves arc-consistency makes unit propagation more powerful which in turn avoids useless repetition for the SAT solver. We then prove that the novel translation technique based on sorter networks is arc-concistent for the special case of linear constraints known as *cardinality constraints*.

1.3 Contributions

The release of the source-code of MiniSat and the paper documenting the implementation [ES03a] have been well received in the SAT community. There are three reasons for this: (1) MiniSat it is easy to understand, (2) it is easy to adapt or extend with new features, and (3) its efficiency has remained state-of-the-art within the academic setting. The following provide some evidence of these points:

Winning SAT Competitions MiniSat first won the industrial track of the SAT competition in 2005 together with the CNF preprocessor SatElite. In 2006 MiniSat 2.0 won the light-weight SAT competition in the SAT-Race organized in between the bi-annual SAT competition. The main new feature was a reimplementation of the preprocessor from SatElite but the efficiency of the resulting solver was mostly the same as the previous year. The coming year MiniSat had not been updated at all and in the next SAT competition the rest of the world had caught up and improved upon MiniSat. The now rather old implementation still got second and third place in three of the industrial tracks of the year 2007 competition. In 2008, MiniSat 2.1 again won the main track of the SAT-Race.

Usages or extensions MiniSat is used in a large number of applications and tools within automated theorem proving as well as software- and hardware verification. One of the earliest adopters is the Satisfiability Modulo Theories (SMT) solver MathSat [BBC$^+$05]. Another similar system that extends MiniSat is OpenSMT [BS], which also shares the MiniSat-philosophy of providing a simple, open, and easy-to-extend architechture for the benefit of the research community. In first-order theorem proving, instantiation-based methods currently have a renaissance and they are built upon SAT solvers. The at present strongest instantiation-based system iProver [Kor08] is using MiniSat.

Language ports The small code-base of MiniSat has made it feasible to make feature-complete translations to programming languages other than C++. For instance, SAT4J [Ber] is a comprehensive modular Java library for SAT solving and other related combinatorial problems. The part providing a configurable SAT solver contains a translation of many MiniSat algorithms (some parts could apparently almost be copied as-is). Another example is the C-version of MiniSat; while originally written by the MiniSat authors, this has been picked up and improved in the system ABC [SG] by Alan Mishchenko. Another, apparently rather straightforward, port to C# also exists [Mos07].

From a few years of experience with MiniSat, it can be concluded that a SAT solver does not have to be complicated to be efficient. In some sense, this can be interpreted as the superiority of brute force approaches over cleverness. So in effect, the simplicity of the algorithms comes at the price that it is very hard to improve upon the current state-of-the-art. This can be seen by the fact that progress on core algorithms and implementations is rather slow, even with the increased interest the SAT community has enjoyed in the past few years. Put simply, there are not a lot of low hanging fruit in the domain of core algorithms for SAT solving.

Besides the open architecture of a SAT solver, an additional category of contributions of this thesis lies in the development of encoding techniques for a number of different applications (Chapters 3– 6). Taken as a whole, this has helped to emphasize the importance of encodings within the SAT community. There are several encoding patterns that occur over and over again. The most noteworthy are: incremental SAT, lazy encoding of constraints, and bit-wise encoding of arithmetic influenced by hardware designs for adders and multipliers. These have shown themselves to be extremely useful tools in the effective use of a SAT-solver for tackling problems in different areas, and should be part of the arsenal of anyone attacking a new problem area.

1.4 Discussion

We close this chapter with some critical considerations on the state of affairs with regard to the Scientific Method in the subcommunity of Computer Science where our work takes place. This should in no way be understood as criticsm of any individual researchers or research group, but as a (self-)critical reflection on how scientific progress can be ensured better than it is currently done.

The issue we would like to discuss is the transitional difficulties that the formal verification community is suffering from, by having gone from a purely theoretical field (as a part of mathematical logic) to a more mixed discipline with strong flavours of engineering and experimental science. It seems that the attitude of the community has not fully adapted to accomodate the additional principles that have to be followed. A publication in theoretical computer science can be verified simply by reading and understanding the material, but results based on experiments require that the experiments can be repeated by others.

Repeatability Experimental results may be hard to repeat for different reasons: (1) The conclusions only apply to a very narrow set of problems. (2) They are dependent on factors not mentioned in the paper. This in turn may be either for space restriction (i.e. page-limit) reasons, or simply because the authors did not know that the dependency exists. (3) Errors were introduced in the (original) evaluation procedure itself.

It is important, that given a failure to repeat a certain result, one should be able to determine whether the error is on ones own behalf or one of the cases above holds. In the field of formal verification, for this to be possible, both benchmarks and implementations used in the experiments need to be published; otherwise, you cannot rule out both (1) and (2). This usually poses a problem to researchers who work in (or in collaboration with) industry. Benchmarks

that are derived from customers, or otherwise contain sensitive data, are simply impossible to share with the public. The implementations used in experiments are often part of some product, and cannot be shared in order to maintain trade secrets.

What could be done? Note that it is by no means only experimental industrial research that lacks the accompanying benchmarks or source code. It seems the community would do well to foster an attitude change so that researchers working in a non-commercial context always provide all the necessary means to reproduce the experiments of a publication. It is, of course, important to accomodate the justified interests of industrial research. Clearly, academic research benefits from tight communication with industry and severing that connection is counter-productive.

One possibility to improve the situation is to reinforce the importance of gathering and maintaining a public body of benchmarks for different problem categories. There already exist such efforts for different subcommunities[2], but there is room for improvement as such collections rarely are comprehensive.

Another trend that is improving on this situation is the growing number of high quality, open, implementations of various theorem proving software of which MiniSat is but one example. This is makes it possible, at least in principle, to implement new ideas on a basis that is independent of trade secrets. Such practice allows disclosure of implementations in more cases.

[2] A good example is Thousands Of Problems For Theorem Proving (TPTP) [SS07].

Chapter 2

An Extensible SAT-solver

Niklas Eén, Niklas Sörensson
Chalmers University of Technology and Göteborg University

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.

2.1 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 [BCC+99, 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 [ARMS02a, ES03b].

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 [SS96], together with watched literals and dynamic variable ordering [MMZ+01b]. The original C++ source code (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 [ES03b]. 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.

2.2 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

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

assumed (see for instance [SS96]). The types ***var***, ***lit***, and ***Vec*** for variables, literals, and vectors respectively are explained in detail in section 2.4.

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 [ES03b].

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.

2.3 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*) [SS96], and boolean constraint propagation (BCP) using *watched literals* [MMZ+01b].[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 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 [MMZ+01b]. For each literal, a list of constraints is kept. These are the constraints that *may* propagate unit information (variable assignments)

[2] This includes SAT-solvers such as: zChaff, Limmat, Berkmin.

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 [SS96]. 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=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* [SS96]).

```
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.

Although this is a good default order, domain specific heuristics have successfully been used in various areas to improve the performance [Str00]. Variable ordering is a traditional target for improving SAT-solvers.

Activity heuristics. One important technique first introduced with CHAFF [MMZ+01b] 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.

2.4 Implementation

The following conventions are used in the code. Atomic types start with a lower-case 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 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 2.1*.

2.4.1 The solver state

A number of things need to be stored in the solver state. *Figure 2.2* 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 2.3*, as well as the interface of *VarOrder* (*Figure 2.4*), explained in section 2.4.6.

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 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.

2.4.2 Constraints

MiniSat can handle arbitrary constraints over boolean variables through the abstraction presented in *Figure 2.5*. Each constraint type needs to implement

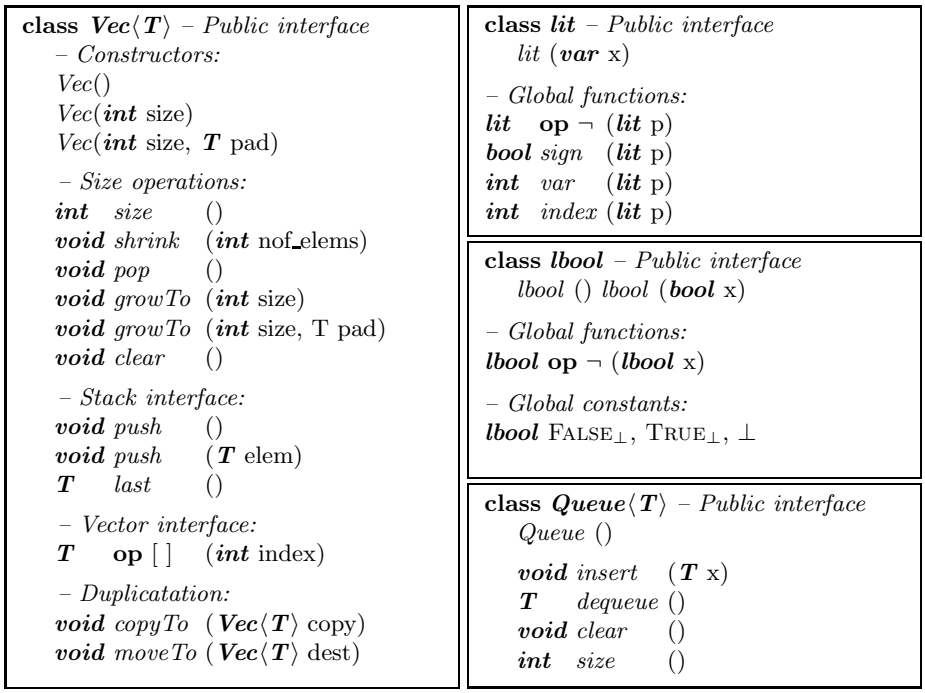


Figure 2.1: *Basic abstract data types used throughout the code.* The vector data type can push a default constructed element by the *push*() method with no argument. The *moveTo*() method will move the contents of a vector to another vector in constant time, clearing the source vector. The literal data 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 $\overline{x}$).

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

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.*

Figure 2.2: 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()

Figure 2.3: 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.*

Figure 2.4: 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
```

Figure 2.5: Abstract base class for constraints.

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.

The code for the *Clause* constraint is presented in *Figure 2.7*. 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 2.6*).

```
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
```

Figure 2.6: Creates a new SAT variable in the solver.

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.

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.

2.4.3 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.

An implementation of this procedure is displayed in *Figure 2.9*. 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.

2.4.4 Learning

This section describes the conflict-driven clause learning. It was first described in [SS96] 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.

```
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 2.8"

  – 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)
```

Figure 2.7: 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
```

Figure 2.8: 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.)

```
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
```

Figure 2.9: ***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).

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 [ZMMM01]. 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 2.10*, 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. Fur-

thermore, 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 $\bot_{lit}$.

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 [SS96], and is referred to as *back-jumping* or *non-chronological backtracking* in the literature.

2.4.5 Search

The search method in *Figure 2.13* works basically as described in section 2.3 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 conflicts. If failing to solve the SAT-problem within the bound, all assumptions will be canceled and $\bot$ 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

```
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
```

Figure 2.10: 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)
```

Figure 2.11: Record a clause and drive backtracking. Pre-condition: 'clause[0]' must contain the asserting literal. In particular, 'clause[]' must not be empty.

```
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()
```

Figure 2.12: ***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.

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 [ES03b, 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()*.

2.4.6 Activity heuristics

The implementation of activity is shown in *Figure 2.14*. 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.

```
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
```

Figure 2.13: Search method. Assumes and propagates until a conflict is found, from which a conflict clause is learnt and backtracking performed until search can continue. **Pre-condition:** 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()
```

Figure 2.14: Bumping of variable and clause activities.

2.4.7 Constraint removal

The methods for reducing the set of learnt clauses as well as the top-level simplification procedure can be found in *Figure 2.15*.

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 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.

2.4.8 Top-level solver

The method implementing MiniSat's top-level strategy can be found in *Figure 2.16*. 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 [ES03b, 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.

2.5 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

```
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
```

Figure 2.15: ***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⊥
```

Figure 2.16: 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.

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, ARMS02a]. Our implementation differs in that it has a simpler 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.

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

Chapter 3

New Techniques that Improve MACE-style Finite Model Finding

Koen Claessen, Niklas Sörensson
Chalmers University of Technology and Göteborg University

Abstract

We describe a new method for finding finite models of unsorted first-order logic clause sets. The method is a MACE-style method, i.e. it "flattens" the first-order clauses, and for increasing model sizes, instantiates the resulting clauses into propositional clauses which are consecutively solved by a SAT-solver. We enhance the standard method by using 4 novel techniques: *term definitions*, which reduce the number of variables in flattened clauses, *incremental SAT*, which enables reuse of search information between consecutive model sizes, *static symmetry reduction*, which reduces the number of isomorphic models by adding extra constraints to the SAT problem, and *sort inference*, which allows the symmetry reduction to be applied at a finer grain. All techniques have been implemented in a new model finder, called Paradox, with very promising results.

3.1 Introduction

There exist many methods for finding finite models of First Order Logic (FOL) clause sets. The two most successful styles of methods are usually called *MACE-style* methods, named after McCune's tool MACE [McC94], and the *SEM-style* methods, named after Zhang and Zhang's tool SEM [JZ96].

A MACE-style method transforms the FOL clause set and a domain size into a propositional logic clause set by introducing propositional variables representing the function and predicate tables, and consecutively flattening and instantiating the clauses in the clause set. A propositional logic theorem prover, also called SAT solver, is then used to attack the resulting problem. Apart from MACE itself, Gandalf [Tam97] makes use of a MACE-style method, and there are reports of usages of SATO in a similar way [JZ95].

A SEM-style method performs a search directly on the problem without converting it into a simpler logic. A basic backtracking search backed up by powerful constraint propagation methods, mainly based on exploiting equality, are used to search for interpretations of the function and predicate tables. A principle called *symmetry reduction* is used to avoid searching for isomorphic models multiple times. Apart from SEM itself, the tools FINDER [Sla94], and ICGNS [McC03] are SEM-style methods, and again Gandalf also makes use of SEM-style methods. SEM-style methods are known to perform well on equational problems, and MACE-style methods are supposed be more all-round.

In this paper, we develop a collection of new techniques for finite model generation for unsorted first-order logic. These techniques improve upon the basic idea behind MACE dramatically. We have implemented the techniques in a new model finder called Paradox. The main novel contributions in the paper are the following.

- The main problem in MACE-style methods is the clause instantiation, which is exponential in the number of first-order variables in a clause. A well-known variable reduction technique is *non-ground clause splitting*, for which however only exponential or incomplete algorithms are known. We have devised a new polynomial heuristic for clause splitting. Moreover, we have come up with a completely new variable reduction technique, called *term definitions*.

- The search for a model goes through consecutive stages of increasing domain sizes. In current-day algorithms, there is hardly any coupling between the search for models of different sizes. We make use of an *incremental satisfiability checker* in order to reuse information about the failed search of a model of size s in the search of a model of size $s + 1$.

- SEM-style model finders make use of techniques like the *least number heuristic* [JZ96] and the *extended least number heuristic* [AH01] in order to reduce the symmetries in the search problem. To our knowledge, wTTe are the first to adapt a similar technique in a MACE-style framework. Our contribution here is that when using a SAT-solver, we must apply the symmetry reduction *statically*, i.e. by adding extra constraints, whereas SEM-style methods can apply this *dynamically*, i.e. during the search.

- It is well-known that *sort information* can help the search for models. However, we are working with unsorted problems, and therefore need to

create unsorted models. We have developed a *sort inference* algorithm, which automatically finds appropriate sort information for unsorted problems. This sort information can then be used in several ways to reduce the complexity of the model search problem, while still searching for an unsorted model.

The rest of the paper is organized as follows. In Section 3.2 we introduce some notation. In Section 3.3 we describe the basic ideas behind MACE-style methods. In Sections 3.4, 3.5, 3.6, and 3.7, we introduce the four techniques: *clause splitting and term definitions*, *incremental search*, *static symmetry reduction*, and *sort inference*. In Section 3.8 we discuss some experimental results. Sections 3.9 and 3.10 discuss related work and conclusions.

3.2 Notation

In this section, we introduce some standard notation we use in the rest of the paper. We use the set $\mathcal{N}$ to stand for the set of natural numbers $\{0, 1, 2, \dots\}$. The set $\mathcal{B}$ is the set of booleans $\{\mathsf{false}, \mathsf{true}\}$.

Terms, literal and clauses A term t is built from function symbols $\mathsf{f}, \mathsf{g}, \mathsf{h}$, constants $\mathsf{a}, \mathsf{b}, \mathsf{c}$, and variables $\mathsf{x}, \mathsf{y}, \mathsf{z}$. Each function symbol f has a single arity $\mathsf{ar}(\mathsf{f}) : \mathcal{N}$ which should be respected when building terms. We will merely regard constants as nullary function symbols.

An atom A is a predicate symbol $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \mathsf{S}$ applied to a number of terms. Each predicate symbol P has a single arity $\mathsf{ar}(\mathsf{P}) : \mathcal{N}$. There exists one special predicate symbol $=$, with arity 2, representing equality, whose atoms are written $t_1 = t_2$.

A literal is a positive or negative occurence of an atom. Negative literals are written using $\neg A$, and negative equalities are written $t_1 \neq t_2$.

A clause C is a finite set of literals, intended to be used disjunctively. We write $FV(C)$ for the set of variables in a clause C.

Interpretations An interpretation $\mathcal{I}$ consists of a non-empty set D (the domain), plus for each function symbol f a function $\mathcal{I}(\mathsf{f}) : D^{\mathsf{ar}(\mathsf{f})} \to D$, and for each predicate symbol P a function $\mathcal{I}(\mathsf{P}) : D^{\mathsf{ar}(\mathsf{P})} \to \mathcal{B}$, where we require $\mathcal{I}(=)(d_1, d_2) = \mathsf{true}$ exactly when d_1 is the same domain element as d_2, and false otherwise. An interpretation is called *finite* if its domain D is a finite set.

3.3 MACE-style Model Finding

This section describes shortly what the basic idea behind MACE-style model finding methods is. The description differs slightly from earlier presentations [McC94].

Finite domains The following observation about models is well-known. Given an interpretation $\mathcal{I}$ with a domain D satisfying a clause set S. Given a set D' and a bijection $\pi : D \leftrightarrow D'$, we construct a new interpretation $\mathcal{I}'$ with domain

D' in the following way:

$$\begin{aligned}\mathcal{I}'(\mathsf{P})(x_1,\ldots,x_m) &= \mathcal{I}(\mathsf{P})(\pi^{-1}(x_1),\ldots,\pi^{-1}(x_m))\\ \mathcal{I}'(\mathsf{f})(x_1,\ldots,x_n) &= \pi(\mathcal{I}(\mathsf{f})(\pi^{-1}(x_1),\ldots,\pi^{-1}(x_n))\end{aligned}$$

Now, $\mathcal{I}'$ also satisfies S. We call two models $\mathcal{I}$ and $\mathcal{I}'$ in the above relationship *isomorphic*. The observation implies that in order to find (finite) models, only the size s of the domain matters, and not the actual elements of the domain. Therefore, we arbitrarily choose D to be $\{1, 2, \ldots, s\}$. A special case of the observation, which we will use later, arises when $D' = D$, and π simply corresponds to a permutation of D.

Propositional encoding We are going to encode the model finding problem for a particular set of FOL clauses using propositional variables. For each predicate symbol P, and each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{P})})$ (with each $d_i \in D$), we introduce a propositional variable $\mathsf{P}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{P})})$ representing the case when $\mathcal{I}(\mathsf{P})(d_1, \ldots, d_{\mathsf{ar}(\mathsf{P})})$ is true. Also, for each function symbol f, for each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$, and for each domain element d, we introduce a propositional variable $\mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) = d$ representing the case when $\mathcal{I}(\mathsf{f})(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$ is d. We stress that these propositional variable names are merely syntactic constructs and have no meaning without context.

Flattening In order to create the necessary propositional constraints on the above variables, the first step is to transform the general FOL clauses into clauses only containing *shallow* literals, a process called *flattening*.

Definition 1 (Shallow Literals) *A literal is* shallow *iff it has one the following forms:*

1. $\mathsf{P}(\mathsf{x}_1, \ldots, \mathsf{x}_m)$, *or* $\neg\mathsf{P}(\mathsf{x}_1, \ldots, \mathsf{x}_m)$,
2. $\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n) = \mathsf{y}$, *or* $\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n) \neq \mathsf{y}$,
3. $\mathsf{x} = \mathsf{y}$.

There are two cases when a given literal is not shallow: (1) it contains at least one subterm t which is not a variable, but should be; (2) the literal is of the form $\mathsf{x} \neq \mathsf{y}$. In case (1), we can lift out an offending term t out of any literal occurring in a clause C by applying the following sequence of rewrite steps:

$$\begin{aligned} C[t] &\longrightarrow \mathsf{let}\ \mathsf{x} = t\ \mathsf{in}\ C[\mathsf{x}] && (\mathsf{x}\ \text{not in}\ C)\\ &\longrightarrow \forall \mathsf{x}.\,[\mathsf{x} = t\ \Rightarrow\ C[\mathsf{x}]]\\ &\longrightarrow t \neq \mathsf{x} \vee C[\mathsf{x}]\end{aligned}$$

In the second step in the above we make use of a standard representation of let-definitions in terms of universal quantification and implication. If t occurs more than once in C, we introduce only one variable x for t, and replace all occurences of t by x. In case (2), we apply the following rewrite rule:

$$C[\mathsf{x}, \mathsf{y}] \vee \mathsf{x} \neq \mathsf{y} \quad \longrightarrow \quad C[\mathsf{x}, \mathsf{x}]$$

If we apply the above transformations repeatedly, in the end all literals will be be shallow literals.

Example 1 *Take the unit clause* $\{ \mathsf{P}(\mathsf{a}, \mathsf{f}(\mathsf{x})) \}$. *After flattening, the clause looks as follows:*

$$\{ \mathsf{a} \neq \mathsf{y},\ \mathsf{f}(\mathsf{x}) \neq \mathsf{z},\ \mathsf{P}(\mathsf{y}, \mathsf{z}) \}.$$

Instantiating The final step is to generate propositional clauses from the flattened clauses. We generate three sets of propositional clauses:

- **Instances** For each flattened clause C, and for each substitution $\sigma : FV(C) \to D$, we generate the propositional clause $C\sigma$. (Recall that shallow literals instantiated with domain elements function as propositional literals.) In the result of the substitution, we immediately simplify all literals of the form $d_1 = d_2$ (whose value is known at instantiation time), leading to either removal of the whole clause in question (when d_1 and d_2 are equal), or simply removal of the equality literal (when d_1 and d_2 are not equal).

- **Functional definitions** For each function symbol f, for each $d, d' \in D, d \neq d'$, and for each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$, we introduce the propositional clause

  $$\{ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) \neq d,\ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) \neq d' \}$$

  representing the fact that a function can not return two different values for the same arguments.

- **Totality definitions** For each function symbol f, and for each argument vector $(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})})$, we introduce the propositional clause

  $$\{ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) = 1,\ \ldots,\ \mathsf{f}(d_1, \ldots, d_{\mathsf{ar}(\mathsf{f})}) = s \}$$

  representing the fact that a function must return at least one value for each argument.

If we can find a propositional model that satisfies all of the above clauses, we have found a finite model satisfying the original set of FOL clauses. We use a SAT solver to find the actual propositional model. The FOL model can be easily built by using the propositional encoding described earlier in this section.

3.4 Reducing Variables in Clauses

The number of instances of each clause that is needed, is exponential in the number variables the clause contains. In general, if a clause contains k variables, the number of instances that will be needed for domain size s are s^k. Moreover, this property is made worse by the fact that term flattening introduces many auxiliary variables (something that a SEM-style model finder does not do). In this section we describe how to remedy this situation.

Splitting Splitting is a well known method [Sch01, Tam97, RV00] that can be used to replace one clause by several other clauses *each containing fewer variables* than the original clause. The following is an example of *non-ground splitting.*

Example 2 *By introducing the completely fresh* split predicate $\mathsf{S}(\mathsf{x})$*, the clause* $\{ \mathsf{P}(\mathsf{x},\mathsf{y}),\ \mathsf{Q}(\mathsf{x},\mathsf{z}) \}$ *can be split into the follwing clauses. Note that we by doing this have reduced the maximum number of variables per clause from three to two.*

$$\{ \mathsf{P}(\mathsf{x},\mathsf{y}),\ \mathsf{S}(\mathsf{x}) \}$$
$$\{ \neg\mathsf{S}(\mathsf{x}),\ \mathsf{Q}(\mathsf{x},\mathsf{z}) \}$$

The general criterion for splitting that we use look like this.

Definition 2 (Binary Split) *Given a clause* $C[\vec{x}] \cup D[\vec{y}]$ *where* C, D *are subclauses and* $\vec{x}, \vec{y}$ *are the sets of variables occuring in them. Then* C *and* D *constitute a* proper binary split *of the given clause iff there exist at least one variable* x *in* $\vec{x}$ *such that* $\mathsf{x} \notin \vec{y}$*, and at least one variable* y *in* $\vec{y}$ *such that* $\mathsf{y} \notin \vec{x}$*. The resulting two clauses after the split are:*

$$\{ \mathsf{S}(\vec{x} \cap \vec{y}) \} \cup C[\vec{x}]$$
$$\{ \neg\mathsf{S}(\vec{x} \cap \vec{y}) \} \cup D[\vec{y}]$$

Here, S *must be a fresh predicate symbol not occurring anywhere in the problem.*

The resulting two clauses contain less variables per clause since x is guaranteed not to appear in the second clause, and y not in the first clause. A special case is when $\vec{x} \cap \vec{y}$ is empty, in which case S becomes nullary predicate, i.e. a logical constant.

Repeated Binary Splitting In general the resulting clauses of a binary split can possibly be splitted further, thus to get best possible result binary splits can be repeated on both resulting clauses for as long as it is possible. However, there might be several possible ways to apply a binary split, and a greedy choice could destroy the possibilities for further splitting of the resulting clauses. It might be worthwhile to come up with an optimal (in terms of number of variables) algorithm for repeated binary splitting, but so far we have reached good results using a cheap heuristic.

Existing heuristics for binary splits Gandalf [Tam97] and Eground [Sch01] both incorporate the same heuristic for finding binary splits, which works as follows. Given a clause C, all proper subsets of variables occurring in C are enumerated (small subsets first). For each subset V, it is checked if the clause can be split into two clauses, such that the intersection of variables occurring in both clauses is equal to V, which takes linear time in the length of the clause. If such a subset V is found, the clause is split accordingly. Since there is an exponential amount of such subsets, there is an upper limit (an arbitrary constant) on the amount of subsets that is tried. Beyond the limit, the algorithm gives up. The problem here is that for clauses containing many variables, we have to be lucky and find the right subset before we pass the upper limit.

Our heuristic In contrast, the heuristic we use is polynomial and it will always find a split if there is one, though it might not always turn out to be the best split. First, given a clause C, we say that two variables are *connected* in C if there is some literal in C in which they both occur. Note that if all variables are connected to each other, then a proper split is impossible, but otherwise it is. The heuristic now finds the variable x which is connected to the least amount of other variables in C. As soon as we find x, we take all literals containing x on one side of the split, and all other literals on the other side. One advantage of this method is that we know that the side of the split containing x cannot be split any further, so we only have to continue splitting the other side. Our heuristic seems to work well in practice, and works even for clauses with many variables.

Term definitions In cases where literals contain deep ground terms we can avoid introducing auxiliary variables, by introducing fresh constants as names for the terms, and substituting the terms for their names.

Example 3 *Flattening the clause* $\{\ \mathsf{P}(\mathsf{f}(\mathsf{a},\mathsf{b}),\mathsf{f}(\mathsf{b},\mathsf{a}))\ \}$ *yields the clause:*

$$\{\ \mathsf{a} \neq \mathsf{x},\ \mathsf{b} \neq \mathsf{y},\ \mathsf{f}(\mathsf{x},\mathsf{y}) \neq \mathsf{z},\ \mathsf{f}(\mathsf{y},\mathsf{x}) \neq \mathsf{w},\ \mathsf{P}(\mathsf{z},\mathsf{w})\ \}$$

This clause, which cannot be splitted (all variables are connected to each other), contains 4 variables. However, if we first transform the original clause by introducing fresh names for its ground terms, we obtain the following satisfiability-equivalent set of clauses:

$$\{\ \mathsf{t}_1 = \mathsf{f}(\mathsf{a},\mathsf{b})\ \}$$
$$\{\ \mathsf{t}_2 = \mathsf{f}(\mathsf{b},\mathsf{a})\ \}$$
$$\{\ \mathsf{P}(\mathsf{t}_1,\mathsf{t}_2)\ \}$$

If we now flatten these, we get the following three clauses:

$$\{\ \mathsf{a} \neq \mathsf{x}, \mathsf{b} \neq \mathsf{y}, \mathsf{f}(\mathsf{x},\mathsf{y}) \neq \mathsf{z}, \mathsf{t}_1 = \mathsf{z}\ \}$$
$$\{\ \mathsf{a} \neq \mathsf{x}, \mathsf{b} \neq \mathsf{y}, \mathsf{f}(\mathsf{y},\mathsf{x}) \neq \mathsf{z}, \mathsf{t}_2 = \mathsf{z}\ \}$$
$$\{\ \mathsf{t}_1 \neq \mathsf{x}, \mathsf{t}_2 \neq \mathsf{y}, \mathsf{P}(\mathsf{x},\mathsf{y})\ \}$$

These clauses each contain 3 variables; a significant improvement.

In the general case, a clause $C[t]$ is translated into the clauses $\{\ \mathsf{a} = t\ \}$ and $C[\mathsf{a}]$, where t is a non-constant ground term and a is a fresh constant, not occurring anywhere else in the problem. In the clause $C[\mathsf{a}]$ only one variable needs to be introduced for the constant a, in constrast to one for all subterms of t. Note also that if the term t occurs in several different clauses in the problem, then there only has to be one definition $\{\ \mathsf{a} = t\ \}$, and the fresh constant a can be reused by all clauses containing t.

3.5 Incremental Search

The most popular basic algorithm for SAT solving, the DPLL procedure [DLL62], is a backtracking procedure based on unit propagation. Modern versions of the

algorithm usually also include several improvements, such as heuristics for variable selection, backjumping, and conflict learning.

In our context, *conflict learning* is of particular interest. It allows the procedure to learn from earlier mistakes. Concretely, for each contradiction that occurs during the search, the reason for the conflict is analysed, resulting in a *learned clause* that may avoid similar situations in future parts of the search. In this way, a set of conflict clauses is gathered during the search, representing information about the search problem. A conflict clause is always logically implied by the original problem, and thus holds without any assumption.

As part of our tool Paradox, we have implemented a Chaff-style [MMZ+01b] version of the DPLL algorithm[1], extended with the possibility to *incrementally* solve a sequence of problems. The idea is that we want to benefit from the similarity of the sequence of SAT instances, generated by our propositional encoding for each domain size. This is done by keeping the learned clauses generated by the search for one instance, also for the next.

Here is a formalization of the kind of sequences of SAT problems that our incremental SAT solver can deal with.

Definition 3 *Given a sequence of sets of propositional clauses δ_i, and a sequence of sets of propositional unit clauses α_i. Then the sequence φ_i, defined as follows, is an* incremental satisfiability problem.

$$\begin{aligned}
\varphi_1 &= \alpha_1 \cup \delta_1 \\
\varphi_2 &= \alpha_2 \cup \delta_1 \cup \delta_2 \\
\varphi_3 &= \alpha_3 \cup \delta_1 \cup \delta_2 \cup \delta_3 \\
&\dots
\end{aligned}$$

That is, to move from one instance to the next, we have to keep all the general clauses δ_i, but can retract and replace the unit clauses α_i. The incremental SAT algorithm we use represents the α_i as assumptions, and not as constraints. Therefore, we can keep *all* learned clauses from one instance and reuse them in the next instance. This is because every learned clause generated by the conflict analysis algorithm is implied by the subset $(\bigcup_{j=1}^{i} \delta_j)$ of the problem instance.

Model Generation as Incremental Satisfiability In Section 3.3 we described how to encode the problem of finding a model of a specific size into propositional logic. It is easy to see that the encodings for different sizes have much in common, but in order to specify it as an incremental satisfiability problem we need to be precise about what the difference is.

Given the SAT instance for a specific size s we want to create the instance for the size $s+1$. Then for the *instances* and *function definitions*, we can keep all previous clauses, and we only have to add the new clauses that mention the new domain element $s+1$. For the *totality definitions* however, we need to take away the clauses and replace them with less restrictive clauses.

In order to fit this in the incremental framework mentioned above, we introduce a special propositional variable $\mathsf{d_s}$ for each domain size s. This variable should be interpreted as "the current domain size is s". Instead of adding a totality clause as it is, we add a *conditional* variant of it, by adding $\neg\mathsf{d_s}$ as a

[1]The SAT solver, called Satnik, is also a stand alone tool in itself

literal to each totality clause. When solving the problem for domains of size s, we simply take $\mathsf{d_s}$ being true as an assumption unit clause α_s. This immediately implies the unconditional versions of the totality clauses. Then, if we find a model, we are done. Otherwise, $\mathsf{d_s}$ was apparently a too strong assumption, and we thus retract the assumption α_s, and add $\neg\mathsf{d_s}$ as a top-level unit clause. By doing this, we have effectively "deactivated" the totality clauses for size s by satisfying them, and are ready to add clauses for the next domain size $s+1$.

Example 4 *Assume that we have two constants* a *and* b *and a current domain size of 2. Then the conditional totality definitions look like this:*

$$\{\ \mathsf{a}=1,\ \mathsf{a}=2,\ \neg\mathsf{d}_2\ \}$$
$$\{\ \mathsf{b}=1,\ \mathsf{b}=2,\ \neg\mathsf{d}_2\ \}$$

The problem for size 2 is now solved by assuming d_2*. We can get to the problem for size 3 by adding the following clauses:*

$$\{\ \neg\mathsf{d}_2\ \}$$
$$\{\ \mathsf{a}=1,\ \mathsf{a}=2,\ \mathsf{a}=3,\ \neg\mathsf{d}_3\ \}$$
$$\{\ \mathsf{b}=1,\ \mathsf{b}=2,\ \mathsf{b}=3,\ \neg\mathsf{d}_3\ \}$$

Assuming d_3 *now gives the right totality clauses for size 3.*

Effectiveness In our preliminary experiments, the method of incrementally solving the model generation problem for increasing domain sizes decreases the overall time spent in the SAT solver in many cases by at least a factor of 2. The implementation of the SAT solver removes clauses that are trivially satisfied because of the presence of other unit clauses. This mechanism takes care of removing the reduntant totality clauses of previous sizes.

There are also some questions left to investigate, particularly which of the learned clauses should be kept between instances. In general it slows the SAT solver down too much to store all of them, apart from the fact that it is also too space consuming. Currently we simply use our SAT solver's basic heuristic for learned clause removal, which is designed to work well for solving single problems. It is likely that one could design other heuristics that would take into account the fact that a clause that seems to be uninteresting in the current part of the current search problem, in fact could be useful in the next problem instance.

3.6 Static Symmetry Reduction

The way we have expressed the model finding problem in SAT implies that for each model, all of its isomorphic valuations (i.e. the valuations we get by permuting the domain elements) are also models. This makes the SAT problem unneccessarily difficult. SEM-style methods use *symmetry reduction techniques* such as the *least number heuristic* (LNH) [JZ96] and the *extended least number heuristic* (XLNH) [AH01] in order to restrict the search space to particular permutations of models. This is done while the search for a model is going on, i.e. *dynamically*. In order not to have to change the inner workings of

the SAT-solver, we adapt some of the ideas behind these symmetry reduction techniques, but implement them by adding extra constraints, which remove symmetries *statically*.

Constant symmetries Let us start with the simple case, and only look at the values of the constants occurring in the problem. If we order all constants occurring in the problem in some arbitrary way, we get a sequence $\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_k$. Suppose that we are searching for a model of size s. We now require that the model we are looking for has a certain *canonical form*. This canonical form corresponds to a certain permutation of the domain elements, namely such that $\mathsf{a}_1 = 1$, and for all $i > 1$, $\mathsf{a}_i = d$ only when there is a $j < i$ such that $\mathsf{a}_j = d - 1$. So, when picking a domain element for a_i, we can either pick an element that we have already seen, or a new element, which must be the *least* element in D which we have not used yet. It is easy to see that every interpretation has an isomorphic permutation where this is the case.

Adding this extra restriction implies that $\mathcal{I}(a_i) \leq i$, which immediately gives rise to the following extra clauses:

$$\begin{aligned}
&\{\ \mathsf{a}_1 = 1\ \}\\
&\{\ \mathsf{a}_2 = 1,\ \mathsf{a}_2 = 2\ \}\\
&\{\ \mathsf{a}_3 = 1,\ \mathsf{a}_3 = 2,\ \mathsf{a}_3 = 3\ \}\\
&\ldots
\end{aligned}$$

These clauses actually subsume the totality clauses for their constants. Also, for any $1 < i \in D$, and for $1 < d \leq i \in D$, we add the following clause, which directly formulates the canonicity requirement:

$$\{\ \mathsf{a}_i \neq d,\ \mathsf{a}_1 = d - 1,\ \mathsf{a}_2 = d - 1,\ \mathsf{a}_3 = d - 1,\ \ldots,\ \mathsf{a}_{i-1} = d - 1\ \}$$

That is, a_i can only get the value d if some previous constant already has used the value $d - 1$.

This process can be adapted for an incremental search in the following way: for each new model size, we add only those symmetry-reducing clauses that contain the new domain element, and none of the greater elements. We never take away any of the generated symmetry reduction clauses in the incremental search.

Function symmetries If the problem only contains function symbols of arity 0, then the above clauses are enough to remove *all* symmetries. However, when there are function symbols of higher arity, this is no longer easy to do statically. We can however remove some of the symmetries by, in addition to the above clauses, adding the following clauses for a function symbol f of arity 1, when we are looking for models of sizes s greater than k (the number of constants). We require that $k > 0$ (we just introduce an arbitrary constant when $k = 0$).

$$\begin{aligned}
&\{\ \mathsf{f}(1) = 1,\ \mathsf{f}(1) = 2,\ \ldots,\ \mathsf{f}(1) = k + 1\ \}\\
&\{\ \mathsf{f}(2) = 1,\ \mathsf{f}(2) = 2,\ \ldots,\ \mathsf{f}(2) = k + 1,\ \mathsf{f}(2) = k + 2\ \}\\
&\{\ \mathsf{f}(3) = 1,\ \mathsf{f}(3) = 2,\ \ldots,\ \mathsf{f}(3) = k + 1,\ \mathsf{f}(3) = k + 2,\ \mathsf{f}(3) = k + 3\ \}\\
&\ldots
\end{aligned}$$

The rationale here is again that, in order to pick a value for a particular $\mathsf{f}(d)$, we can simply use an element that we have already seen, or the least element that has not been used yet.

Note that we add only *one* such clause for each size increase beyond k. We have not investigated how to decide which function symbol to pick. In our current implementation, we simply pick an arbitrary function symbol. In principle it is possible to use a different function symbol in every clause. We can also generalize the above for function symbols g of arity larger than 1, by defining a fresh function f in terms of g by e.g. $\mathsf{f}(\mathsf{x}) = \mathsf{g}(\mathsf{x}, \mathsf{x})$.

The resulting SAT problem, even though it is a little bit bigger than without the symmetry reducing clauses, is often dramatically easier to solve, both in cases where there is a model, and in cases where there is no model.

3.7 Sort Inference

When formalizing a problem in terms of unsorted FOL, there often exist different *concepts* in the problem, which when finding a model, all have to be interpreted using the same domain D. This can be quite unnatural, both when trying to understand a model and when trying to find a model. Examples of these kinds of concepts are points, lines, and planes in geometry problems, and booleans and numbers in system descriptions. A 'typed' version of FOL, Multi-Sorted First Order Logic (MSFOL), requires these concepts, the *sorts*, to be explicit in the formulation of the problem. It is well-known that sort information helps searching for models. In this section, we describe how to use the sort information, and, more interestingly, how to *infer* the sort information such that it can be used for originally unsorted problems as well.

Sorted models In the MSFOL world, apart from predicate symbols and function symbols, there exists *sort* symbols $\mathsf{A}, \mathsf{B}, \mathsf{C}$. Each function symbol f has an associated sort $\mathsf{sort}(\mathsf{f})$ of the form $\mathsf{A}_1 \times \cdots \times \mathsf{A}_{\mathsf{ar}(\mathsf{f})} \to \mathsf{A}$, and each predicate symbol P (except for $=$, which works on all sorts) has an associated sort $\mathsf{sort}(\mathsf{P})$ of the form $\mathsf{A}_1 \times \cdots \times \mathsf{A}_{\mathsf{ar}(\mathsf{P})}$. Moreover, in each clause, each variable x has an associated sort $\mathsf{sort}(\mathsf{x})$ of the form A. These sorts have to be respected in order to build only well-sorted terms and literals.

An MSFOL interpretation $\mathcal{M}$ consists of a domain D_A for each sort A, plus for each function symbol f a function $\mathcal{I}(\mathsf{f}) : D_{\mathsf{A}_1} \times \cdots \times D_{\mathsf{A}_{\mathsf{ar}(\mathsf{f})}} \to D_\mathsf{A}$ matching the sort of f, and for each predicate symbol P a function $\mathcal{I}(\mathsf{P}) : D_{\mathsf{A}_1} \times \cdots \times D_{\mathsf{A}_{\mathsf{ar}(\mathsf{P})}} \to \mathcal{B}$, matching the sort of P.

We define the notion of satisfiability for MSFOL interpretations in the obvious way: quantification of variables in clauses becomes sort-dependent.

Unsorted vs. sorted Now, since our objective is to find an unsorted model, in order to make use of sorts, we must link unsorted models and sorted models in some way. It is not automatically the case that if we find a sorted model, there is also an unsorted model of the same problem. However, it is the case that we can turn any unsorted model of a problem into a sorted model of the same problem.

The basic observation is that any unsorted interpretation $\mathcal{I}$ with a domain D can be turned into a sorted interpretation $\mathcal{M}_\mathcal{I}$ by taking $D_\mathsf{A} = D$ for each

sort A, and by simply reusing all function and predicate tables from $\mathcal{I}$. Now, for a suitably well-sorted set of clauses S, we have that $\mathcal{I}$ satisfies S iff $\mathcal{M}_\mathcal{I}$ satisfies S. So, the key idea is that when searching for an unsorted model $\mathcal{I}$, we can just as well search for the sorted model $\mathcal{M}_\mathcal{I}$, i.e. search for a sorted model where all sorts have the same domain size.

The advantage of searching for a sorted model becomes clear in the following, which is a more fine-grained version of the symmetry observation for unsorted interpretations. Given an MSFOL interpretation $\mathcal{M}$ containing a domain D_A for a sort A satisfying an MSFOL clause set S. Given a set D' and a bijection $\pi : D_\mathsf{A} \leftrightarrow D'$, we construct a new interpretation $\mathcal{M}'$ by replacing D_A by D' and applying π in the obvious way. Now, $\mathcal{M}'$ also satisfies S.

Sorted symmetry reduction We can now make the following refinement of our earlier symmetry reduction method. Given a valid sort-assignment to each function and predicate symbol, we can simply search for an MSFOL model where the domains for each sort are the same, but we can apply symmetry reduction for each sort separately. That is, for each sort, we create a sequence of the constants of that sort, and we add the extra clauses mentioned in Section 3.6.

Example 5 *Given a problem with three constants* $\mathsf{a}, \mathsf{b}, \mathsf{c}$ *and two sorts* A, B*, where* $\mathsf{sort}(\mathsf{a}) = \mathsf{sort}(\mathsf{b}) = \mathsf{A}$ *and* $\mathsf{sort}(\mathsf{c}) = \mathsf{B}$*, we get the following symmetry reduction clauses:*

$$\{\ \mathsf{a} = 1\ \}$$
$$\{\ \mathsf{b} = 1,\ \mathsf{b} = 2\ \}$$
$$\{\ \mathsf{c} = 1\ \}$$

Sort inference The big question is then: How do we get such a suitable sort-assignment for a flattened unsorted clause set? The algorithm we use is simple. In the beginning, we assume that all function symbols and predicate symbols have completely unrelated sorts. Then, for each variable in each clause, we force the sorts of the occurences of that variable to be the same. Also, we force the sorts on both sides of the $=$ symbol to be the same. This can be implemented by a straight-forward union-find algorithm, so that the whole agorithm runs in linear time. In the end, the hope is to be left with more than one sort.

We have found that this simple sort inference algorithm really finds multiple sorts in about 30% of the (unsorted) problems occurring in the current TPTP benchmark suite [SS07] over all, and in about 50% of the satisfiable problems particularly.

Sort size reduction There is another way in which we can make use of the inferred sort information in model finding. Under certain conditions, we can restrict the size of the domains for particular sorts, which reduces the complexity of the instantiation procedure.

Suppose that we have a sort A, and k constants $\mathsf{a}_1, \ldots, \mathsf{a}_k$ of sort A, but no function symbols of arity greater than 0 which have A as their result sort. Moreover, assume that we are not using the $=$ symbol positively on terms of sort A anywhere in the problem S, then the following holds. There exists an MSFOL

model of S with a domain D_{A} of size k iff there exists an MSFOL model of S with a domain D_{A} of size greater than k. In other words, to find an MSFOL model of S, we do not have to instantiate variables of sort A with more than k domain elements. This can considerably reduce instantiation time for problems where sorts are inferred.

The proof looks as follows. ($\Leftarrow$) If we have a model where D_{A} has more than k elements, there must be an element d which is not the value of any constant, so we can safely take it away from the domain, and all function tables and predicate tables will still be still well-defined. It is also still a model, since making a domain smaller only increases the number of clause sets that are satisfied. ($\Rightarrow$) If we have a model of S, then we can always add a new element d' to D_{A} by picking an existing element $d \in D_{\mathsf{A}}$ and making all functions and predicates produce the same results for d' as they do for d. The resulting interpretation is still a model, because every literal evaluates to the same value for d' as it does for d. (However, this is only true for non-equality literals.) For negative equality, making the domain bigger can only increase the number of clause sets that are satisfied. This is not true for the use positive equality literals, which is the reason why it is disallowed in the assumption.

(It is however possible to weaken the restriction on the use of positive equality. In order to be able to restrict the size of the domain of A, it is enough to require that we do not use positive literals of the form $\mathsf{x} = \mathsf{y}$, where x and y are variables of sort A. So, it is okay to use positive literals of the form $t = t'$ and $t = \mathsf{x}$, where t and t' are not variables. In the latter case however, we generally need to consider $k + 1$ elements instead of k.)

EPR problems A special case of sort size reduction is the case where the problem is an *Effectively Propositional* (EPR) problem, i.e. no functions of arity greater than 0 occur in the problem at all. In this case, each sort only contains constants, and the number of constants k in the largest sort is an upper bound on the size of models we need to consider. (This is independent of the use of equality in the problem, since we only need the ($\Leftarrow$) part of the above proof.) When no model of size up to k is found, we know there can be no model of greater size, and therefore the problem must be contradictory. Thus we have a complete method for EPR problems.

3.8 Experimental Results

We have implemented all the techniques in a new finite model finder called Paradox. Here is a list of promising concrete results we have obtained so far with our model finder:

- On the current TPTP version 2.5.0 [SS07], and with a time limit of 2 minutes for each problem, we can solve 90% of the satisfiable problems. This is significantly better than last year's CASC winner in the satisfiability category on the same problems with a time limit of 5 minutes.
- Within a time limit of 10 minutes, we have solved 28 problems from the current TPTP which currently have a rating 1.0 (i.e. Paradox is the first to solve those problems), including 15 "open" or "unknown" problems that were solved within seconds.

- With an older version of Paradox (using different term definitions heuristics and SAT solver parameters), in the search for counter models for the combinatory logic question if the fragment {B,M} posseses the fixed point property, we have shown that there are no counter models of sizes smaller than or equal to 10, which took us 9 hours. The previously known bound was 7.

Our preliminary findings are that the techniques described in this paper strictly improve on all known MACE-based methods. Also, they perform almost always better than SEM-based methods on problems that contain more than just unit equalities. SEM-based methods however are superior on most problems that contain lots of unit equalities, such as group theory problems. Interestingly, there are some exceptions, such as combinatory logic problems, where Paradox seems to behave well.

3.9 Related Work

There are several tools that solve similar problems as we do, or use similar techniques to the ones we use.

Eground [Sch01] is a tool that takes an EPR problem, i.e. a problem that does not contain any function symbols of arity greater than 0, and generates a SAT problem that is satisfiable iff the original problem is. Interestingly, Eground has many of the same problems as a MACE-style model finder. Eground was the first tool to perform non-ground splitting in order to reduce the number of variables in clauses. Also, Eground performs an analysis that computes sets of constants for each variable in a clause for which the variable should be instantiated. The hope is that these sets are smaller than the set of all constants in the problem. The analysis is somewhat similar to sort inference, with three main differences. Our analysis also works for non-EPR problems, and also works for problems containing equality. Eground's analysis makes use of the sign of predicate symbols, which makes it more precise in some cases.

Comparing Eground as an EPR solver with our proposed method of solving EPR problems mentioned at the end of Section 3.7, it seems that they are complementary. Assuming that no equality is used in the problem, and that the analyses work equally well, and ignoring the symmetry reduction, we can make the following rough observations. Given an EPR problem that is contradictory, Eground's method is probably going to win over ours since it immediately tries the "biggest" case, whereas our method will go through all smaller model sizes first. Given an EPR problem that is satsifiable, it is very likely that it is not needed to go all the way up to the biggest case, and that a smaller model can be found much quicker.

MACE [McC94] is McCune's first finite model finder. The basic idea behind MACE is described in Section 3.3. Since it does not perform any variable reduction techniques, there are many problems where MACE cannot deal with largish domain sizes. It has its own built-in SAT solver which is currently not up-to-date with the current state-of-the-art SAT technology.

SEM [JZ96], FINDER [Sla94], and ICGNS [McC03] are all SEM-based tools. SEM and FINDER are specifically designed for sorted problems, whereas ICGNS only works for unsorted problems. Comparing the symmetry reduction in SEM-

based tools with ours, we can say that their symmetry reduction works dynamically, i.e. during the search they will always pick the smallest not-used domain element when a new element is needed. We apply the symmetry reduction statically, which removes the same symmetries when picking values for constants, but for functions, our extra function symmetry clauses do not remove as many symmetries. Still, having a state-of-the-art SAT solver as the underlying search engine seems to be superior in many cases once one is able to instantiate the problem for the desired domain size. It will continually be useful to investigate these complementary method's strengths and weaknesses in order to understand the problem area better.

Gandalf [Tam97] is a general theorem prover that also implements model finding. Gandalf contains lots of different complementary algorithms for particular problem domains, which are, upon receiving a problem, scheduled, together occupying all available time. For satisfiability, Gandalf provides saturation techniques (that help finding cases with infinite models), SEM-style techniques, and one MACE-style method. As far as we know, Gandalf was the first to use splitting techniques in MACE-style model finding. Gandalf was the winner of last year's CASC satisfiability division.

A more general approach to incremental SAT solving than what we use here is used in the tool Satire [WKS01]. In Satire, one can take away and add clauses arbitrarily. This requires extra bookkeeping to implement. Our method requires one to decide on beforehand which clauses are going to be retracted. Fortunately, Satire's extra generality is not needed in our application, and our simple, more efficient (but more restrictive) approach is enough.

3.10 Conclusions and Future Work

We have shown that MACE-style methods can be improved upon by incorporating symmetry reduction methods (inspired by well-known related work in the SEM world), by adding them as static constraints to the generated SAT problem. The automatic inference of sorts in order to refine the symmetry reduction turned out to be a very powerful tool in this context. We have also shown that it is good to intimately integrate a SAT solver with the algorithm that uses it, in order to get the most benefit out of it.

However, our work on reducing the number of variables in clauses by using splitting methods and term definitions is, though very promising, only showing the tip of the ice berg of what is left to do in the area. Our splitting heuristic seems to perform well in practice, but it is unsatisfactory that it is based on repeated binary splits. We have not been able to formulate a "most general" clause *hyper*-splitting condition, which all correct splitting algorithms must obey; all previous attempts have been too restrictive. This has made it impossible for us to explore the design space of splitting algorithms in a satisfactory way. A similar situation holds for the term definitions, where it is unclear exactly when term definitions should be introduced. Ultimately we would like to integrate splitting and term definitions in order to get the best of both worlds.

Other future work includes improving the sort inference algorithm which is very simple at the moment. The problem can be thought of as a *flow-analysis* problem from the field of program analysis, and much inspiration can be found there. Also, we would generalize the sort inference to already sorted problems,

in order to find more fine-grained sort assignments than the sort-assignment declared in the problem.

Another direction of research is to adapt clausification algorithms in order to perform clausification, flattening, and splitting at the same time. The basic decision a clausification algorithm must make is when to introduce a new name for a subformula. This decision is guided by optimizing certain parameters of the resulting problem, usually the number of resulting clauses or literals. We could adapt such an algorithm to minimize number of variables per clause instead.

Some problems are inherently complex because they for example contain predicate or function symbols with a huge arity. In order to even represent (let alone search for) models of reasonable domain size would require too much memory. One idea we have started to explore is to *strengthen* the original theory by replacing the offending symbols by nested expressions containing function symbols of much lower arity. If we find a model of the strengthened problem, which might not exist anymore but is hopefully easier to do, that model can be translated back into a model of the original problem. (Of course, introducing these huge predicates is ultimately a modelling question; something that the modeller of the original problem should think about.)

Lastly, we have looked at non-standard applications of our model finding techniques in the fields of planning (where a found model represents a plan), finite state system verification (where a found model represents a proof of the correctness of the system), and general FOL theorem proving (where we use finite models to approximate possible infinite models, and the absence of such a finite approximation model beyond a certain precision represents the absence of a model alltogether).

Chapter 4

Temporal Induction by Incremental SAT Solving

Niklas Eén, Niklas Sörensson
Chalmers University of Technology and Göteborg University

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.

4.1 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 [MMZ+01b]. 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.

4.2 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 $\mathbf{T}(\vec{s}, \vec{s}')$, the set of initial states by a formula $\mathbf{I}(\vec{s})$, and further denote the safety property by $\mathbf{P}(\vec{s})$. We will use $\vec{s_n}$ 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 $\mathbf{I}(\vec{s_n})$, $\mathbf{P}(\vec{s_n})$, and $\mathbf{T}(\vec{s_n}, \vec{s_{n+1}})$.

4.2.1 The SAT problem

Let *Bool* denote the *boolean* domain $\{0, 1\}$, and $Vars := \{x_0, x_1, x_2, \ldots\}$ 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 $Vars \rightarrow Bool$. 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 $\{ \{\overline{p}, x\}, \{\overline{p}, y\}, \{p, \overline{x}, \overline{y}\} \}$.

4.2.2 Temporal Induction

This section briefly summarizes the verification technique *temporal induction* presented in [SSS00].[1] The word "temporal" suggests that the induction is carried 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 4.4. Let us formalize the strengthened induction by defining the following formulas:

[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.

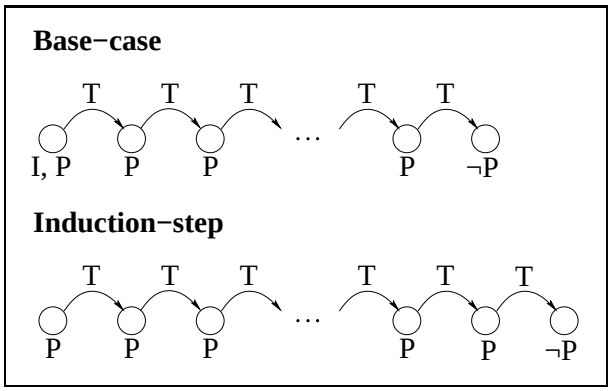


Figure 4.1: If the n-th *base-case* is unsatisfiable, it should be read as "There exists no n-step path to a state violating the property, assuming the property holds the first $n-1$ steps." If the n-th *induction-step* is unsatisfiable, it should be read as "Following an n-step trace where the property holds, there exists no next state where it fails".

$$\begin{aligned}
\mathbf{Base}_n &:= \mathbf{I}_0 \ \wedge \ \Big((\mathbf{P}_0 \wedge \mathbf{T}_0) \ \wedge \ldots \wedge \ (\mathbf{P}_{n-1} \wedge \mathbf{T}_{n-1})\Big) \ \wedge \ \overline{\mathbf{P}_n} \\
\mathbf{Step}_n &:= \Big((\mathbf{P}_0 \wedge \mathbf{T}_0) \ \wedge \ldots \wedge \ (\mathbf{P}_n \wedge \mathbf{T}_n)\Big) \ \wedge \ \overline{\mathbf{P}_{n+1}} \\
\mathbf{Unique}_n &:= \bigwedge_{i \le j \le n} (\vec{s_i} \neq \vec{s_{j+1}}) \quad = \bigwedge_{i \le j \le n} \bigvee_k \neg(s_{i,k} \leftrightarrow s_{j,k})
\end{aligned}$$

An interpretation of these formulas is depicted in *Fig. 4.1*. 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.

4.3 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 [SS96], 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 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 [Hoo93], 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		

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

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 the approach is simpler is that *all* learnt 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.

Furthermore, general clause deletion 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.

4.4 Incremental Induction

In section 4.2.2 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 4.3.

Algorithm 2 "Extending base".

addClauses($[\mathbf{I}_0]$)
for n ∈ 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 ∈ -1..−∞ **do**
  solve({})
  if (UNSATISFIABLE)
    return IND. STEP HOLDS
  addClauses($[\mathbf{T}_n]$)
  addClauses($[\mathbf{P}_n]$)
  for i ∈ 0..n+1 **do**
    addClauses($[\vec{s_i} \neq \vec{s_n}]$)

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 4.2.2, 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.

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.

4.4.1 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:

$$
\begin{array}{ll}
\textit{length 0:} & \mathbf{I}\mathbf{B} \\
\textit{length 1:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
\textit{length 2:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
\textit{length 3:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
 & \qquad\qquad \vdots \\
\textit{length n:} & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B}
\end{array}
$$

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.:

$$
\begin{array}{lll}
 & (1) & \overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\mathbf{B} \\
\textit{or} & (2) & \mathbf{I}\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \ldots \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}\,\overline{\mathbf{B}} \overset{\mathbf{T}}{\frown} \overline{\mathbf{I}}
\end{array}
$$

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 contain-

ing any $\overline{\mathbf{I}}$:s but including the uniqueness constraints dictated by condition 3.[3] 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) 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.

4.4.2 Improving the Unique States Requirement

The uniqueness constraints described in section 4.2.2 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

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

$\mathbf{T}$ it is less clear what variables can be excluded.[4] 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*.

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:

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

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

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

$\mathcal{M}'$ 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. 4.2*). The only potential problem lies in the "gluing" of the head and the tail at state i. However, the only constraints containing $\vec{s_i}$ are $\mathbf{T}_{i-1}$ and $\mathbf{T}_i$. But $\mathbf{T}_{i-1}$ does not contain any variables from $\vec{s_i}^{in}$, so letting $\mathcal{M}(\vec{s_i}^{in}) \neq \mathcal{M}'(\vec{s_i}^{in})$ cannot make $\mathbf{T}_{i-1}$ false in $\mathcal{M}'$. Similarly for $\mathbf{T}_i$ which does not contain any variables from $\vec{s_i}^{out}$. Finally $\mathcal{M}(\vec{s_i}^{reg}) = \mathcal{M}(\vec{s_j}^{reg})$, 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 $\mathbf{T}$.

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

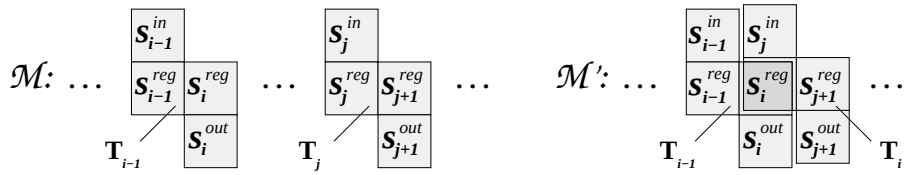


Figure 4.2: The picture shows the contraction of the counter-example $\mathcal{M}$ to $\mathcal{M}'$. The state variables constrained by the transition relations at the point of "gluing" are printed in the boxes; the remaining trace is represented by the "...".

4.5 Experimental Results

The ideas presented in this paper were implemented in the prototype tool TIP[5] 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* from Berkeley University.
eijk	–	ISCAS'89 sequential equivalence checking from [vE98].
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

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

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

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 4.4.1. 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.

[6] These problems were all "TCAS II" problems from the NUSMV distribution, originally used in "Model Checking Large Software Specifications" [CAB+98].

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.

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 sharpening.

4.6 Related Work

Incremental BMC was independently introduced by Ofer Strichman in [Sht01] 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 [Str00].

Related techniques for proving upper bounds for BMC are presented in [KS03a] (computing the recurrence diameter) and [BKA02] (approximating the diameter by structural analysis). In particular, the authors of [KS03a] suggest another solution to the quadratic blow-up of uniqueness constraints by adding a sorting network for the state variables to the SAT-problem.

4.7 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 [Sht01, 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.

4.8 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 [vE98, 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 [Str00, Sht01] for specialized variable orderings and constraint replication.

Chapter 5

Applying Logic Synthesis for Speeding Up SAT

Niklas Eén
Cadence Berkeley Labs, USA

Alan Mishenko
University of California Berkeley, USA

Niklas Sörensson
Chalmers University of Technology and Göteborg University, Sweden

Abstract

SAT solvers are often challenged with very hard problems, which remain unsolved after hours of CPU time. The research community meets the challenge in two ways: (1) by improving the SAT solver technology, for example, perfecting heuristics for variable ordering, and (2) by inventing new ways of simplifying the original SAT problems to make them easier for the solvers, for example, minimizing the number of clauses. This paper explores preprocessing of circuit-based SAT problems using recent advances in logic synthesis. Two fast logic synthesis techniques are considered: DAG-aware logic minimization and a novel type of structural technology mapping, which reduces the size of the CNF derived from the circuit. These techniques are experimentally compared to CNF-based preprocessing. The conclusion is that the proposed techniques are complementary to CNF-based techniques and speed up SAT solving substantially on industrial examples.

5.1 Introduction

Many of today's real-world applications of SAT stem from formal verification, test-pattern generation, and post-synthesis optimization. In all these cases, the SAT solver is used as a tool for reasoning on boolean circuits. Traditionally, instance of SAT are represented on conjunctive normal form (CNF), but for these applications, there is also the option of applying circuit based transformations and reasoning as part of the SAT solving process.

For tougher SAT problems, applying CNF based transformations as a preprocessing step [EB05] has been shown to effectively improve SAT run-times by (1) minimizing the size of the CNF representation, and (2) removing superfluous variables that do not benefit the SAT solving process. In the last decade, advances in logic synthesis has produced powerful and highly scalable algorithms that perform similar tasks on circuits. In this paper, two such techniques are applied to SAT.

The first technique, *DAG-aware circuit compression*, was introduced in the paper [BB04b] and extended in [MCB06]. In this work, it is shown that a circuit can be minimized efficiently and effectively by applying a series of local transformations taking logic sharing into account. Minimizing the number of nodes in a circuit tends to reduce the size of the derived CNFs that are passed to the SAT engine. The process is similar to CNF preprocessing where a smaller representation is also achieved through a series of local rewrites.

The second technique applied in this work is closely related to technology mapping for lookup-table (LUT) based FPGAs. Technology mapping is the task of partitioning a circuit graph into cells with k inputs and one output that fits the LUTs of the FPGA hardware, while using as little area as possible. Many of the signals present in the unmapped circuit become internal to some LUT. In this manner, the procedure can be used to filter out superfluous variables, which is useful to derive a compact CNF.

The purpose of this paper is to draw attention to the applicability of these two techniques in the context of SAT solving. The paper makes a two-fold contribution: (1) it proposes a novel CNF generation based on technology mapping, and (2) it experimenally demonstrated the practicality of the logic synthesis techniques for speeding up SAT.

5.2 Preliminaries

A combinational *boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. Incoming edges of a node are called *fanins* and outgoing edges are called *fanouts*. The *primary inputs* (PIs) of the network are nodes without fanins. The *primary outputs* (POs) are nodes without fanouts. The PIs and POs indicate the external connections of the network.

A special case of a boolean network is the *and-inverter graph* (AIG), containing only PIs, POs, two-input AND-nodes, and the constant TRUE modelled as a node with one output and no inputs. Inverters are represented as complemented attributes on the edges, dividing them into *unsigned* edges and *signed* (or complemented) edges. An AIG is said to be *reduced and constant-free* if (1) all the fanouts of the constant TRUE, if any, feeds into POs; and (2) no AND-

node has both of its fanins point to the same node. Furthermore, an AIG is said to be *structurally-hashed* if no two AND-nodes have the same two fanin edges including the sign. By decomposing k-input functions into two-input ANDs and inverters, any logic network can be reduced to an AIG implementing the same boolean function of the POs in terms of the PIs.

A *cut* C of node n is a set of nodes of the AIG, called *leaves*, such that any path from a PI to n passes through at least one leaf. A *trivial cut* of a node is the cut composed of the node itself. A cut is *k-feasible* if the number of nodes in it does not exceed k. A cut C is *subsumed* by C' of the same node if and $C' \subset C$.

5.3 Cut Enumeration

Here we review the standard procedure for enumerating all k-feasible cuts of an AIG. Let Δ_1 and Δ_2 be two sets of cuts, and the merge operator $\otimes_k$ be defined as follows:

$$\Delta_1 \otimes_k \Delta_2 \;=\; \{\; C_1 \cup C_2 \;\mid\; C_1 \in \Delta_1,\; C_2 \in \Delta_2,\; |C_1 \cup C_2| \leq k \;\}$$

Further, let n_1, n_2 be the first and second fanin of node n, and let $\Phi(n)$ denote all k-feasible cuts of n, recursively computed as follows:

$$\Phi(n) \;=\; \begin{cases} \Phi(n_1) & , n \in \text{PO} \\ \{\{n\}\} & , n \in \text{PI} \\ \{\{n\}\} \;\cup\; \Phi(n_1) \otimes_k \Phi(n_2) & , n \in \text{AND} \end{cases}$$

This formula gives a simple procedure for computing all k-feasible cuts in a single topological pass from the PIs to the POs. Informally, the cut set of an AND node is the trivial cut plus the pair-wise unions of cuts belonging to the fanins, excluding those cuts whose size exceeds k. Reconvergent paths in the AIG lead to generating subsumed cuts, which may be filtered out for most applications.

In practice, all cuts can be computed for $k \leq 4$. A partial enumeration when working with larger k can be achieved by introducing an *order* on the cuts, and keeping only L best cuts at each node. Formally: substitute Φ for Φ_L where $\Phi_L(n)$ is defined as the trivial cut plus the L best cuts of $\Delta_1 \otimes_k \Delta_2$.

5.4 DAG-Aware Minimization

The concept of DAG-aware minimization was introduced by Bjesse et. al. in [BB04b], and further developed by Mishchenko et. al. in [MCB06]. The method works by making a series of local modifications to the AIG, called rewrites, such that each rewrite reduces the *total* number of AIG nodes. To accurately compute the effect of a rewrite on the total number of nodes, *logic sharing* is taken into account. Two equally-sized implementations of a logical function may have different impact on the total node count if one of them contains a subgraph that is already present in the AIG (see *Figure 5.1*).

In [MCB06] the authors propose to limit the rewrites to 4-input functions. There exists $2^{16} = 65536$ such functions. By normalizing the order and polarity of input and output variables, these functions are divided into 222 equivalence

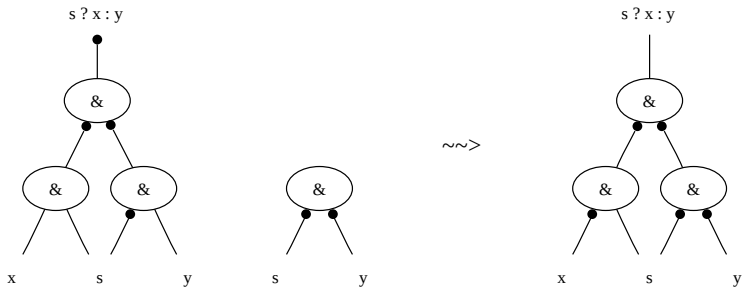


Figure 5.1: Given a netlist containing the two fragments on the left, one node can be saved by rewriting the MUX "s ? x : y" to the form on the right, reusing the already present node "$\neg s \wedge \neg y$".

classes.[1] Good AIG structures, or *candidate implementations*, for these 222 classes can be precomputed and stored in a table. The algorithm of [MCB06] is reviewed below:

> ***DAG-Aware Minimization.*** Perform a 4-feasible cut enumeration, as described in the previous section, proceeding topologically from the PIs to the POs. During the cut enumeration, after computing the cuts depending on the current node n, try to improve its implementation as follows: For every cut C of n, let f be the function of n in terms of the leaves of C. Consider all the candidate implementations of f and choose the one that reduces the total number of AIG nodes the most. If no reduction is possible, leave the AIG unchanged; otherwise recompute the cuts for the new implementation of node n and continue the topological traversal.

Several components are necessary to implement this procedure:

- A cut enumeration procedure, as described in the previous section.
- A bottom-up topological iterator over the AIG nodes that can handle rewrites during the iteration.
- An incremental procedure for structural hashing. In order to efficiently search for the best substitution candidate, the AIG must be kept structurally-hashed, reduced and constant-free. After a rewrite, these properties may be violated and must be restored efficiently.
- A pre-computed table of good implementations for 4-input functions. We propose to enumerate all structurally-hashed, reduced and constant-free AIGs with 7 nodes or less, discarding candidates not meeting the following property: For each node n, there should be no node m in the subgraph rooted in n, such that replacing n with m leads to the same boolean

[1] Often referred to as the NPN-classes, for Negation (of inputs), Permutation (of inputs), Negation (of the output).

function. Example: "$(a \wedge b) \wedge (a \wedge c)$" would be discarded since replacing the node "$(a \wedge b)$" with its subnode "b" does not change the function.

- An efficient procedure to evaluate the effect of replacing the current implementation of a node with a candidate implementation.

The implementation of the above components is straight-forward, albeit tedious. We observe that in principle, the topological iterator can be modified to revisit nodes as their fanouts change. When this happens, new opportunities for DAG-aware minimization may be exposed. Modifying the iterator in this way yields an idempotent procedure, meaning that nothing will change if it is run a second time. In practice, we found it hard to make such a procedure efficient.

A simpler and more useful modification to the above procedure is to run it several times with a *perturbation phase* in between. By changing the structure of the AIG, without changing its functionality or increasing its size, new cuts can conservatively be introduced, with the potential of revealing further node saving rewrites. One way of perturbing the AIG structure is to enumerate all k-input conjunctions and modify their decomposition into two-input AND-nodes ("rebalancing" big ANDs). Another way is to run the above minimization algorithm, but allow for zero-gain rewrites.

5.5 CNF through the Tseitin Transformation

Many applications rely on the standard way of producing CNFs from circuits known as the Tseitin transformation [Tse68]. When applied to AIGs, two improvements are often used: (1) multi-input ANDs are recognized in the AIG structure and translated into clauses as one gate, and (2) if-then-else expressions (MUXes) are detected in the AIG through simple pattern matching and given a specialize CNF translation. The clauses generated for these two cases are:

x $\leftrightarrow$ And(a_1, a_2, ..., a_n). Clause representation:

$$a_1 \wedge a_2 \wedge \ldots \wedge a_n \rightarrow x$$
$$\overline{a}_1 \rightarrow \overline{x},\ \overline{a}_2 \rightarrow \overline{x},\ \ldots\ , \overline{a}_n \rightarrow \overline{x}$$

x $\leftrightarrow$ ITE(s,t,f). If-then-else with selector s, true-branch t, false-branch f. Clause representation:

$$s \wedge t \rightarrow x \qquad \overline{s} \wedge f \rightarrow x \qquad \text{(red)}\ \ t \wedge f \rightarrow x$$
$$s \wedge \overline{t} \rightarrow \overline{x} \qquad \overline{s} \wedge \overline{f} \rightarrow \overline{x} \qquad \text{(red)}\ \ \overline{t} \wedge \overline{f} \rightarrow \overline{x}$$

The two clauses labeled "red" are redundant, but including them increases the strength of unit propagation. It should be noted that a two-input XOR is handled as a special case of a MUX with t and f pointing to the same node in opposite polarity. This results in representing each XOR with four three-literal clauses (the redundant clauses are trivially satisfied). In the experiments presented in section 5.7, the following precise translation was used:

- The *roots* are defined as the AND-nodes with either (1) multiple fanouts; or (2) a single fanout that is either complemented or leads to a PO.

- If a root node is found to define an if-then-else, the above translation with 6 clauses, including redundant clauses, is used.

- The remaining root nodes are encoded as multi-input ANDs. The scope of the conjunction rooted at n is computed as follows: Let S be the set of the two fanins of n. While S contains a non-root node, repeatedly replace that node by its two fanins. The above clause translation for multi-input ANDs is then used, unless the conjunction collected in this manner contains both x and $\neg x$, in which case, a unit clause coding for $x \leftrightarrow$ FALSE is used.

- Unlike some other work [ES06, JS04], there is no special treatment of nodes that occur only *positively* or *negatively*.

5.6 CNF through Technology Mapping

Technology mapping is the process of expressing an AIG in the form representative of an implementation technology, such as standard cells or FPGAs. In particular, *lookup-table (LUT) mapping* for FPGAs consists in grouping AND-nodes of the AIG into logic nodes with no more than k inputs, each of which can be implemented by one LUT.

Normally, technology mapping procedures optimize the area of the mapped circuit under delay constraints. Optimal delay mapping can be achieved efficiently [CC04], but is not useful for SAT where size matters more than logic depth. Therefore, in this paper, we propose to map for area *only*, in such a way that a small CNF can be derived from the mapped circuit. In the next subsections, we review an improved algorithm for structural technology mapping [MCB07]. We modify the algorithm to allow for partial cut enumeration, which further increases its speed and scalability.

5.6.1 Definitions

A *mapping* $\mathbf{M}$ of an AIG is a partial function that takes a non-PI (i.e. AND or PO) node to a k-feasible non-trivial cut of that node. Nodes, for which mapping $\mathbf{M}$ is defined, are called *active* (or mapped), the remaining nodes are called *inactive* (or unmapped). A *proper mapping* of an AIG meets the following three criteria: (1) all POs are active, (2) if node n is active, every leaf of cut $\mathbf{M}(n)$ is active, and (3) for every active AND-node m, there is at least one active node n such that m is a leaf of cut $\mathbf{M}(n)$. The *trivial mapping* (or mapping induced by the AIG) is the proper mapping which takes every non-PI node to the cut composed of its immediate fanins.

An *ordered cut-set* Φ_L is a total function that takes a non-PI node to a non-empty ordered sequence of L or less k-feasible cuts. In the next section, $\mathbf{M}$ and Φ_L as will be viewed as updateable objects and treated imperatively with two operations: For an inactive node n, procedure *activate*($\mathbf{M}$, Φ_L, n), sets $\mathbf{M}(n)$ to the first cut in the sequence $\Phi_L(n)$, and then recursively activates inactive leaves of $\mathbf{M}(n)$. Similarly, for an active node n, procedure *inactivate*($\mathbf{M}$, n), makes node n inactive, and then recursively inactivates any leaf of the former cut $\mathbf{M}(n)$ that is violating condition (3) of a proper mapping.

Furthermore, *nFanouts*($\mathbf{M}$, n) denotes the number of fanouts of n in the subgraph induced by the mapping. The *average fanout* of a cut C, denoted by

avgFanout($\mathbf{M}$, C), is the sum of the number of fanouts of its leaves, divided by the number of leaves. Finally, the *maximally fanout-free cone* (MFFC) of node n, denoted *mffc*($\mathbf{M}$, n), is the set of nodes used exclusively by n. More formally, a node m is part of n's MFFC iff every path in the current mapping $\mathbf{M}$ from m to a PO passes through n. For an *inactive* node, *mffc*($\mathbf{M}$, Φ_L, n) includes the nodes that would belong to the MFFC of node n if it was first activated.

5.6.2 A Single Mapping Phase

Technology mapping includes a sequence of refinement phases, each updating the current mapping $\mathbf{M}$ in an attempt to reduce the *total cost*. The cost of a single cut, *cost*(C), is given as a parameter to the refinement procedure, and the total cost is defined as sum of *cost*($\mathbf{M}(n_{act})$) over all active nodes n_{act}.

Let $\mathbf{M}$ and Φ_L be the proper mapping and the ordered cut-sets from the previous phase. A refinement is performed by a bottom-up topological traversal of the AIG, modifying $\mathbf{M}$ and Φ_L for each AND-node n as follows:

- All k-feasible cuts of node n (with fanins n_1 and n_2) are computed, given the sets of cuts for the children: $\Delta = \{\{n\}\} \cup \Phi_L(n_1) \otimes_k \Phi_L(n_2)$

- If the first element of $\Phi_L(n)$ is not in Δ, it is added. This way, the previously best cut is always eligible for selection in the current phase. It is also a sufficient condition to ensure monotonicity for certain cost functions.

- $\Phi_L(n)$ is set to be the L best cuts from Δ, where smaller cost, higher average fanout, and smaller cut size is better. The best element is put first.

- If n is active in the current mapping $\mathbf{M}$, and if the first cut of $\Phi_L$has changed, the mapping is updated to reflect the change by calling *inactivate*($\mathbf{M}$, n) followed by calling *activate*($\mathbf{M}$, Φ_L, n). After this, $\mathbf{M}$ is guaranteed to be a proper mapping.

5.6.3 The Cost of Cuts

This subsection defines two complementary heuristic cost function for cuts:

> ***Area Flow.*** This heuristic estimates the *global* cost of selecting a cut C by recursively approximating the cost of other cuts that have to be introduced in order to accommodate cut C as follows:
>
> $$cost_{AF}(C) = area(C) + \sum_{n \in C} \frac{cost_{AF}(first(\Phi_L(n)))}{\max(1, nFanouts(\mathbf{M}, n))}$$
>
> ***Exact Local Area.*** For nodes currently not mapped, this heuristic computes the increase in the cost of the mapping incurred by activating n with cut C. For mapped nodes, the computations is the same but n is first deactivated. Formally,

```
cover isop(boolfunc L, boolfunc U)
{
    if (L == FALSE) return ∅
    if (U == TRUE) return {∅}

    x = topVariable(L, U)
    (L0, L1) = cofactors(L, x)
    (U0, U1) = cofactors(U, x)

    c0      = isop(L0 ∧ ¬U1, U0)
    c1      = isop(L1 ∧ ¬U0, U1)
    Lnew    = (L0 ∧ ¬func(c0)) ∨ (L1 ∧ ¬func(c1))
    c*      = isop(Lnew, U0 ∧ U1)

    return ({x} ×  c0) ∪ ({¬x} ×  c1) ∪ c*
}
```

Figure 5.2: *Irredundant sum-of-product generation.* A ***cover*** (= SOP = DNF) is a set, representing a disjunction, of *cubes* (= product = conjunction of literals). A cover c induces a boolean function $func(c)$. An *irredundant SOP* is a cover c where no cube can be removed without changing $func(c)$. In the code, ***boolfunc*** denotes a boolean function of a fixed number of variables $x_1, x_2, \ldots, x_n$ (in our case, the width of a LUT). L and U denotes the lower and upper bound on the cover to be returned. At top-level, the procedure is called with $L = U$. Furthermore, *topVariable(L,U)* selects the first variable, from a fixed variable order, which L or U depends on. Finally, $cofactors(F,\ x)$ returns the pair $(F[x = 0], F[x = 1])$.

$$
\begin{aligned}
mffc(C) &= \bigcup_{n \in C} mffc(\mathbf{M}, \Phi_L, n) \\
cost_{ELA}(C) &= \sum_{n \in mffc(C)} area(first(\Phi_L(n))
\end{aligned}
$$

In the standard FPGA mapping, each cut is given the area of 1 because it takes one LUT to represent it. In the CNF generation, a small but important adjustment is to define area in terms of the number of CNF clauses introduced by that cut. Doing so affects both the area flow and the exact local area heuristic, making them prefer cuts corresponding to functions representable by a small number of clauses.

The boolean function of a cut is translated into clauses by deriving its *irredundant sum-of-products (ISOP)* using Minato-Morreale algorithm [Min92] (reviewed in *Figure 5.2*). ISOPs are computed for both f and $\neg f$ to generate clauses for both sides of the bi-implication $t \leftrightarrow f(x_1, \ldots, x_k)$. For the sizes of k used in the experiments, Boolean functions are efficiently represented using truth-tables. In practice, it is useful to impose an bound on the number of products generated and abort the procedure if it is exceeded, giving the cut an infinitly high cost.

5.6.4 The Complete Mapping Procedure

Depending on the time budget, technology mapping may involve different number of refinement passes. For SAT, only a very few passes seem to pay off. In

our experiments, the following two passes were used, starting from the trivial mapping induced by the AIG:

- An initial pass, using the area-flow heuristic which captures the global characteristics of the AIG.

- A final pass, based on the exact local area heuristic. From the definition of the local area, it is guaranteed not to increase the total cost of the mapping.

Finally, there is a trade-off between the quality of the result and the speed of the mapper, controlled by the cut size k and the maximum number of cuts stored at each node L. To limit the scope of experimental evaluation, these parameters were fixed to $k = 8$ and $L = 5$ for all benchmarks. From a limited testing, these values seemed to be a good trade-off. It is likely that better results could be achieved by setting the parameters in a problem-dependent fashion.

5.7 Experimental Results

To measure the effect of the proposed CNF reduction methods, 30 hard SAT problems represented as AIGs were collected from three different sources. The first suite, "Cadence BMC", consists of internal Cadence verification problems, each of which took more than one minute to solve using SMV's BMC engine. Each of the selected problem contains a bug and has been unrolled upto the length k, which reveals this bug (yielding a satisfiable instance) as well as upto length $k - 1$ (yielding an unsatisfiable instance).

The second suite, "IBM BMC", is created from publically available IBM BMC problems [Zar05]. Again, problems containing a bug were selected and unrolled to length k and $k - 1$. Problems, which MiniSat could not solve in 60 minutes, were remomved, as well as problems solved in under 5 seconds.

Finally, the third suite, "SAT Race", was derived from problems of SAT-Race 2006. Armin Biere's tool "cnf2aig", part of the AIGER package [Bie06], was applied to convert the CNFs to AIGs. Among the problems that could be completely converted to AIGs, the "manol-pipe" class were the richest source. As before, very hard and very easy problems were not considered.

For the experiments, we used the publically available synthesis and verification tool ABC [Gro] and the SAT solver MiniSat2. The exact version of ABC used in these experiments, as well as other information useful for reproducing the experimental results presented in this paper, can be found at: *http://www.cs.chalmers.se/~een/SAT-2007*

Clause Reduction. In *Figure 5.3* we compare the difference between generating CNFs using only the Tseitin encoding (section 5.5) and generating CNFs by applying different combinations of the presented techniques, as well as CNF preprocessing [EB05] (as implemented in MiniSat2). Reductions are measured against the Tseitin encoding. For example, a reduction of 62% means that, on average, the transformed problem contains 0.38 times the original number of clauses.

We see a consistent reduction in the CNF size, especially in the case where CNF was derived using technology mapping. The preprocessing scales well, although its runtime, in our current implementation, is not negligible.

SAT Runtime. In *Figure 5.4* we compare the SAT runtimes of the differently preprocessed problems. Runtimes do *not* include preprocessing times. At this stage, when the preprocessing has not been fully optimized for the SAT context, it is arguably more interesting to see the potential speedup. If the preprocessing is too slow, its application can be controlled by modifying one of the parameters (such as the number of cuts computed), or preprocessing may be delayed until plain SAT solving has been tried for some time without solving the problem.

Speedup is given both as a total speedup (the sum total of all runtimes) and as arithmetic and harmonic average of the individual speedups. For BMC, we see a clear gain in the proposed methods, most notably for the Cadence BMC problems where a total speedup of 6.9x was achieved not using SATELITE style preprocessing, and 5.3x using SATELITE style preprocessing (for a total of 22.3x speedup compared to plain SAT on Tseitin). However, the problems from the SAT-Race benchmark exhibit a different behavior resulting in an increased runtime. It is hard to explain this behavior without knowing the source of the benchmarks, but in this case part of the reason seems to be MINISAT's handling of learned clauses. Periodically, half of the derived clauses are discarded, and the frequency of the period is dependent on the original number of clauses. As the CNFs get smaller by minimization, the heuristic discard learned clauses more often, which seems to affect the solving process negatively for these benchmarks.

CNF Generation based on Technology Mapping. Here we measure the effect of using the number of CNF clauses as the size estimator of a LUT, rather than a unit area as in the standard technology mapping. In both cases, we map using LUTs of size 8, keeping the 5 best cuts at each node during cut enumeration. The results are presented in *Figure 5.7*. As expected, the proposed technique lead to fewer *clauses* but more *variables*. In these experiments, the clause reduction consistently resulted in shorter runtimes of the SAT solver.

Incremental BMC. An alternative and cheaper use of the proposed techniques in the context of BMC, is to minimize the AIG before unrolling. This prevents simplification across different time frames, but is much faster (in our benchmarks, the runtime was negligible). The clause reduction and the SAT runtime using DAG-aware minimization are given in *Figure 5.6*. In this particular experiment, ABC was not used, but an in-house Cadence implementation of DAG-aware minimization and incremental BMC. Ideally, we would like to test the CNF generation based on technology mapping as well, but this is currently not available in the Cadence tool. For licence reasons, IBM benchmarks could not be used in this experiment. Instead, 5 problems from the TIP-suite [Bie06] were used, but they suffer from being too easy to solve.

5.8 Conclusions

The paper explores logic synthesis as a way to speedup solving of circuit-based SAT problems. Two logic synthesis techniques are considered and experimentally evaluated. The first technique applies recent work on DAG-aware circuit compression to preprocess a circuit before converting it to CNF. The second technique is a novel method for directly translating a circuit into a compact CNF using area-oriented technology mapping for LUT-based FPGAs, when the area of a LUT is defined as the number of clauses needed to represent the

Problem	**Clause Reduction** (k clauses) (orig)	S	D	DS	T	TS	DT	DTS	**Preprocessing Time** (sec) S	D	DS	T	TS	DT	DTS
Cdn1-70u	160	113	69	43	54	41	36	29	1	6	7	14	15	11	12
Cdn1-71s	166	117	71	44	55	43	37	30	1	6	6	14	15	12	12
Cdn2-154u	682	452	467	310	312	257	282	254	6	31	35	48	51	66	68
Cdn2-155s	693	459	475	316	318	262	287	259	7	32	36	49	52	67	69
Cdn3.1-18u	1563	813	952	511	905	529	506	306	12	91	99	151	159	189	193
Cdn3.1-19s	1686	898	1028	559	977	593	547	336	12	98	107	162	170	208	212
Cdn3.2-19u	1684	899	1027	561	977	578	547	337	12	98	106	163	171	206	210
Cdn3.2-20s	1807	979	1102	611	1049	612	588	368	13	104	114	175	184	219	224
Cdn3.3-19u	1686	897	1027	560	977	578	547	338	12	100	109	163	171	204	208
Cdn3.3-20s	1809	974	1103	611	1049	647	588	368	14	104	113	174	183	224	229
ibm18-28u	151	95	72	55	67	54	50	48	1	5	6	11	11	11	12
ibm18-29s	158	99	75	57	70	56	53	50	1	5	6	11	12	12	12
ibm20-43u	253	156	127	97	120	99	89	85	2	10	11	19	20	20	21
ibm20-44s	259	161	131	100	123	101	91	88	2	10	11	19	20	21	21
ibm22-51u	415	269	211	160	201	174	149	143	4	16	17	31	33	33	34
ibm22-52s	425	275	216	164	205	178	153	147	4	16	18	32	33	34	34
ibm23-35u	231	147	116	86	100	85	80	76	2	9	9	17	18	18	19
ibm23-36s	239	152	120	89	103	89	83	78	2	9	10	17	18	19	19
ibm29-25u	53	35	28	21	22	20	18	17	0	2	2	4	4	5	5
ibm29-26s	55	36	29	22	24	21	19	18	0	2	3	5	5	5	5
c10id-s	293	273	280	258	177	159	167	151	2	20	21	31	33	46	48
c10nidw-s	643	593	612	563	416	380	394	363	4	47	52	77	84	119	126
c6nidw-i	154	142	147	134	97	89	93	87	1	10	11	18	19	26	27
c7b	41	36	39	33	27	26	26	25	0	3	3	5	6	7	8
c7b-i	40	36	38	33	27	26	26	25	0	3	4	5	5	7	8
c9	23	20	20	17	15	14	13	12	0	2	2	3	3	4	4
c9nidw-s	535	489	507	465	340	312	326	300	4	39	42	66	71	96	101
g10b	128	116	127	111	87	82	83	76	1	9	10	15	16	23	24
g10id	258	240	254	234	161	147	156	143	2	20	21	30	32	47	49
g7nidw	119	110	118	107	78	72	75	70	1	8	8	13	14	20	21
Avg. red.	–	29%	32%	47%	46%	56%	57%	62%							

Figure 5.3: *CNF generation with different preprocessing.* "**(orig)**" denotes the original Tseitin encoding; "**D**" DAG-Aware minimization; "**T**" CNF generation through Technology Mapping; "**S**" SATELITE style CNF preprocessing. On the left, the number of clauses in the CNF formulation is given, in thousands. On the right, the runtimes of applied preprocessing are summed up. No column for the time of generating CNFs through Tseitin encoding is given, as they are all less than a second. The "Cdn" problems are internal Cadence BMC problems; the "ibm" problems are IBM BMC problems from [Zar05]; the remaining ten problems are the "manol-pipe" problems from SAT-Race 2006 [Sin] back-converted by "cnf2aig" into the AIG form.

Problem	**SAT Runtime** (sec) – Cadence BMC (orig)	S	D	DS	T	TS	DT	DTS
Cdn1-70u	21.9	12.3	3.6	3.1	2.5	4.1	**1.2**	1.3
Cdn1-71s	15.2	8.8	7.7	3.9	**2.1**	3.1	4.0	2.7
Cdn2-154u	116.4	48.3	41.1	37.7	11.6	34.4	15.6	**9.3**
Cdn2-155s	101.8	22.9	12.9	16.2	18.2	50.6	13.4	**6.9**
Cdn3.1-18u	1516.0	139.4	361.9	119.4	196.3	78.8	78.8	**39.0**
Cdn3.1-19s	1788.2	276.7	535.0	154.8	317.8	137.1	131.9	**42.5**
Cdn3.2-19u	403.8	214.4	239.8	169.7	140.9	**73.7**	114.8	78.1
Cdn3.2-20s	3066.1	893.4	1002.9	353.2	376.2	313.5	687.5	**96.5**
Cdn3.3-19u	316.1	225.6	133.9	104.7	107.9	107.6	**53.2**	55.0
Cdn3.3-20s	2305.4	456.4	863.1	385.8	507.0	236.9	307.2	**101.2**
Total speedup:		4.2x	3.0x	7.2x	5.7x	9.3x	6.9x	22.3x
Arithmetic average speedup:		3.9x	3.6x	6.5x	6.3x	7.6x	9.2x	19.7x
Harmonic average speedup:		2.7x	2.9x	4.8x	5.3x	4.9x	6.6x	11.5x

Problem	**SAT Runtime** (sec) – IBM BMC (orig)	S	D	DS	T	TS	DT	DTS
ibm18-28u	83.7	82.6	39.2	41.9	45.0	54.2	23.2	**18.5**
ibm18-29s	93.6	47.6	46.8	25.1	36.9	23.5	25.9	**20.9**
ibm20-43u	805.5	890.1	402.3	488.0	540.3	283.6	219.9	**215.1**
ibm20-44s	1260.2	278.4	305.6	**83.8**	277.2	422.2	265.7	303.6
ibm22-51u	361.8	194.6	109.2	88.6	145.8	170.8	**67.0**	82.5
ibm22-52s	408.4	489.0	148.3	135.7	187.2	177.9	120.5	**91.3**
ibm23-35u	540.3	365.9	264.2	241.5	260.1	220.2	181.4	**130.7**
ibm23-36s	856.2	743.4	527.9	356.8	436.2	585.7	**144.7**	238.1
ibm29-25u	329.7	375.6	39.0	29.4	42.9	56.6	28.5	**11.4**
ibm29-26s	71.3	190.5	41.7	**20.9**	71.5	31.5	28.0	25.4
Total speedup:		1.3x	2.5x	3.2x	2.4x	2.4x	4.4x	4.2x
Arithmetic average speedup:		1.5x	3.0x	4.9x	2.8x	2.8x	4.7x	6.5x
Harmonic average speedup:		1.0x	2.4x	3.1x	2.1x	2.4x	4.0x	4.3x

Figure 5.4: *SAT runtime with different preprocessing.* "**(orig)**" denotes the original Tseitin encoding; "**D**" DAG-Aware minimization; "**T**" CNF generation through Technology Mapping; "**S**" SATELITE style CNF preprocessing. Given times do *not* include preprocessing, only SAT runtimes. Speedups are relative to the "**(orig)**" column.

Problem	**SAT Runtime** (sec) – SAT Race (orig)	S	D	DS	T	TS	DT	DTS
c10id-s	26.7	**5.1**	25.1	23.6	50.6	25.2	49.8	14.7
c10nidw-s	710.5	624.7	700.3	880.4	383.6	698.1	**212.7**	856.6
c6nidw-i	414.4	267.1	734.7	412.5	244.5	**209.7**	540.1	710.3
c7b	**29.4**	167.2	76.3	58.4	34.6	43.9	63.9	435.5
c7b-i	101.4	54.2	68.1	52.0	**49.5**	93.2	293.4	154.5
c9	**10.8**	51.2	11.4	32.8	11.8	21.0	44.1	83.1
c9nidw-s	**122.5**	625.2	246.9	864.8	287.2	446.7	952.6	285.2
g10b	385.3	388.8	446.0	183.6	**106.5**	225.6	291.2	182.5
g10id	736.0	350.7	524.0	723.9	98.3	**92.0**	190.6	188.4
g7nidw	119.4	24.8	78.3	67.3	**13.5**	17.2	63.6	37.8
Total speedup:		1.0x	0.9x	0.8x	2.1x	1.4x	1.0x	0.9x
Arithmetic average speedup:		1.8x	1.0x	1.1x	2.8x	2.3x	1.3x	1.4x
Harmonic average speedup:		0.5x	0.8x	0.6x	1.2x	0.9x	0.5x	0.3x

Figure 5.5: *SAT runtime with different preprocessing (cont. from Figure 5.4).*

Problem	Nodes before and after minimization	BMC runtimes before and after minimization
Cdn1	3,527 → 949	37.8 s → 9.6 s
Cdn2	7,918 → 3,126	17.5 s → 0.8 s
Cdn3.1	84,718 → 28,637	607.1 s → 275.3 s
Cdn3.3	84,698 → 28,611	>1 h → 1823.7 s
Cdn4	2,936 → 1,538	>1 h → >1 h
nusmv:tcas$_5$	2,661 → 1,975	9.11 s → 2.27 s
nusmv:tcas$_6$	2,656 → 1,965	4.12 s → 0.67 s
texas.parsesys$_1$	11,860 → 939	0.64 s → 0.03 s
texas.two-proc$_2$	791 → 335	0.23 s → 0.01 s
vis.eisenberg	720 → 306	1.63 s → 2.01 s

Figure 5.6: *Incremental BMC on original and minimized AIG.* The above problems all contain bugs. Runtimes are given for running incremental BMC upto the shortest counter example. In the columns to the right of the arrows, the design has been minimized by DAG-aware rewriting *before* unrolling it. The node count is the number of ANDs in the design. Note that in this scheme, there can be no cross-timeframe simplifications. The experiment confirms the claim in [BB04b] of the applicability of DAG-aware circuit comparession to formal verification. The original paper only listed compression ratios and did not include the runtimes.

boolean function of the LUT.

Experimental results on several sets of benchmarks have shown that the proposed techniques tend to substantially reduce the runtime of SAT solving. The net result of applying both techniques is a 3-4x speedup in solving for hard industrial problems. At the same time, some slow-downs were observed on several benchmarks from the previous year's SAT Race. This indicates that more work is needed for understanding the interaction between the circuit structure and the heuristics of a modern SAT-solver.

5.9 Acknowledgments

The authors acknowledge helpful discussions with Satrajit Chatterjee on technology mapping and, in particular, his suggestion to use the average number of fanins' fanouts as a tie-breaking heuristic in sorting cuts.

Problem	Technology Mapping for CNF		
	#clauses	#vars	SAT-time
Cdn1-70u	62 k → 54 k	12 k → 15 k	6.6 s → 4.1 s
Cdn1-71s	64 k → 55 k	13 k → 15 k	6.6 s → 3.1 s
Cdn2-154u	327 k → 312 k	58 k → 77 k	23.3 s → 34.4 s
Cdn2-155s	333 k → 318 k	58 k → 78 k	21.4 s → 50.6 s
Cdn3.1-18u	1990 k → 905 k	145 k → 248 k	125.9 s → 78.8 s
Cdn3.1-19s	2147 k → 977 k	156 k → 267 k	161.2 s → 137.1 s
Cdn3.2-19u	2146 k → 977 k	156 k → 266 k	189.9 s → 73.7 s
Cdn3.2-20s	2302 k → 1049 k	167 k → 285 k	501.6 s → 313.5 s
Cdn3.3-19u	2147 k → 977 k	156 k → 267 k	136.4 s → 107.6 s
Cdn3.3-20s	2302 k → 1049 k	167 k → 285 k	311.7 s → 236.9 s

Problem	T.M. for CNF after DAG-Aware Minimization		
	#clauses	#vars	SAT-time
Cdn1-70u	50 k → 36 k	11 k → 13 k	2.5 s → 1.3 s
Cdn1-71s	52 k → 37 k	11 k → 13 k	2.0 s → 2.7 s
Cdn2-154u	371 k → 282 k	61 k → 81 k	24.9 s → 9.3 s
Cdn2-155s	378 k → 287 k	62 k → 82 k	17.4 s → 6.9 s
Cdn3.1-18u	1385 k → 506 k	126 k → 185 k	71.0 s → 39.0 s
Cdn3.1-19s	1499 k → 547 k	136 k → 199 k	96.0 s → 42.5 s
Cdn3.2-19u	1495 k → 547 k	136 k → 199 k	165.0 s → 78.1 s
Cdn3.2-20s	1603 k → 588 k	145 k → 214 k	446.6 s → 96.5 s
Cdn3.3-19u	1498 k → 547 k	136 k → 199 k	107.4 s → 55.0 s
Cdn3.3-20s	1607 k → 588 k	145 k → 213 k	273.2 s → 101.2 s

Figure 5.7: *Comparing CNF generation through standard technology mapping and technology mapping with the cut cost function adapted for SAT.* In the adapted CNF generation based on technology mapping (**righthand side of arrows**), the area of a LUT is defined as the number of clauses needed to represent its boolean function. In the standard technology mapping (**lefthand side of arrows**), each LUT has unit area "1". In both cases, the mapped design is translated to CNF by the method described in section 5.6.4, which introduces one variable for each LUT in the mapping. The standard technology mapping minimizes the number of LUTs, and hence will have a lower number of introduced variables. However, from the table it is clear that using the number of clauses as the area of a LUT gives significantly fewer clauses, and also reduces SAT runtimes.

Chapter 6

Translating Pseudo-Boolean Constraints into SAT

Niklas Eén
Cadence Berkeley Labs, USA

Niklas Sörensson
Chalmers University of Technology and Göteborg University, Sweden

Abstract

In this paper, we describe and evaluate three different techniques for translating *pseudo-boolean* constraints (linear constraints over boolean variables) into *clauses* that can be handled by a standard SAT-solver. We show that by applying a proper mix of translation techniques, a SAT-solver can perform on a par with the best existing native pseudo-boolean solvers. This is particularly valuable in those cases where the constraint problem of interest is naturally expressed as a SAT problem, except for a handful of constraints. Translating those constraints to get a pure clausal problem will take full advantage of the latest improvements in SAT research. A particularly interesting result of this work is the efficiency of sorting networks to express pseudo-boolean constraints. Although tangential to this presentation, the result gives a suggestion as to how synthesis tools may be modified to produce arithmetic circuits more suitable for SAT based reasoning.

6.1 Introduction

SAT-solvers have matured greatly over the last five years and have proven highly applicable in the *Electronic Design Automation* field. However, as SAT techniques are being embraced by a growing number of application fields, the need to handle constraints beyond pure propositional SAT is also increasing. One popular approach is to use a SAT-solver as the underlying decision engine for a more high-level proof procedure working in a richer logic; typically as part of an abstraction-refinement loop [BBC$^+$05, BDS02]. Another common approach is to extend the SAT procedure to handle other types of constraints [ES03a], or to work on other domains, such as finite sets or unbounded integers [LKM03, SS05].

In this paper we will study how a SAT-solver can be used to solve *pseudo-boolean* problems by a translation to clauses. These problems are also known as *0-1 integer linear programming* (ILP) problems by the linear programming community, where they are viewed as just a domain restriction on general linear programming. From the SAT point of view, pseudo-boolean constraints (from now on "PB-constraints") can be seen as a *generalization* of clauses. To be precise, a PB-constraint is an inequality on a linear combination of boolean variables: $C_0p_0 + C_1p_1 + \ldots + C_{n-1}p_{n-1} \geq C_n$, where the variables $p_i \in \{0, 1\}$. If all constants C_i are 1, then the PB-constraint is equivalent to a standard SAT clause.

Stemming from the ILP community, pseudo-boolean problems often contain an *objective function*, a linear term that should be minimized or maximized under the given constraints. Adding an objective function is also an extension to standard SAT, where there is no ranking between different satisfiable assignments.

Recently, a number of PB-solvers have been formed by extending existing SAT-solvers to support PB-constraints *natively*, for instance PBS [ARMS02b], PUEBLO [SS06], GALENA [CK03], and, somewhat less recently, OPBDP [Bar95], which is based on pre-Chaff [MMZ$^+$01a] SAT techniques, but still performs remarkably well.

In this paper we take the opposite approach, and show how PB-constraints can be handled through translation to SAT without modifying the SAT procedure itself. The techniques have been implemented in a tool called MINISAT+, including support for objective functions. One of the contributions of our work is to provide a reference to which native solvers can be compared. Extending a SAT-solver to handle PB-constraints natively is, arguably, a more complex task than to reformulate the constraint problem for an existing tool. Despite this, a fair number of native solvers have been created without any real exploration of the limits of the simpler approach.

Furthermore, translating to SAT results in an approach that is particularly suited for problems that are *almost* pure SAT. Given a reasonable translation of the few non-clausal constraints, one may expect to get a faster procedure than by applying a native PB-solver, not optimized towards propositional SAT. Hardware verification is a potential application of this category.

6.2 Preliminaries

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

The pseudo-boolean optimization problem. A *PB-constraint* is an inequality $C_0p_0 + C_1p_1 + \ldots + C_{n-1}p_{n-1} \geq C_n$, where, for all i, p_i is a literal and C_i an integer coefficient. A true literal is interpreted as the value 1, a false literal as 0; in particular $\neg x = (1 - x)$. The left-hand side will be abbreviated by *LHS*, and the right-hand constant C_n referred to as *RHS*. A coefficient C_i is said to be *activated* under a partial assignment if its corresponding literal p_i is assigned to TRUE. A PB-constraint is said to be *satisfied* under an assignment if the sum of its activated coefficients exceeds or is equal to the right-hand side constant C_n. An *objective function* is a sum of weighted literals on the same form as an LHS. The pseudo-boolean optimization problem is the task of finding a satisfying assignment to a set of PB-constraints that minimizes a given objective function.

6.3 Normalization of PB-constraints

PB-constraints are highly non-canonical in the sense that there are many syntactically different, yet semantically equivalent, constraints for which the equivalence is non-trivial to prove. For obvious reasons, it would be practical to get a canonical form of the PB-constraints before translating them to SAT, but to the best of our knowledge, no efficiently computable canonical form for PB-constraints has been found. Still, it makes sense to try to go some of the way by defining a *normal form* for PB-constraints. Firstly, it simplifies the implementation by giving fewer cases to handle; and secondly it may reduce some constraints and make the subsequent translation more efficient. In MINISAT+ we apply the following straightforward rules during parsing:

- $\leq$-constraints are changed into $\geq$-constraints by negating all constants.
- Negative coefficients are eliminated by changing p into $\neg p$ and updating the RHS.
- Multiple occurrences of the same variable are merged into one term C_ix or $C_i\neg x$:
- The coefficients are sorted in ascending order: $C_i \leq C_j$ if $i < j$.
- Trivially satisfied constraints, such as "$x + y \geq 0$" are removed. Likewise, trivially unsatisfied constraints ("$x + y \geq 3$") will abort the parsing and make MINISAT+ answer UNSATISFIABLE.

- Coefficients greater than the RHS are *trimmed* to (replaced with) the RHS.
- The coefficients of the LHS are divided by their greatest common divisor ("gcd"). The RHS is replaced by "RHS/gcd", rounded upwards.

Example: $4x + 3y - 3z \geq -1$ would be normalized as follows:

$$\begin{array}{lcll} 4x + 3y + 3\neg z & \geq & 2 & \text{(positive coefficients)} \\ 3y + 3\neg z + 4x & \geq & 2 & \text{(sorting)} \\ 2y + 2\neg z + 2x & \geq & 2 & \text{(trimming)} \\ y + \neg z + x & \geq & 1 & \text{(gcd)} \end{array}$$

Furthermore, after all constraints have been parsed, trivial conclusions are propagated. Concluding "$x =$TRUE" is considered trivial if setting x to FALSE would immediately make a constraint unsatisfiable. For example "$3x + y + z \geq 4$", would imply "$x =$TRUE". Any assigned variable will be removed from all constraints containing that variable, potentially leading to new conclusions. This propagation is run to completion before any PB-constraint is translated into SAT.

Finally, MINISAT+ performs one more transformation at the PB level to reduce the size of the subsequent translation to SAT. Whenever possible, PB-constraints are split into a *PB part* and a *clause part*. The clause part is represented directly in the SAT-solver, without further translation. *Example:*

$$4x_1 + 4x_2 + 4x_3 + 4x_4 + 2y_1 + y_2 + y_3 \ \geq \ 4$$

would be split into

$$\begin{array}{lcll} x_1 + x_2 + x_3 + x_4 + \neg z & \geq & 1 & \text{(clause part)} \\ 2y_1 + y_2 + y_3 + 4z & \geq & 4 & \text{(PB part)} \end{array}$$

where z is a newly introduced variable, not present in any other constraint. The rule is only meaningful to apply if the clause part contains at least two literals, or if the PB part is empty (in which case the constraint is equivalent to a clause).

For practical reasons, equality constraints ("$3x + 2y + 2z = 5$") are often considered PB-constraints. To get a uniform treatment of inequalities and equalities, MINISAT+ internally represents the RHS of a PB-constraint as an interval $[lo, hi]$ (eg. "$3x + 2y + 2z \in [5, 5]$"). Open intervals are represented by $lo = -\infty$ or $hi = +\infty$. The parser puts some effort into extracting closed intervals from the input, so larger-than singleton intervals may exist. During normalization, closed intervals are first split into two constraints, normalized individually, and then, if the LHS:s are still the same, merged. However, for simplicity of presentation, only the "LHS $\geq$ RHS" form of PB constraints is considered in the remainder of the paper.

A slightly more formal treatment of PB-normalization can be found in [Bar95], and some more general work on integer linear constraints normalization in [Pug91]. We note that there exist 2^{2^n} constraints over n variables. A clause can only express 2^n of these, whereas a PB-constraint can express strictly more. To determine exactly how many more, which would give a number on the expressiveness of PB-constraints, and how to efficiently compute a canonical representation from any given constraint, is interesting future work.

6.4 Optimization – the objective function

Assume we have a PB minimization problem with an objective function $f(\mathbf{x})$. A minimal satisfying assignment can readily be found by iterative calls to the solver. First run the solver on the set of constraints (without considering the objective function) to get an initial solution $f(\mathbf{x_0}) = k$, then add the constraint $f(\mathbf{x}) < k$ and run again. If the problem is unsatisfiable, k is the optimum solution. If not, the process is repeated with the new smaller solution.

To be efficient, the procedure assumes the solver to provide an incremental interface, but as we only *add* constraints, this is almost trivial. A problem with the approach is that if many iterations are required before optimum is reached, the original set of constraints might actually become dominated by the constraints generated by the optimization loop. This will deteriorate performance.

The situation can be remedied by *replacing* the previous optimization constraint with the new one. This is sound since each optimization constraint subsumes all previous ones. In MiniSat+ this is non-trivial, as the constraint has been converted into a number of clauses, possibly containing some extra variables introduced by the translation. Those extra variables might occur among the *learned* clauses [ES03a] of the SAT-solver, and removing the original clauses containing those variables will vastly reduce the pruning power of the learned clauses. For that reason MiniSat+ implements the naive optimization loop above as it stands; but as we shall see, other mechanisms prevent this from hurting us too much.

6.5 Translation of PB-constraints

This section describes how PB-constraints are translated into clauses. The primary technique used by MiniSat+ is to first convert each constraint into a single-output circuit, and then translate all circuits to clauses by a variation of the Tseitin transformation [Tse68]. The inputs of each circuit correspond 1-to-1 to the literals of the PB-constraints. The single output indicates whether the constraint is satisfied or not. As part of the translation to clauses, every output is forced to True. In MiniSat+, there are three main approaches to how a circuit is generated from a PB-constraint:

- Convert the constraint into a BDD.
- Convert the constraint into a network of adders.
- Convert the constraint into a network of sorters.

As circuit representation, we use RBCs (reduced boolean circuits) [ABE00], meaning that the constant signals True and False have been eliminated by propagation, and that any two syntactically identical nodes have been merged by so-called *structural hashing*.

The structural hashing is important as all constraints are first converted to circuits, *then* to clauses. Constraints over similar sets of variables can often generate the same sub-circuits. Structural hashing also prevents the optimization loop from blowing up the SAT problem. Because the optimization constraints differ only in the RHS:s, their translations will be almost identical. Structural hashing will detect this, and few (or no) clauses will be added after the first iter-

ation of the optimization loop. This is very important, as the objective function is often very large.

During the translation to clauses, extra variables are introduced in the CNF to get a compact representation. However, the goal of translating a PB-constraint into clauses is not only to get a compact representation, but also to preserve as many *implications* between the literals of the PB-constraint as possible. This concept is formalized in th CSP community under the name of *arc-consistency*. Simply stated, arc-consistency means that whenever an assignment could be propagated on the original constraint, the SAT-solver's unit propagation, operating on our *translation* of the constraint, should find that assignment too. More formally:

> **Definition.** Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a set of constraint variables, $\mathbf{t} = (t_1, t_2, \ldots, t_m)$ a set of introduced variables. A satisfiability equivalent CNF translation $\phi(\mathbf{x}, \mathbf{t})$ of a constraint $\mathcal{C}(\mathbf{x})$ is said to be *arc-consistent* under unit propagation *iff* for every partial assignment σ, performing unit propagation on $\phi(\mathbf{x}, \mathbf{t})$ will extend the assignment to σ' such that every unbound constraint variable x_i in σ' can be bound to either TRUE or FALSE without the assignment becoming inconsistent with $\mathcal{C}(\mathbf{x})$ in either case.

It follows directly from this definition that if σ is already inconsistent with $\mathcal{C}(\mathbf{x})$ (that is $\mathcal{C}(\mathbf{x})$ restricted to σ is empty), then unit propagation will find a conflict in $\phi(\mathbf{x}, \mathbf{t})$.

Although it is desirable to have a translation of PB-constraints to SAT that maintains arc-consistency, no polynomially bound translation is known. Some important special cases have been studied in [Gen02, BB03, BB04a, BBR06]. In particular, cardinality constraints, $x_1 + x_2 + \ldots + x_n \geq k$, can be translated efficiently while maintaining arc-consistency. Using the notion of arc-consistency, the translations of this paper can be categorized as follows:

- **BDDs.** Arc-consistent but exponential translation in the worst case.
- **Adders.** Not arc-consistent, $O(n)$ sized translation.
- **Sorters.** Not arc-consistent, but closer to the goal. $O(n \log^2 n)$ sized translation.

The parameter n here is the total number of digits[1] in all the coefficients. Translation through sorting networks is closer to the goal than adder networks in the sense that more implications are preserved, and for particular cases (like cardinality constraints), arc-consistency is achieved.

6.5.1 The Tseitin transformation

The first linear transformation of propositional formulas into CNF by means of introducing extra variables is usually attributed to Tseitin [Tse68], although the construction has been independently discovered, in different variations, many times since then. For our purposes, a propositional formula is nothing but a single-output, tree-shaped circuit. The basic idea of the transformation is to introduce a variable for each output of a gate. For example, if the input signals

[1] The radix does not matter for O-notation.

of an AND-gate have been given names a and b, a new variable x is introduced for the output, and clauses are added to the CNF to establish the relation $(x \leftrightarrow a \wedge b)$.

The final step of the transformation is to insert a unit clause containing the variable introduced for the single output of the circuit (or, equivalently, the top-node of the formula). It is easy to see that the models (satisfying assignments) of the resulting CNF are also models of the original propositional formula, disregarding the extra assignments made to the introduced variables.

As observed in [PG86], the transformation can be made more succinct by taking the *polarity* under which a gate occurs into account. A gate is said to occur *positively* if the number of negations on the path from the gate to the output of the circuit is even, and *negatively* if it is odd. Depending on the polarity, only the leftwards or rightwards part of the bi-implication $x \leftrightarrow \phi(a_1, a_2, \ldots, a_n)$, where ϕ is the gate being translated, needs to be introduced. When applying the Tseitin transformation to a *DAG-shaped* circuit,[2] a gate may occur both positively and negatively, in which case the full bi-implication must be established.

In the PB-translations to follow, the gate types listed below will be used. The inverter gate is not part of the list as negations can be handled by choosing the appropriate polarity x or $\neg x$ of a signal, without adding any clauses. In the listed clause representations, variable x always denotes the output of the gate under consideration. Clauses that need to be introduced for a positive/negative occurrence of a gate are marked with a $(+)/(-)$. For brevity, an over-line is used in place of "$\neg$" to denote negation:

- **And($\mathbf{a}_1$, $\mathbf{a}_2$, …, $\mathbf{a}_n$).** N-ary AND-gate. Clause representation:

$$\begin{array}{ll} (-) & a_1 \wedge a_2 \wedge \ldots \wedge a_n \rightarrow x \\ (+) & \overline{a}_1 \rightarrow \overline{x},\ \overline{a}_2 \rightarrow \overline{x},\ \ \ldots\ ,\overline{a}_n \rightarrow \overline{x} \end{array}$$

- **Or($\mathbf{a}_1$, $\mathbf{a}_2$, …, $\mathbf{a}_n$).** N-ary OR gate. Converted to AND by DeMorgan's law.

- **Xor(a,b).** Binary XOR. Clause representation:

$$\begin{array}{llll} (-) & a \wedge \overline{b} \rightarrow x & (+) & a \wedge b \rightarrow \overline{x} \\ (-) & \overline{a} \wedge b \rightarrow x & (+) & \overline{a} \wedge \overline{b} \rightarrow \overline{x} \end{array}$$

- **ITE(s,t,f).** If-then-else node with selector s, true-branch t, false-branch f, and output x. ***Semantics:*** $(s \wedge t) \vee (\neg s \wedge f)$. Clause representation:

$$\begin{array}{lllllll} (-) & s \wedge t \rightarrow x & (-) & \overline{s} \wedge f \rightarrow x & (\text{red}-) & t \wedge f \rightarrow x \\ (+) & s \wedge \overline{t} \rightarrow \overline{x} & (+) & \overline{s} \wedge \overline{f} \rightarrow \overline{x} & (\text{red}+) & \overline{t} \wedge \overline{f} \rightarrow \overline{x} \end{array}$$

  The two "red"-clauses are redundant, but including them will increase the strength of unit propagation.

- **FA_sum(a,b,c).** Output x is the "sum"-pin of a full-adder.[3] ***Semantics:*** $\text{XOR}(a, b, c)$. Clause representation:

[2] Directed Acyclic Graph. A circuit where outputs may feed into more than one gate.

[3] A *full-adder* is a 3-input, 2-output gate producing the sum of its inputs as a 2-bit binary number. The most significant bit is called "carry", the least significant "sum". A half-adder does the same thing, but has only 2 inputs (and can therefore never output a "3").

$$
\begin{array}{ll}
(-)\ \ \overline{a} \wedge \overline{b} \wedge \overline{c} \rightarrow x & (+)\ \ a \wedge b \wedge c \rightarrow \overline{x} \\
(-)\ \ \overline{a} \wedge b \wedge c \rightarrow x & (+)\ \ a \wedge \overline{b} \wedge \overline{c} \rightarrow \overline{x} \\
(-)\ \ a \wedge \overline{b} \wedge c \rightarrow x & (+)\ \ \overline{a} \wedge b \wedge \overline{c} \rightarrow \overline{x} \\
(-)\ \ a \wedge b \wedge \overline{c} \rightarrow x & (+)\ \ \overline{a} \wedge \overline{b} \wedge c \rightarrow \overline{x}
\end{array}
$$

- **FA_carry(a,b,c).** Output x is the "carry"-pin of a full-adder. ***Semantics:*** $a + b + c \geq 2$. Clause representation:

$$
\begin{array}{ll}
(-)\ \ b \wedge c \rightarrow x & (+)\ \ \overline{b} \wedge \overline{c} \rightarrow \overline{x} \\
(-)\ \ a \wedge c \rightarrow x & (+)\ \ \overline{a} \wedge \overline{c} \rightarrow \overline{x} \\
(-)\ \ a \wedge b \rightarrow x & (+)\ \ \overline{a} \wedge \overline{b} \rightarrow \overline{x}
\end{array}
$$

- **HA_sum.** The "sum"-output of a half-adder. Just another name for XOR.

- **HA_carry.** The "carry"-output of a half-adder. Just another name for AND.

For the ITE-gate, two redundant clauses (marked by "red") are added, even though they are logically entailed by the other four clauses. The purpose of these two clauses is to allow unit propagation to derive a value for the gate's output when the two inputs t and f are the same, but the selector s is still unbound. These extra propagations are necessary to achieve arc-consistency in our translation through BDDs (section 6.5.3). In a similar manner, propagation can be increased for the full-adder by adding the following clauses to the representation:[4]

$$
\begin{array}{ll}
x_{carry} \wedge x_{sum} \rightarrow a & \overline{x}_{carry} \wedge \overline{x}_{sum} \rightarrow \overline{a} \\
x_{carry} \wedge x_{sum} \rightarrow b & \overline{x}_{carry} \wedge \overline{x}_{sum} \rightarrow \overline{b} \\
x_{carry} \wedge x_{sum} \rightarrow c & \overline{x}_{carry} \wedge \overline{x}_{sum} \rightarrow \overline{c}
\end{array}
$$

In general, one wants the propagation of the SAT-solver to derive as many unit facts as possible. The alternative is to let the solver make an erroneous assumption, derive a conflict, generate a conflict-clause and backtrack—a much more costly procedure. If we can increase the number of implications made by the unit propagation (the *implicativity*[5] of the CNF) without adding too many extra clauses, which would slow down the solver, we should expect the SAT-solver to perform better. In general it is not feasible (assuming $P \neq NP$) to make all implications derivable through unit propagation by adding a sub-exponential number of clauses; that would give a polynomial algorithm for SAT. However, two satisfiability equivalent CNFs of similar size may have very different characteristics with respect to their implicativity. This partially explains why different encodings of the same problem may behave vastly differently. It should be noted that implicativity is not just a matter of "much" or "little", but also a matter of what cascading effect a particular choice of CNF encoding has. For a specific problem, some propagation paths may be more desirable than others, and the clause encoding should be constructed to reflect this.

[4] Since we have split the full-adder into two single-output gates, we need to keep track of what sums and carries belong together to implement this.

[5] This terminology is introduced in [NB04], but the concept is studied in the CSP community as different *consistency* criteria (and methods to maintain them).

6.5.2 Pseudo-code Conventions

In the pseudo-code of subsequent sections, the type "***signal***" represents either a primary input, or the output of a logical gate. The two special signals "TRUE" and "FALSE" can be viewed as outputs from pre-defined, zero-input gates. They are automatically removed by simplification rules whenever possible. For brevity, we will use standard C operators "&, |, ^" (also in contracted form "&=") to denote the construction of an AND, OR, and XOR gates respectively. The datatype "***vec***" is a dynamic vector, and "***map***" is a hash-table. In the code, we will not make a distinction between word-sized integers and so-called *big-ints*. In a practical implementation, one may need to use arbitrary precision integers for the coefficients of the PB-constraints.

6.5.3 Translation through BDDs

A simple way of translating a PB-constraint into clauses is to build a BDD representation [Bry86] of the constraint, and then translate that representation into clauses. The BDD construction is straightforward, and provided that the final BDD is small, a simple dynamic programming procedure works very efficiently (see Figure 6.3). In general, the variable order can be tuned to minimize a BDD, but a reasonable choice is to order the variables from the largest coefficient to the smallest. A rule of thumb is that important variables, most likely to affect the output, should be put first in the order. In principle, sharing between BDDs originating from different constraints may be improved by a more globally determined order, but we do not exploit that option.

Once the BDD is built, it can simply be treated as a circuit of ITEs (*if-then-else* gates) and translated to clauses by the Tseitin transformation. This is how MINISAT+ works. An example of a BDD translation, before and after reduction to the constant-free RBC representation, is shown in Figure 6.2. An alternative to using the Tseitin transformation, which introduces extra variables for the internal nodes, is to translate the BDD directly to clauses without any extra variables. This is done in [ARMS02a], essentially by enumerating all paths to the FALSE terminal of the BDD. Yet another way to generate clauses from a decision diagram is to synthesize a multi-level And-Inverter graph, for instance by weak-division methods suggested by [Min96], and then apply the Tseitin transformation to that potentially much smaller circuit.

Analysis. BDDs can be used to translate cardinality constraints to a polynomially sized circuit. More precisely, $x_1 + x_2 + \ldots + x_n \geq k$ results in a BDD with $(n - k + 1) \times k$ nodes, as illustrated in Figure 6.1. It is proven that in general a PB-constraint can generate an exponentially sized BDD [BBR06]. Hence, in practice, a limit on the size must be imposed during the BDD construction. However, if the BDD is successfully built and translated by the Tseitin transformation, the resulting CNF preserves arc-consistency.

Proof: For simplicity, consider the ITE circuit without compaction by RBC rules. The result generalizes to RBCs in a straightforward manner. The BDD terminals are translated to literals fixed to TRUE and FALSE. Now, consider a translation of constraint $\mathcal{C}$ under the partial assignment σ. Let p_k be the unbound literal of $\mathcal{C}$ with the highest index. Observation: Since p_k is toggling the biggest coefficient, either $\mathcal{C}, \sigma \models p_k$, or no implications exist. A smaller coefficient cannot be necessary if a bigger one is not. By the Tseitin transformation,

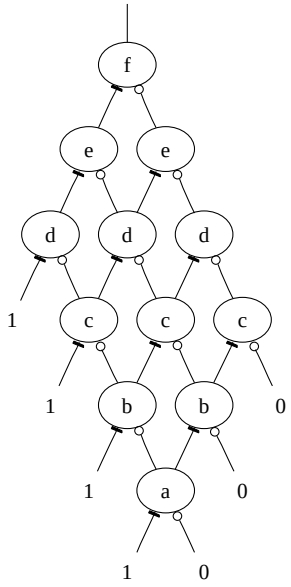


Figure 6.1: BDD for the cardinality constraint $a + b + c + d + e + f \geq 3$.

the single output of the top-node is forced to TRUE. For any node, if the BDD-variable ("selector") is bound, a TRUE on the output will propagate down to the selected child. Because all literals p_i, where $i > k$, are bound and appear above p_k in the BDD (due to the variable order we use), TRUE will propagate down to a unique BDD-node, call it N, branching on p_k. Furthermore, if a BDD-variable is bound, and the selected child is FALSE, it will propagate upwards. Similarly, if both children are bound to FALSE, it will propagate upwards by the redundant "red" clauses of our ITE translation (section 6.5.1). Inductively, if a BDD-node is equivalent to FALSE under σ, the variable introduced for its output will be bound to FALSE by unit propagation. Thus, if $\mathcal{C}, \sigma \models p_k$, the variable introduced for the *false*-branch of N is FALSE, and the output of N is TRUE, which together will propagate $p_k =$TRUE. □

Related Work. The PB-solver PB2SAT [BBR06] also translates PB-constraints to clauses via BDDs. The authors observe that the *false*-branch always implies the *true*-branch if the BDD is generated from a PB-constraint (or indeed any unate function). Using this fact, the Tseitin transformation can be improved to output two 3-clauses and two 2-clauses instead of six 3-clauses,[6] and still maintain arc-consistency.

The translation in [ARMS02a] produces a minimal CNF under the assumption that no extra variables may be introduced. However, even polynomially sized BDDs can have an exponential translation to clauses under that assumption.

[6] Although only half of the clauses will be instantiated if the BDD node occurs with only one polarity.

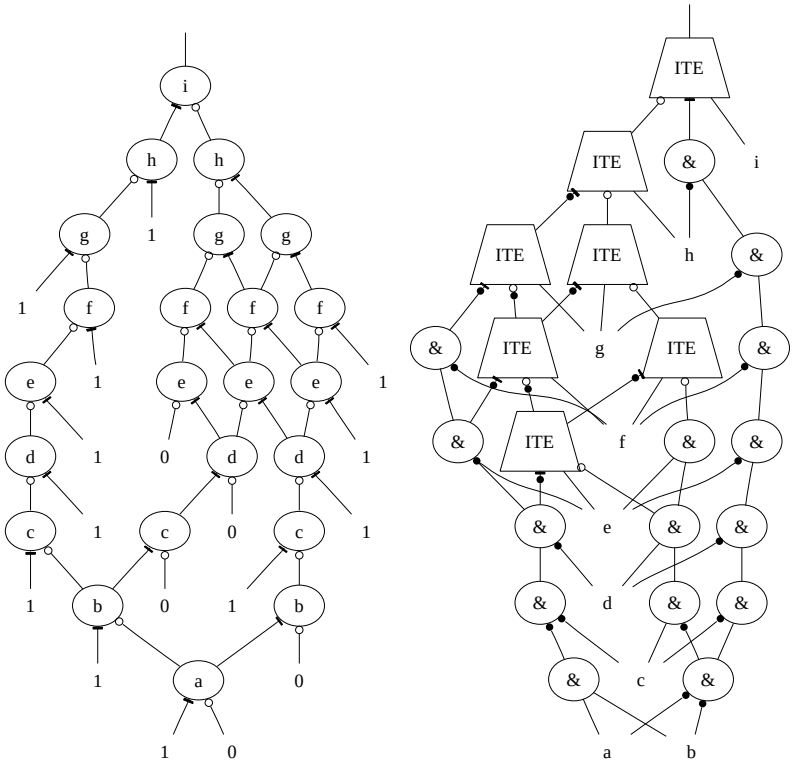


Figure 6.2: *The BDD, and corresponding RBC, for the constraint "*$a + b + 2c + 2d + 3e + 3f + 3g + 3h + 7i \geq 8$*".* The BDD terminals are denoted by "0" and "1". A circle ("o") marks the *false*-branch of a BDD-node, a dash ("-") marks the *true*-branch. In the RBC representation on the right, the unmarked edge is the selector (written, as customary, inside the node for the BDD). A solid circle ("•") denotes negation. Variables are ordered from the largest coefficient to the smallest, which has as consequence that no edge skips a variable in the BDD. Because the *false*-branch implies the *true*-branch of each node (assuming only positive coefficients, as in this example), no edge to the FALSE terminal comes from a *true*-branch.

6.5.4 Translation through Adder Networks

This section describes the translation of PB-constraints into clauses through half- and full-adders. First let us introduce some notation:

> **Definition.** A *k-bit* is a signal whose interpretation in the current context is the numeric value k. We will simply say that the bit *has* the value k.

The concept of k-bits is useful when manipulating bits representing numbers, or sums of numbers. *Example:* When adding two binary numbers "$a_3a_2a_1a_0$" and "b_1b_0", we call a_3 an *8-bit* (it contributes 8 to the sum if set). Signals a_1 and b_1 are the two *2-bits* of the sum.

```
signal buildBDD(vec⟨int⟩ Cs, vec⟨signal⟩ ps, int rhs,
          int size, int sum, int material_left, map⟨pair⟨int,int⟩, signal⟩ memo)
{
    if       (sum ≥ rhs)                    return TRUE
    else if (sum + material_left < rhs) return FALSE

    key = (size, sum)
    if (memo[key] == UNDEF) {
        size−−
        material_left −= Cs[size]
        hi_sum = sign(ps[size]) ? sum : sum + Cs[size]
        lo_sum = sign(ps[size]) ? sum + Cs[size] : sum
        hi_result = buildBDD(Cs, ps, rhs, size, hi_sum, material_left, memo)
        lo_result = buildBDD(Cs, ps, rhs, size, lo_sum, material_left, memo)
        memo[key] = ITE(var(ps[size]), hi_result, lo_result)
    }
    return memo[key]
}
```

Figure 6.3: *Building a BDD from the PB-constraint "***Cs** $\cdot$ **ps** $\geq$ *rhs".* The functions used in the code do the following: "*sign*(p)" returns the sign of a literal (TRUE for $\neg x$, FALSE for x); "*var*(p)" returns the underlying variable of a literal (i.e. removes the sign, if any); and "*ITE*(cond,hi,lo)" constructs an *if-then-else* gate. The types "***vec***$\langle\cdot\rangle$" (a dynamic vector) and "***map***$\langle\cdot,\cdot\rangle$" (a hash table) are assumed to be passed-by-reference (the reasonable thing to do), whereas primitive datatypes, such as "***int***", are assumed to be passed-by-value. The BDD construction is initiated by calling "*buildBDD*()" with "size" set to the number of coefficients in the LHS, "sum" set to zero, "material_left" set to the sum of all coefficients in the LHS, and "memo" to an empty map.

> **Definition.** A *bucket* is a bit-vector where each bit has the same value. A *k-bucket* is a bucket where each bit has the value k.
>
> **Definition.** A *binary number* is a bit-vector where the bits have values in ascending powers of 2 (the standard representation of numbers in computers).

The translation of PB-constraints to clauses is best explained through an example. Consider the following constraint:

$$2a + 13b + 2c + 11d + 13e + 6f + 7g + 15h \geq 12$$

One way to enforce the constraint is to synthesize a circuit which adds up the activated coefficients of the LHS to a binary number. This number can then be compared with the binary representation of the RHS (just a lexicographical comparison). The addition and the corresponding buckets for the above constraint look as follows:

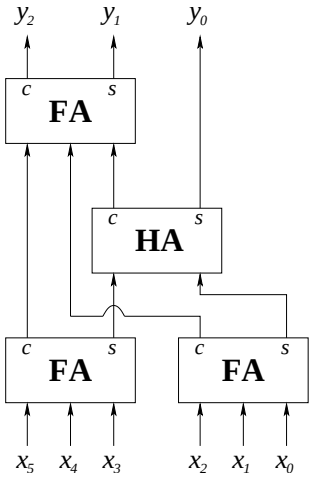


Figure 6.4: Adder circuit for the sum $x_0 + ... + x_5$.

```
  00a0
  bb0b
  00c0
  d0dd
  ee0e
  0ff0
  0ggg
+ hhhh
```

Bucket:	Content:
1-bits:	[b,d,e,g,h]
2-bits:	[a,c,d,f,g,h]
4-bits:	[b,e,f,g,h]
8-bits:	[b,d,e,h]

The goal is to produce the sum as a binary number. It can be done as follows: Repeatedly pick three 2^n-bits from the smallest non-empty bucket and produce, through a full-adder, one 2^{n+1}-bit (the carry) and one 2^n-bit (the sum). The new bits are put into their respective buckets, possibly extending the set of buckets. Note that each iteration eliminates one bit from the union of buckets. When a bucket only has two bits left, a half-adder is used instead of a full-adder. The last remaining bit of the 2^n-bucket is removed and stored as the n^{th} output bit of the binary sum. It is easy to see that the number of adders is linear in the combined size of the initial buckets.

Pseudo-code for the algorithm is presented in Figure 6.5. We note that the buckets can be implemented using any container, but that choosing a (FIFO) queue—which inserts and removes from different ends—will give a balanced, shallow circuit, whereas using a stack—which inserts and removes from the same end—will give an unbalanced, deep circuit. It is not clear what is best for a SAT-solver, but MiniSat+ uses a queue.

For the lexicographical comparison between the LHS sum and the RHS constant, it is trivial to create a linear sized circuit. However, as we expect numbers not to be exceedingly large, we synthesize a comparison circuit that is quadratic in the number of bits needed to represent the sum (Figure 6.6). It has the advantage of not introducing any extra variables in the SAT-solver, as well as producing fewer (although longer) clauses.

Analysis. Adder networks provide a compact, linear (both in time and space) translation of PB-constraints. However, the generated CNF does not preserve arc-consistency under unit propagation. Consider the cardinality constraint $x_0 + x_1 ... + x_5 \geq 4$. The adder network synthesized for the LHS is shown in Figure 6.4. The RHS corresponds to asserting y_2. Now, assume x_0 and x_3 are both FALSE. In this situation, the remaining inputs must all be TRUE, but no assignment will be derived by unit propagation.

Another drawback of using adder networks is the *carry propagation problem*. Assume that x_0 and x_3 are now TRUE instead. The two lower full-adders and the half-adder are summing 1-bits. Ideally, feeding two TRUE-signals should generate a carry-out to the top full-adder, which is summing 2-bits. But because it cannot be determined which of the three outputs to the top full-adder is going to generate the carry, no propagation takes place.

Related Work. The algorithm described in this section is very similar to what is used for *Dadda Multipliers* to sum up the partial products [Dad64]. A more recent treatment on multipliers and adder networks can be found in [BSS01]. Using adder networks to implement a linear translation of PB-constraint to circuits has been done before in the papers [War96, ARMS02a, Sin05], but the construction presented here uses fewer adders.

```
adderTree(vec⟨queue⟨signal⟩⟩ buckets, vec⟨signal⟩ result)
{
    for (i = 0; i < buckets.size(); i++) {
        while (buckets[i].size() ≥ 3) {
            (x,y,z) = buckets[i].dequeue3()
            buckets[i  ].insert(FA_sum(x,y,z))
            buckets[i+1].insert(FA_carry(x,y,z)) }

        if (buckets[i].size() == 2) {
            (x,y) = buckets[i].dequeue2()
            buckets[i  ].insert(HA_sum(x,y))
            buckets[i+1].insert(HA_carry(x,y)) }

        result[i] = buckets[i].dequeue()
    }
}
```

Figure 6.5: *Linear-sized addition tree for the coefficient bits.* The bits of "buckets[]" are summed up and stored in the output vector "result[]" (to be interpreted as a binary number). Each vector is dynamic and extends automatically when addressed beyond its current last element. The "***queue***" could be any container type supporting "*insert*()" and "*dequeue*()" methods. The particular choice will influence the shape of the generated circuit. Abbreviations "FA" and "HA" stand for full-adder and half-adder respectively.

6.5.5 Translation through Sorting Networks

Empirically it has been noted that SAT-solvers tend to perform poorly in the presence of parity constraints. Because all but one variable must be bound before anything can propagate, parity constraints generate few implications during

```
// Generates clauses for "xs ≤ ys", assuming one of them
// has only constant signals.

lessThanOrEqual(vec⟨signal⟩ xs, vec⟨signal⟩ ys, SatSolver S)
{
    // Make equal-sized by padding
    while (xs.size() < ys.size()) xs.push(False)
    while (ys.size() < xs.size()) ys.push(False)

    for (i = 0; i < xs.size(); i++) {
        c = False
        for (j = i+1; j < xs.size(); j++)
            c |= (xs[j] ^ ys[j])          // "c = OR(c, XOR(xs[j], ys[j]))"
        c |= ¬xs[i] | ys[i]               // Note, at this point "c ≠ False"
        S.addClause(c)
    }
}
```

Figure 6.6: *Compare a binary number "xs" to the RHS constant "ys".* One of the input vectors must only contain the constant signals True and False. If not, the function still works, except that "c" is not necessarily a clause when reaching "addClause()", so the call has to be replaced with something that can handle general formulas. *Notation:* the method "*push*()" appends an element to the end of the vector. In practice, as we put signals into the clause "c", we also apply the Tseitin transformation to convert the transitive fan-in ("all logic below") of those signals into clauses. Those parts of the adder network that are not necessary to assert "xs ≤ ys" will not be put into the SAT-solver.

unit propagation. Furthermore, they tend to interact poorly with the resolution based conflict-clause generation of contemporary SAT-solvers. Because full-adders contain Xor (a parity constraint) we might expect bad results from using them extensively. In the previous section it was shown that translating cardinality constraints to adder networks does not preserve arc-consistency, which gives some theoretical support for this claim. Furthermore, the interpretation of a single bit in a binary number is very weak. If, for example, the third bit of a binary number is set, it means the number must be ≥ 8. But if we want to express ≤ 8, or ≥ 6 for that matter, a constraint over several bits must be used. This slows down the learning process of the SAT-solver, as it does not have the right information entities to express conflicts in.

To alleviate the problems inherent to full-adders we propose, as in [BB03], to represent numbers in *unary* instead of in binary. In a unary representation, all bits are counted equal, and the numerical interpretation of a bit-vector is simply the number of bits set to True. A bit-vector of size 8, for example, can represent the numbers $n \in [0, 8]$. Each such number will in the construction to follow be connected to a sorting network,[7] which allows the following predicates to be expressed by asserting a single bit: $n \geq 0$, $n \geq 1$, ..., $n \geq 8$, and $n \leq 0$, $n \leq 1$, ..., $n \leq 8$. Although the unary representation is more verbose than the binary, we hypothesize that the Xor-free sorters increase the implicativity, and that the SAT-solver benefits from having better information entities at its disposal

[7] MiniSat+ uses *odd-even merge sorters* [Bat68].

for conflict-clause generation. The hypothesis is to some extent supported by our experiments, and we furthermore prove that the sorter-based translation of this section is arc-consistent for the special case of cardinality constraints.

To demonstrate how sorters can be used to translate PB-constraints, consider the following example:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + 2y_1 + 3y_2 \geq 4$$

The sum of the coefficients is 11. For this constraint one could synthesize a sorting network of size 11, feeding y_1 into two of the inputs, y_2 into three of the inputs, and all the signals x_i into one input each. To assert the constraint, one just asserts the fourth output bit of the sorter.

Now, what if the coefficients of the constraint are bigger than in this example? To generalize the above idea, we propose a method to decompose the constraint into a number of interconnected sorting networks. The sorters will essentially play the role of adders on unary numbers. Whereas the adder networks of the previous section worked on binary numbers—restricting the computation to buckets of 1-bits, 2-bits, 4-bits, and so on for each power of 2— the unary representation permits us the use of any base for the coefficients, including a mixed radix representation. The first part of our construction will be to find a natural base in which the constraint should be expressed.

> **Definition.** A *base* $\mathbf{B}$ is a sequence of positive integers B_i, either finite $\langle B_0, \ldots, B_{n-1} \rangle$ or infinite.
>
> **Definition.** A *number* $\mathbf{d}$ in a base $\mathbf{B}$ is a finite sequence $\langle d_0, \ldots, d_{m-1} \rangle$ of *digits* $d_i \in [0, B_i - 1]$. If the base is finite, the sequence of digits can be at most one element longer than the base. In that case, the last digit has no upper bound.
>
> **Definition.** Let $\mathit{tail}(s)$ be the sequence s without its first element. The *value* of a number $\mathbf{d}$ in base $\mathbf{B}$ is recursively defined through:
>
> $$\mathit{value}(\mathbf{d}, \mathbf{B}) \;=\; d_0 \;+\; B_0 \times \mathit{value}(\mathit{tail}(\mathbf{d}),\, \mathit{tail}(\mathbf{B}))$$
>
> with the value of an empty sequence $\mathbf{d}$ defined as 0.

Example: The number $\langle 2, 4, 10 \rangle$ in base $\langle 3, 5 \rangle$ should be interpreted as $2 + 3 \times (4 + 5 \times 10) \;=\; 2 \times \mathbf{1} + 4 \times \mathbf{3} + 10 \times \mathbf{15} \;=\; 164$ (bucket values in boldface). Note that numbers may have one more digit than the size of the base due to the ever-present bucket of 1-bits. Let us outline the decomposition into sorting networks:

- Find a (finite) *base* $\mathbf{B}$ such that the sum of all the *digits* of the coefficients written in that base, is as small as possible. The sum corresponds to the number of inputs to the sorters we will synthesize, which in turn roughly estimates their total size.[8]

[8] In MiniSat+, the base is an approximation of this: the best candidate found by a brute-force search trying all prime numbers < 20. This is an ad-hoc solution that should be improved in the future. Finding the optimal base is a challenging optimization problem in its own right.

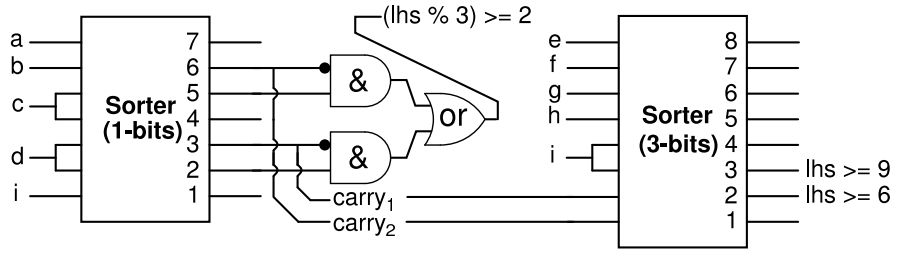


Figure 6.7: *Sorting networks for the constraint "*$a+b+2c+2d+3e+3f+3g+3h+7i \geq 8$*" in base* $\langle 3 \rangle$. Sorters will output 0:s at the top and 1:s at the bottom. In the figure, "lhs" is the value of the left-hand side of the PB-constraint, and "%" denotes the modulo operator. In base $\langle 3 \rangle$, 8 becomes $\langle 2, 2 \rangle$, which means that the most significant digit has to be either > 2 (which corresponds to the signal "lhs ≥ 9" in the figure), or it has to be $= 2$ (the "lhs ≥ 6" signal) and at the same time the least significant digit ≥ 2 (the "(lhs % 3) ≥ 2" signal). For clarity this logic is left out of the figure; adding it will produce the single-output circuit representing the whole constraint.

- Construct one sorting network for each element B_i of the base $\mathbf{B}$. The inputs to the i^{th} sorter will be those digits $\mathbf{d}$ (from the coefficients) where d_i is non-zero, *plus* the potential carry bits from the i-1^{th} sorter.

We will explain the details through an example. Consider the constraint:

$$a + b + 2c + 2d + 3e + 3f + 3g + 3h + 7i \geq 8$$

Assume the chosen base $\mathbf{B}$ is the singleton sequence $\langle 3 \rangle$, meaning we are going to compute with buckets of 1-bits and 3-bits. For the constraint to be satisfied, we must *either* have at least three 3-bits (the LHS ≥ 9), *or* we must have two 3-bits and two 1-bits (the LHS $= 8$). Our construction does not allow the 1-bits to be more than two; that would generate a carry bit to the 3-bits.

In general, let $\mathbf{d}^i$ be the number in base $\mathbf{B}$ representing the coefficient C_i, and $\mathbf{d}^{rhs}$ the number for the RHS constant. In our example, we have:

$$\begin{array}{lcll} \mathbf{d}^0,\ \mathbf{d}^1 & = & \langle 1, 0 \rangle & \text{(digits for the } a \text{ and } b \text{ terms)} \\ \mathbf{d}^2,\ \mathbf{d}^3 & = & \langle 2, 0 \rangle & \text{(digits for the } c \text{ and } d \text{ terms)} \\ \mathbf{d}^4 .. \mathbf{d}^7 & = & \langle 0, 1 \rangle & \text{(digits for the } e, f, g, \text{ and } h \text{ terms)} \\ \mathbf{d}^8 & = & \langle 1, 2 \rangle & \text{(digits for the } i \text{ term)} \\ \mathbf{d}^{rhs} & = & \langle 2, 2 \rangle & \text{(digits for the RHS constant)} \end{array}$$

The following procedure, initiated by "*genSorters*$(0, \emptyset)$", recursively generates the sorters implementing the PB-constraint:

genSorters(n, *carries*)

- Synthesize a sorter of size: $(\sum_i d_n^i) + |carries|$. Inputs are taken from the elements of *carries* and the p_i:s of the constraint: signal p_i is fed to d_n^i inputs of the sorter.[9]

[9]Implementation note: The sorter can be improved by using the fact that the carries are already sorted.

```
// Does the sorter output "out[1..size]" represent
   a number ≥ "lim" modulo "N"?
signal modGE(vec⟨signal⟩ out, int N, int lim)
   if (lim == 0) return TRUE
   result = FALSE
   for (j = 0; j < out.size(); j += N)
        result |= out[j + lim] & ¬out[j+N]
        // let out[n] = FALSE if n is out-of-bounds
   return result
```

```
signal lexComp(int i)
   if (i == 0)
        return TRUE
   else
        i−−
        out = "output of sorters[i]"
        gt = modGE(out, B_i, d_i^rhs + 1)
        ge = modGE(out, B_i, d_i^rhs)
        return gt | (ge & lexComp(i))
```

Figure 6.8: *Adding lexicographical comparison.* In the code, "sorters" is the vector of sorters produced by "*genSorters*()"—at most one more than $|\mathbf{B}|$, the size of the base used. The procedure is initialized by calling "*lexComp*(sorters.*size*())". Let $B_i = +\infty$ for $i = |\mathbf{B}|$ (in effect removing the modulus part of "*modGE*()" for the most significant digit). The $\mathbf{d}^{rhs}$ is the RHS constant written in the base $\mathbf{B}$. Operators "&" and "|" are used on signals to denote construction of AND and OR gates.

- Unless $n = |\mathbf{B}|$: Pick out every B_n:th output bit and put in *new_carries* (the outputs: $out[B_n]$, $out[2B_n]$, $out[3B_n]$ etc.). Continue the construction with *genSorters*$(n + 1$, *new_carries*$)$.

Figure 6.7 shows the result for our example. Ignore for the moment the extra circuitry for modulo operation (the "lhs % $3 \geq 2$" signal). We count the output pins from 1 instead of 0 to get the semantics $out[n]$ = "at least n bits are set".

On the constructed circuit, one still needs to add the logic that asserts the actual inequality LHS $\geq$ RHS. Just like in the case of adder networks, a lexicographical comparison is synthesized, but now on the mixed radix base. The most significant digit can be read directly from the last sorter generated, but to extract the remaining digits from the other sorters, bits contributing to carry-outs have to be deducted. This is equivalent to computing the number of TRUE bits produced from each sorter *modulo* B_i. Figure 6.8 shows pseudo-code for the complete lexicographical comparison synthesized onto the sorters. In our running example, the output would be the signal "$(\text{lhs} \geq 9) \vee ((\text{lhs} \geq 6) \wedge (\text{lhs} \,\%\, 3 \geq 2))$" (again, see Figure 6.7), completing the translation.

Analysis. In base $\langle 2* \rangle = \langle 2, 2, \ldots \rangle$, the construction of this section becomes congruent to the adder based construction described in the previous section.

Sorters now work as adders because of the unary number representation, and as an effect of this, the carry propagation problem disappears. Any carry bit that can be produced *will* be produced by unit propagation. Moreover, the unary representation provides the freedom to pick any base to represent a PB-constraint, which recovers some of the space lost due to the more verbose representation of numbers.

Let us study the size of the construction. An odd-even merge sorter contains $n \cdot log\ n \cdot (1 + log\ n) \in O(n\ log^2 n)$ comparators,[10] where n is the number of inputs. Dividing the inputs between two sorters cannot increase the total size , and hence we can get an upper bound by counting the total number of inputs to *all* sorters and pretend they were all connected to the same sorter. Now, the base used in our construction minimizes the number of inputs, including the ones generated from carry-ins, so choosing the specific base $\langle 2* \rangle$ can only increase the number of inputs. In this base, every bit set to 1 in the binary representation of a coefficient will generate an input to a sorter. Call the total number of such inputs N. On average, every input will generate $1/2$ a carry-out, which in turn will generate $1/4$ a carry-out and so forth, bounding the total number of inputs, including carries, to $2N$. An upper limit on the size of our construction is thus $O(N\ log^2 N)$, where N can be further bound by $\lceil log_2(C_0) \rceil + \lceil log_2(C_1) \rceil + \ldots + \lceil log_2(C_{n-1}) \rceil$, the number of digits of the coefficient in base 2. Counting digits in another base does not affect the asymptotic bound.

By our construction, the cardinality constraint $p_1 + \ldots + p_n \geq k$ translates into a single sorter with n inputs, $p_1, \ldots, p_n$, and n outputs, $q_1, \ldots, q_n$ (sorted in *descending* order), where the k^{th} output is forced to TRUE. We claim that unit propagation preserves arc-consistency. First we prove a simpler case:

Theorem: ASSUME exactly $n - k$ of the inputs p_i are set to 0, and that all of the outputs $q_1, \ldots, q_k$ are set to 1 (not just q_k), THEN the remaining unbound inputs will be assigned to 1 by unit propagation.

Proof: First note that the clauses generated for a single comparator locally preserve arc-consistency. It allows us to reason about propagation more easily. Further note that unit propagation has a unique result, so we are free to consider any propagation order. For brevity write *1*, *0*, and X for TRUE, FALSE and unbound signals.

Start by considering the forward propagation of *0*s. A comparator receiving two *0*s will output two *0*s. A comparator receiving a 0 and an X will output a *0* on the min-output and an X on the max-output. Essentially, the Xs are behaving like *1*s. No *0* is lost, so they must all reach the outputs. Because the comparators comprise a sorting network, the *0*s will appear contiguously at the highest outputs (see Figure 6.9).

Now all outputs are assigned. Consider the comparators in a topologically sorted order, from the outputs to the inputs. We show that if both outputs of a comparator are assigned, then by propagation both inputs must be assigned. From this follows by necessity that the *1*s will propagate backwards to fill the Xs of the inputs. For each comparator, there are two cases (i) both outputs have the same value, in which case propagation will assign both inputs to that value, and (ii) the min-output of the comparator is *0* and the max-output is *1*. In the latter case, the *0* must have been set during the forward propagation of

[10] A comparator is a two-input, two-output gate which sorts two elements. For boolean inputs, the outputs, called "min" and "max", corresponds to an AND-gate and an OR-gate respectively.

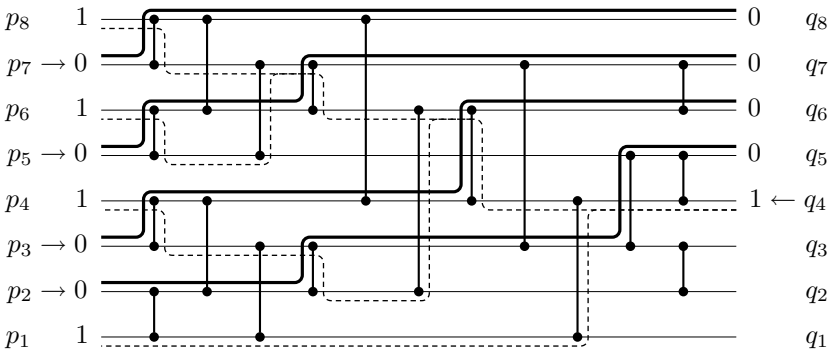
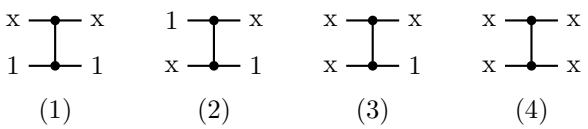


Figure 6.9: *Propagation trough an 8-sorter.* Forced values are denoted by an arrow; four inputs are set to *0* and one output is set to *1*. The thick solid lines show how the *0*s propagate towards the outputs, the thin dashed lines show how *1* propagate backwards to fill the unassigned inputs.

*0*s, so one of the comparator's inputs must be *0*, and hence, by propagation, the other input will be assigned to *1*. □

To show the more general case—that it is enough to set only the output q_k to *1*—takes a bit more work, but we outline the proof (Figure 6.9 illustrates the property):

- Consider the given sorter where some inputs are forced to *0*. Propagate the *0*s to the outputs. Now, remove any comparator (disconnecting the wires) where both inputs and outputs are *0*. For any comparator with both *0* and X inputs (and outputs), connect the X signals by a wire and remove the comparator. Do the same transformation if X is replaced by *1*. Keep only the network reachable from the primary inputs not forced to *0*. In essence: remove the propagation paths of the *0*s.
- Properties of the construction: (a) propagations in the new network correspond 1-to-1 with propagations in the original network, (b) the new network is a sorter.
- The original problem is now reduced to show that for a sorter of size k, forcing its top output[11] will propagate *1*s to all of its inputs (and hence all signals of the network).
- Assume the opposite, that there exists an assignment of the signals that cannot propagate further, but which contains X. From the set of such assignments, pick a maximal element, containing as many *1*s as possible. An X can only appear in one of the following four configurations:



Assume first that (3) and (4) do not exist. In such a network, X behaves exactly as a *0* would do, which means that if we put *0*s on the X inputs,

[11] Assume the orientation of Figure 6.9 for "up" and "down".

they would propagate to the outputs beneath the asserted *1* at the top, which violates the assumption that the network is sorting.
– One of the situations (3) or (4) must exist. Pick a comparator with such a configuration of Xs and set the lower input to *1*. A detailed analysis, considering the four ways Xs can occur around a comparator, shows that unit propagation can never assign a value to the X of the upper output of the selected comparator, contradicting the assumption of the assignment being maximal and containing X. To see this, divide the possible propagations into *forward propagations* (from inputs to outputs) and *non-forward propagations* (from outputs to inputs/outputs). A non-forward propagation can only propagate *1*s leftwards or downwards. A forward-propagation can never initiate a non-forward propagation. □

Unfortunately, arc-consistency is broken by the duplication of inputs, both to the same sorter and between sorters. Duplication within sorters can be avoided by using base $\langle 2* \rangle$, but duplication between sorters will remain. Potentially some improvement to our translation can re-establish arc-consistency in the general case, but it remains future work.

Related Work. Sorting networks is a well-studied subject, treated thoroughly by Knuth in [Knu73]. For a brief tutorial on odd-even merge sorters, the reader is referred to [She03]. In [BB03] an arc-consistent translation of cardinality constraints based on a $O(n^2)$ sized sorter (or "totalizer" by the authors terminology) is presented. The proof above shows that *any* sorting network will preserve arc-consistency when used to represent cardinality constraints. More generally, the idea of using sorting networks to convert non-clausal constraints into SAT has been applied in [KS03b]. The authors sort not just bits, but words, to be able to more succinctly express uniqueness between them. Without sorting, a quadratic number of constraints must be added to force every word to be distinct, whereas the *bitonic sorter* used in that work is $O(n \log^2 n)$. In [Fio99] a construction similar to the one presented in this section is used to produce a multiplier where the partial products are added by sorters computing on unary numbers. Although the aim of the paper is to remove XORs, which may have a slow implementation in silicon, the idea might be even better suited for synthesis geared towards SAT-based verification. Finally, in [WSK04], the carry propagation problem of addition networks is also recognized. The authors solve the problem a bit differently, by preprocessing the netlist and extending the underlying SAT-solver, but potentially sorters can be used in this context too.

6.6 Evaluation

This section will report on two things:

- The performance of MINISAT+ compared to other PB-solvers.
- The effect of the different translation techniques.

It is not in the scope of this paper to advocate that PB-solving is the best possible solution for a particular domain specific problem, and no particular application will be studied.

Small/medium ints. (577 benchmarks)

Solver	#solved-to-opt.
bsolo	187
minisat+	200
PBS4	166
Pueblo	194
sat4jpseudo	139
pb2sat+zchaff	150

Big integers (482 benchmarks)

Solver	#solved-to-opt.
bsolo	9
minisat+	26
PBS4	(buggy)
Pueblo	N/A
sat4jpseudo	3
pb2sat+zchaff	11

Figure 6.10: *PB-evaluation 2005.* Problems with objective function, solved to optimality.

6.6.1 Relative performance to other solvers

No optimization function (113 benchmarks)

Solver	#solved	(sat/unsat)
bsolo	44	(8/36)
minisat+	78	(35/43)
PBS4	89	(28/61)
Pueblo	103	(42/61)
sat4jpseudo	69	(17/52)
pb2sat+zchaff	78	(36/42)

Figure 6.11: *PB-evaluation 2005.* No objective function, just satisfiability (on small integers).

In conjunction with the SAT 2005 Competition, a pseudo-boolean evaluation track was also arranged.[12] MINISAT+ participated in this evaluation together with 7 other solvers. Because not all solvers could handle arbitrary precision integers, the benchmark set was divided into *small*, *medium* and *big* integers. From the results, it seems that only the *big* category was problematic (greater than 30-bit integers), so we have merged the other two categories here. Not all problems had an objective function, so problems were also divided into *optimization* and pseudo-boolean *satisfiability* problems.

Out of the the 8 solvers, 3 was found to be unsound, reporting "UNSAT" for problems where other solvers found verifiable models. However, one of these solver, PBS4, seemed only to be incorrect for *big* integers, so we still consider this solver. But the other two, GALENA and VALLST are excluded from our tables, as they have multiple erroneous "UNSAT" answers, even in the *small* integer category. The results of the remaining solvers can be found in Figure 6.10 and Figure 6.11. In the evaluation, MINISAT+ selected translation method using the following ad-hoc heuristic, applied to each constraint: *If the BDD translation is really compact, use that; otherwise, if the sorting network is not extremely large, use that; otherwise, fall back on adder networks (which are compact).*

From the results we conclude that MINISAT+ and PUEBLO were the two strongest solvers participating in the evaluation. Although it would be interesting to compare these solvers to commercial LP solvers such as CPLEX, for practical reasons this has not been done—no LP solver was part of the evaluation, and CPLEX requires a license.

Constraint (Obj. fct.)	Adders (Adder)	BDDs (Adder)	Sorters (Adder)	Adders (BDD)	BDDs (BDD)	Sorters (BDD)	Adders (Sorter)	BDDs (Sorter)	Sorters (Sorter)
afiro	293 $^{(2.8)}$	299 $^{(5.4)}$	**190** $^{(3.3)}$	936 $^{(4.4)}$	447 $^{(5.4)}$	975 $^{(4.8)}$	470 $^{(2.9)}$	765 $^{(5.4)}$	373 $^{(3.5)}$
sc205	166 $^{(2.4)}$	**3** $^{(2.7)}$	86 $^{(2.8)}$	166 $^{(2.4)}$	**3** $^{(2.7)}$	86 $^{(2.8)}$	166 $^{(2.4)}$	**3** $^{(2.7)}$	86 $^{(2.8)}$
bk4x3	7 $^{(2.4)}$	4 $^{(2.4)}$	10 $^{(2.6)}$	21 $^{(3.9)}$	32 $^{(4.1)}$	15 $^{(3.9)}$	9 $^{(2.9)}$	6 $^{(2.9)}$	**3** $^{(2.9)}$
neos1	– $^{(1.2)}$	– $^{(1.2)}$	– $^{(0.8)}$	– $^{(1.4)}$	– $^{(1.5)}$	90 $^{(1.3)}$	705 $^{(1.2)}$	195 $^{(1.2)}$	**23** $^{(0.9)}$
neos20	8 $^{(1.3)}$	3 $^{(1.3)}$	14 $^{(1.5)}$	9 $^{(1.3)}$	**2** $^{(1.3)}$	13 $^{(1.5)}$	8 $^{(1.3)}$	3 $^{(1.3)}$	14 $^{(1.5)}$
lseu	– $^{(2.6)}$	– $^{(3.0)}$	– $^{(2.9)}$	– $^{(4.2)}$	– $^{(4.2)}$	– $^{(4.2)}$	235 $^{(3.1)}$	**203** $^{(3.3)}$	331 $^{(3.4)}$
misc03	30 $^{(2.2)}$	– $^{(4.8)}$	33 $^{(2.8)}$	**24** $^{(2.1)}$	– $^{(4.8)}$	26 $^{(2.8)}$	27 $^{(2.1)}$	– $^{(4.8)}$	32 $^{(2.8)}$
sample2	4 $^{(2.6)}$	**3** $^{(2.7)}$	6 $^{(2.9)}$	7 $^{(3.9)}$	8 $^{(4.1)}$	5 $^{(3.8)}$	21 $^{(3.6)}$	17 $^{(3.5)}$	17 $^{(3.5)}$
stein45	20 $^{(0.5)}$	25 $^{(0.9)}$	21 $^{(0.7)}$	16 $^{(0.8)}$	18 $^{(1.0)}$	16 $^{(0.9)}$	20 $^{(0.7)}$	20 $^{(0.9)}$	**14** $^{(0.7)}$
enigma	**5** $^{(2.3)}$	122 $^{(4.8)}$	9 $^{(2.9)}$	**5** $^{(2.3)}$	122 $^{(4.8)}$	9 $^{(2.9)}$	**5** $^{(2.3)}$	123 $^{(4.8)}$	9 $^{(2.9)}$
noswot	– $^{(3.4)}$	119 $^{(2.7)}$	– $^{(-\infty)}$	– $^{(3.4)}$	131 $^{(2.7)}$	– $^{(-\infty)}$	– $^{(3.4)}$	**64** $^{(2.7)}$	– $^{(-\infty)}$
p0282	– $^{(2.4)}$	– $^{(2.5)}$	– $^{(2.7)}$	360 $^{(2.1)}$	355 $^{(2.3)}$	368 $^{(2.6)}$	– $^{(3.1)}$	732 $^{(3.3)}$	**281** $^{(3.2)}$
vpm1	– $^{(2.4)}$	830 $^{(3.4)}$	– $^{(2.9)}$	– $^{(2.4)}$	375 $^{(3.4)}$	– $^{(2.9)}$	– $^{(2.4)}$	**176** $^{(3.4)}$	– $^{(2.9)}$
sc50b	– $^{(2.6)}$	147 $^{(2.7)}$	147 $^{(2.8)}$	– $^{(2.6)}$	**39** $^{(2.7)}$	143 $^{(2.8)}$	– $^{(2.6)}$	108 $^{(2.7)}$	139 $^{(3.0)}$
neos8	– $^{(1.6)}$	359 $^{(1.3)}$	**20** $^{(-\infty)}$	– $^{(1.7)}$	263 $^{(1.8)}$	**20** $^{(-\infty)}$	– $^{(1.6)}$	266 $^{(1.4)}$	**20** $^{(-\infty)}$
maros	12 $^{(3.0)}$	**3** $^{(3.2)}$	5 $^{(3.2)}$	77 $^{(4.5)}$	57 $^{(4.5)}$	96 $^{(4.6)}$	10 $^{(3.1)}$	7 $^{(3.3)}$	6 $^{(3.4)}$
l152lav	– $^{(2.8)}$	429 $^{(3.2)}$	704 $^{(3.2)}$	– $^{(2.8)}$	110 $^{(3.4)}$	**43** $^{(3.6)}$	– $^{(2.8)}$	139 $^{(4.4)}$	324 $^{(4.7)}$
mod008	570 $^{(3.6)}$	378 $^{(3.6)}$	355 $^{(3.7)}$	30 $^{(5.4)}$	47 $^{(5.4)}$	356 $^{(5.9)}$	28 $^{(4.6)}$	**20** $^{(4.6)}$	67 $^{(4.6)}$
clip-b	245 $^{(0.9)}$	245 $^{(0.9)}$	245 $^{(0.9)}$	51 $^{(1.4)}$	51 $^{(1.4)}$	51 $^{(1.4)}$	**6** $^{(1.4)}$	**6** $^{(1.4)}$	**6** $^{(1.4)}$
hanoi5	**9** $^{(0.5)}$	**9** $^{(0.5)}$	**9** $^{(0.5)}$	414 $^{(2.6)}$	426 $^{(2.6)}$	426 $^{(2.6)}$	12 $^{(1.2)}$	12 $^{(1.2)}$	12 $^{(1.2)}$
ii32b3	– $^{(0.5)}$	– $^{(0.5)}$	– $^{(0.5)}$	720 $^{(1.8)}$	720 $^{(1.8)}$	721 $^{(1.8)}$	**25** $^{(0.9)}$	**25** $^{(0.9)}$	**25** $^{(0.9)}$
ii32e4	– $^{(0.7)}$	– $^{(0.7)}$	– $^{(0.7)}$	– $^{(1.8)}$	– $^{(1.8)}$	– $^{(1.8)}$	**40** $^{(0.8)}$	**40** $^{(0.8)}$	**40** $^{(0.8)}$
par16-3	**1** $^{(0.7)}$	**1** $^{(0.7)}$	**1** $^{(0.7)}$	44 $^{(2.5)}$	44 $^{(2.5)}$	43 $^{(2.5)}$	**1** $^{(1.4)}$	**1** $^{(1.4)}$	**1** $^{(1.4)}$
ssa7552-159	– $^{(1.0)}$	– $^{(1.0)}$	– $^{(1.0)}$	– $^{(3.0)}$	– $^{(3.0)}$	– $^{(3.0)}$	**21** $^{(1.6)}$	**21** $^{(1.6)}$	**21** $^{(1.6)}$
s4-4-3-2pb	– $^{(1.2)}$	– $^{(1.3)}$	– $^{(1.2)}$	– $^{(2.2)}$	942 $^{(2.2)}$	393 $^{(2.2)}$	131 $^{(1.5)}$	320 $^{(1.6)}$	**8** $^{(1.4)}$
s4-4-3-9pb	26 $^{(1.2)}$	150 $^{(1.2)}$	23 $^{(1.2)}$	454 $^{(2.3)}$	684 $^{(2.2)}$	153 $^{(2.1)}$	14 $^{(1.6)}$	61 $^{(1.7)}$	**9** $^{(1.8)}$
frb30-15-3	– $^{(0.1)}$	– $^{(0.1)}$	– $^{(0.1)}$	761 $^{(0.7)}$	761 $^{(0.7)}$	761 $^{(0.7)}$	**164** $^{(0.3)}$	**164** $^{(0.3)}$	**164** $^{(0.3)}$
frb35-17-5	– $^{(0.1)}$	– $^{(0.1)}$	– $^{(0.1)}$	– $^{(0.5)}$	– $^{(0.5)}$	– $^{(0.5)}$	811 $^{(0.1)}$	811 $^{(0.1)}$	**809** $^{(0.1)}$
woodw	55 $^{(-\infty)}$	**54** $^{(-\infty)}$	55 $^{(-\infty)}$	55 $^{(-\infty)}$	**54** $^{(-\infty)}$	55 $^{(-\infty)}$	55 $^{(-\infty)}$	55 $^{(-\infty)}$	55 $^{(-\infty)}$
chnl10-11	60 $^{(1.5)}$	**20** $^{(1.4)}$	32 $^{(1.5)}$	60 $^{(1.5)}$	**20** $^{(1.4)}$	32 $^{(1.5)}$	60 $^{(1.5)}$	**20** $^{(1.4)}$	32 $^{(1.5)}$
chnl10-20	**24** $^{(1.8)}$	87 $^{(1.5)}$	**24** $^{(1.6)}$	**24** $^{(1.8)}$	87 $^{(1.5)}$	**24** $^{(1.6)}$	**24** $^{(1.8)}$	87 $^{(1.5)}$	**24** $^{(1.6)}$
fpga35-35	– $^{(1.0)}$	74 $^{(0.9)}$	**3** $^{(1.3)}$	– $^{(1.0)}$	75 $^{(0.9)}$	**3** $^{(1.3)}$	– $^{(1.0)}$	75 $^{(0.9)}$	**3** $^{(1.3)}$
fpga40-40	– $^{(1.0)}$	701 $^{(0.9)}$	**2** $^{(1.1)}$	– $^{(1.0)}$	700 $^{(0.9)}$	**2** $^{(1.1)}$	– $^{(1.0)}$	700 $^{(0.9)}$	**2** $^{(1.1)}$
22s-smv	3 $^{(1.0)}$	**1** $^{(0.7)}$	20 $^{(1.2)}$	3 $^{(1.0)}$	**1** $^{(0.7)}$	20 $^{(1.2)}$	3 $^{(1.0)}$	**1** $^{(0.7)}$	20 $^{(1.2)}$
cache-inv12	36 $^{(0.1)}$	**14** $^{(0.0)}$	19 $^{(0.2)}$	36 $^{(0.1)}$	**14** $^{(0.0)}$	19 $^{(0.2)}$	36 $^{(0.1)}$	**14** $^{(0.0)}$	19 $^{(0.2)}$
burch-dill	16 $^{(0.8)}$	**3** $^{(0.4)}$	16 $^{(0.9)}$	16 $^{(0.8)}$	**3** $^{(0.4)}$	16 $^{(0.9)}$	16 $^{(0.8)}$	**3** $^{(0.4)}$	16 $^{(0.9)}$
ex-br-mem	**101** $^{(0.9)}$	117 $^{(0.6)}$	407 $^{(1.1)}$	**101** $^{(0.9)}$	116 $^{(0.6)}$	407 $^{(1.1)}$	**101** $^{(0.9)}$	117 $^{(0.6)}$	407 $^{(1.1)}$
rf10	28 $^{(0.4)}$	**15** $^{(0.2)}$	18 $^{(0.4)}$	28 $^{(0.4)}$	**15** $^{(0.2)}$	18 $^{(0.4)}$	28 $^{(0.4)}$	**15** $^{(0.2)}$	18 $^{(0.4)}$
tag10	96 $^{(0.5)}$	**5** $^{(0.3)}$	16 $^{(0.6)}$	96 $^{(0.5)}$	**5** $^{(0.3)}$	16 $^{(0.6)}$	96 $^{(0.5)}$	**5** $^{(0.3)}$	16 $^{(0.6)}$
100s:	18	17	22	18	19	23	23	24	**29**
1000s:	23	29	28	26	33	33	31	**38**	37

Figure 6.12: *Runtime of* MiniSat+ *on a random selection of benchmarks.* The upper part contains optimization problems with an objective function; in the lower part are pure satisfiability problems. The numbers state runtime in seconds. A dash indicates timeout at 1000 seconds. The super-script numbers give the translation blow-up, written as log_{10}(clauses / pb-constraints). A value of "3" means each constraint was translated to 1000 clauses on average. The two top lines show what translation method was used for the constraints and objective function respectively. The two bottom lines show the total number of problems solved at a timeout of 100/1000 seconds.

6.6.2 Efficiency of different translation techniques

In this section, the three different translation techniques are evaluated on a random sample of benchmarks, drawn from the PB-evaluation set. Each benchmark is evaluated over 9 different parameters to MiniSat+:

- All constraints are translated using one and the same technique (3 choices).
- The objective function is translated using any of the techniques (3 choices).

The reason for treating the objective function separately, is that the optimization constraints generated from it are often very different from the problem constraints. The benchmarks were selected by the following procedure:

- Pick one of the 9 settings for MiniSat+.
- Pick one of the 1176 benchmarks from the evaluation set.
- If MiniSat+ could solve it within 10 minutes, keep it.
- Repeat until 40 benchmarks have been accumulated.

The procedure should give a reasonably unbiased benchmark set.[13] Run-times and translation sizes (relative to the PB-formulation) are presented in Figure 6.12. The experiments were carried out on a cluster of AMD Athlon XP 2800+ machines, each with 1 GB of RAM.

The results indicate that the translation through *adders* does not work well, either for the constraints or the objective function. This is particularly interesting as the adder-translation is the most compact one on average. For the objective function, it seems best to use the sorter-translation. If it is best combined with BDDs or sorters for the problem constraints cannot be concluded from the table, but there are instances where the result of using BDDs differ widely from the result of using sorters. In the table, the translation blow-up includes the objective function which can be dominating, as seen by comparing the results of optimization problems with the results of pure satisfiability problems.

Some remarks about the table: (i) The lower part (below the line) contains problems without objective function. As conversion of the non-existent objective does not affect the result, the same figures occur in three places, modulo timing fluctuation. (ii) The relative blow-up is given in logarithmic scale (the x of 10^x), and so $-\infty$ means that the problem was solved by the parser and pre-processor, producing zero clauses.

[12] http://www.cril.univ-artois.fr/PB05/ (see also [MR06, MMS06, BBR06, ARMS02b, SS06, CK03])

[13] One benchmark was later detected as a duplicate, and therefore removed.

6.7 Conclusions and Future Work

Coding integer arithmetic into boolean operations in a manner well suited for hardware implementation is a thoroughly studied topic. Contrary to this the focus of this paper lies on coding arithmetic, in particular the constructs present in PB-constraints, in ways that are suitable for a SAT-solver. One of the results shown herein is that the compact implementation of adder networks operating on binary numbers works poorly for SAT compared to the more verbose implementation using unary numbers. By change of representation SAT solving could be leveraged into PB solving with very reasonable results, which should have implications on, for example, how SAT-based circuit verification is carried out on designs containing arithmetic. Although making domain specific modification to a SAT-solver is an approach likely to outperform translation based methods in many cases, our belief is that translation is particularly well suited for the kind of problems where SAT-solvers are already successful. An example of such a problem might be finding error-traces in circuits with as many Xs on the inputs as possible. The pragmatics of the approach is also appealing, as encoding into SAT is often much easier than modifying the core solver.

A theoretical result of our study is a proven better bound on the smallest arc-consistent translation of cardinality constraints. Furthermore, our studies reinforce the common opinion that lack of carry propagation in adders are bad for SAT solving, and that more generally high implicativity or arc-consistency is desirable for the subcomponents of a SAT encoding. In the translation using BDDs, arc-consistency was achieved by adding redundant clauses to strengthen unit propagation, which further shows that "small" does not necessarily mean "good" when it comes to CNF encodings. In fact, an interesting branch of future research would be to study how a circuit can be partitioned into chunks that lend themselves to clausification with high implicativity or arc-consistency, such that (a) the encoding is still compact, and (b) the interaction of the components still preserves high implicativity. A more direct future work is to explore the freedom of base-selection in the translation using sorters, in particular by minimizing the number of duplicated inputs rather than the total number of inputs, to increase the implicativity.

6.8 Acknowledgments

The authors wish to express their gratitude to Alan Mishchenko, Armin Biere, Reiner Hähnle, Mary Sheeran and the reviewers for their careful reading and helpful suggestions for improvements of the manuscript. We would also like to thank Christian Szegedy who came up with the generalized version of the proof of arc-consistency of sorters, something we had hitherto only verified experimentally.

Bibliography

[ABE00] P.A. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on SAT-solvers. In *Proceedings of the 6^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000.

[AH01] Gilles Audemard and Laurent Henocque. The eXtended Least Number Heuristic. *Lecture Notes in Computer Science*, 2083, 2001.

[ARMS02a] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Generic ILP vs. Specialized 0-1 ILP: an Update. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 2002.

[ARMS02b] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Pbs: A backtrack search pseudo-boolean solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT'2002)*, 2002.

[Bar95] Peter Barth. A davis-putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[Bat68] K.E. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[BB03] O. Bailleux and Y. Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, CP 2003*, volume 2833. LNCS, 2003.

[BB04a] O. Bailleux and Y. Boufkhad. Problem encoding into sat : the counting constraints case. In *Proceedings of The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'2004)*, 2004.

[BB04b] P. Bjesse and A. Boralv. "dag-aware circuit compression for formal verification". In *Proc. ICCAD'04*, 2004.

[BBC+05] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. v. Rossum, S. Schulz, and R. Sebastiani. The mathsat 3 system. In *Conference on Automated Deduction (CADE-20)*, Springer Verlag, 2005.

[BBR06] O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of pseudo boolean constraints to sat. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 183–192, 2006.

[BC00] Per Bjesse and Koen Claessen. SAT-based Verification without State Space Traversal. In *Formal Methods in Computer-Aided Design*, 2000.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proceedings 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[BCRZ99] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs. In *Proceedings of the 11th International Conference on Computer Aided Verification*, 1999.

[BDS02] C.W. Barret, D.L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to sat. In *Proc. of the 14^{th} Int. Conf. on Computer Aided Verification (CAV'02), LNCS 2404*, 2002.

[Ber] Daniel Le Berre. SAT4J. `http://www.sat4j.org`.

[Bie06] Armin Biere. AIGER (aiger is a format, library and set of utilities for and-inverter graphs (aigs)). *http://fmv.jku.at/aiger/*, 2006.

[BKA02] Jason Baumgartner, Andreas Kuehlmann, and Jacob A. Abraham. Property Checking via Structural Analysis. In *Proceedings of the 14th International Conference on Computer Aided Verification*, 2002.

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

[BS] Roberto Bruttomesso and Natasha Sharygina. Opensmt. `http://code.google.com/p/opensmt/`.

[BSS01] K.C. Bickerstaff, E.E. Swartzlander, and M.J. Schulte. Analysis of column compression multipliers. In *Proceedings of 15^{th} IEEE Symposium on Computer Arithmetic (ARITH-15'01)*, 2001.

[CAB+98] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7), 1998.

[CC04] D. Chen and J. Cong. Daomap: A depth-optimal area optimization mapping algorithm for fpga designs. In *ICCAD*, pages 752–759, 2004.

[CFF+01] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of the 13th International Conference on Computer Aided Verification*, 2001.

[CK03] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of Design Automation Conference (DAC'03)*, pages 830–835, 2003.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[CS00] Koen Claessen and Mary Sheeran. A tutorial on Lava: A hardware description and verification system, 2000.

[CS03] Koen Claessen and Niklas Sörensson. New Techniques that Improve MACE-style Finite Model Finding. In *CADE-19, Workshop W4. Model Computation – Principles, Algorithms, Applications*, 2003.

[Dad64] L. Dadda. Some schemes for parallel multipliers. In *Alta Frequenza*, volume 34, pages 14–17, 1964.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, 1960.

[EB05] N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of Theory and Applications of Satisfiability Testing, 8^{th} International Conference (SAT'2005)*, volume 3569 of *LNCS*, 2005.

[ES03a] Niklas Een and Niklas Sörensson. An extensible sat solver. In *Proceedings of the 6^{th} Int. Conference on Theory and Applications of Satisfiability Testing*, 2003.

[ES03b] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Proceedings of the First International Workshop on Bounded Model Checking*, 2003.

[ES06] N. Een and N. Sörensson. "translating pseudo-boolean constraints into sat". In *Journal on Satisfiability, Boolean Modelling and Computation (JSAT)*, volume 2 of *IOS Press*, 2006.

[Fio99] P.D. Fiore. Parallel multiplication using fast sorting networks. In *IEEE Transactions on Computers, vol 48, no 6*, 1999.

[Gen02] I.P. Gent. Arc consistency in sat. In *Proceedings of the Fifteenth European Conference on Artificial Intelligence (ECAI 2002)*, 2002.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver, 2002.

[Gro] Berkeley Logic Synthesis Group. Abc: A system for sequential synthesis and verification. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[Hoo93] John N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15, 1993.

[HV95] John N. Hooker and V. Vinay. Branching rules for satisfiability. *J. Autom. Reasoning*, 15(3):359–383, 1995.

[JS04] Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. In *Proc. of Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3542 of *LNCS*, 2004.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1:167–187, 1990.

[JZ95] H. Zhang J. Zhang. SEM: a System for Enumerating Models. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1995.

[JZ96] H. Zhang J. Zhang. Generating Models by SEM. In *Proc. of International Conference on Automated Deduction (CADE'96)*, pages 308–312. Springer-Verlag, 1996.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison Wesley, 1973.

[Kor08] Konstantin Korovin. iProver — an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.

[KS03a] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2003.

[KS03b] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In 4^{th} *International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, 2003.

[Lar92] Tracy Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1), 1992.

[LKM03] Cong Liu, Andreas Kuehlmann, and Matthew W. Moskewicz. Cama: A multi-valued satisfiability solver. In *Int. Conf. on Computer Aided Design*, 2003.

[LS05] Inês Lynce and João P. Marques Silva. Efficient data structures for backtrack search sat solvers. *Ann. Math. Artif. Intell.*, 43(1):137–152, 2005.

[MCB06] A. Mishchenko, S. Chatterjee, and R. Brayton. "dag-aware aig rewriting: A fresh look at combinational logic synthesis". In *Proc. DAC'06*, pages 532–536, 2006.

[MCB07] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to technology mapping for lut-based fpgas. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26:240–253, February 2007.

[McC94] W. McCune. A Davis-Putnam Program and Its Application to Finite First-Order Model Search: Quasigroup Existence Problems. Technical report, Argonne National Laboratory, 1994. http://www-unix.mcs.anl.gov/AR/mace/.

[McC03] W. McCune. *Mace4 Reference Manual and Guide*. Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, August 2003. Memo ANL/MCS-TM-264.

[Min92] S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Proc. SASIMI'92*, 1992.

[Min96] S. Minato. Fast factorization method for implicit cube representation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 15, pages 377–384, 1996.

[MMS06] Vasco M. Manquinho and JoÃ£o Marques-Silva. On using cutting planes in pseudo-boolean optimization. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 191–210, 2006.

[MMZ+01a] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, 2001.

[MMZ+01b] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[Mos07] Michal Moskal. Fx7 or in software, it is all about quantifiers, 2007. `http://nemerle.org/~malekith/smt/en.html`.

[MR06] Vasco M. Manquinho and Olivier Roussel. The first evaluation of pseudo-boolean solvers. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 97–136, 2006.

[NB04] Y. Novikov and R. Brinkmann. Foundations of hierarchical sat-solving. In *6^{th} Intl. Workshop on Boolean Problems, (extended ver.: ZIB-Report 05-38, 2005)*, 2004.

[PG86] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. In *Journal on Symbolic Computation 2*, 1986.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[RV00] A. Riazanov and A. Voronkov. Splitting Without Backtracking. Technical Report Preprint CSPP-10, University of Manchester, 2000.

[RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science B.V., 2001.

[Sch01] S. Schulz. A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae. In *Proc. 15th International FLAIRS Conference*, pages 72–76, 2001.

[SG] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. `http://www.eecs.berkeley.edu/~alanmi/abc`.

[She03] Mary Sheeran. Describing and reasoning about sorting networks. In *slides from invited talk at the Nordic Workshop on Programming Theory (http://www.cs.chalmers.se/~ms/Turku.ppt)*, 2003.

[Sht01] Ofer Shtrichman. Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2001.

[Sil99] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA*, pages 62–74, 1999.

[Sin] Carsten Sinz. Sat-race 2006 benchmark set. `http://fmv.jku.at/sat-race-2006/`.

[Sin05] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *11^{th} Int. Conf. on Principles and Practice of Constraint Prog.*, 2005.

[Sla94] John K. Slaney. Finder: Finite domain enumerator - system description. In Alan Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France*, volume 814 of *Lecture Notes in Computer Science*, pages 798–801. Springer, 1994.

[SS96] J. Silva and K. Sakallah. GRASP – A New Search Algorithm for Satisfiability. Technical Report CSE-TR-292-96, University of Michigan, 1996.

[SS05] Hossein M. Sheini and Karem A. Sakallah. A sat-based decision procedure for mixed logical/integer linear problems. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, volume 3524 of *LNCS*, 2005.

[SS06] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-boolean sat solver. In *Journal on Satisfiability, Boolean Modeling and Computation (JSAT'06), issue 2*, pages 157–181, 2006.

[SS07] Geoff Sutcliffe and Christian Suttner. TPTP v. 3.3.0, 2007. `http://www.tptp.org`.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, 2000.

[Str00] Ofer Strichman. Tuning SAT Checkers for Bounded Model Checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, 2000.

[Tam97] Tanel Tammet. Gandalf. *J. Autom. Reasoning*, 18(2):199–204, 1997.

[Tse68] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constr. Math. and Math. Logic*, 1968.

[vE98] C.A.J van Eijk. Sequential Equivalence Checking without State Space Traversal. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 1998.

[War96] J.P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. In *Inf. Proc. Letters, 68, ISSN 0169-118X*, 1996.

[WKS01] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[WSK04] M. Wedler, D. Stoffel, and W. Kunz. Arithmetic reasoning in dpll-based sat solving. In *Design, Automation and Test in Europe Conference*, 2004.

[Zar05] E. Zarpas. Benchmarking sat solvers for bounded model checking. In *Proc. SAT'05*, number 3569 in LNCS. Springer-Verlag, 2005.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, 2001.

[ZS00] Hantao Zhang and Mark E. Stickel. Implementing the davis-putnam method. *J. Autom. Reasoning*, 24(1/2):277–296, 2000.