Department of Computer Science and Engineering

Incremental Satisfiability Solving and its Applications

Siert Wieringa

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 20/2014

Incremental Satisfiability Solving and its Applications

Siert Wieringa

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 14 March 2014 at 12.

Aalto University
School of Science
Department of Computer Science and Engineering

Supervising professor
Assoc. Prof. Keijo Heljanko

Thesis advisor
Assoc. Prof. Keijo Heljanko

Preliminary examiners
Prof. João Marques-Silva, University College Dublin, Ireland
Dr. rer. nat. Carsten Sinz, Karlsruhe Institute of Technology, Germany

Opponent
Prof. Karem Sakallah, University of Michigan. Currently on leave at the Qatar Computing Research Institute.

Aalto University publication series
DOCTORAL DISSERTATIONS 20/2014

© Siert Wieringa

ISBN 978-952-60-5568-8
ISBN 978-952-60-5569-5 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-5569-5

Unigrafia Oy
Helsinki 2014

Finland

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Siert Wieringa

Name of the doctoral dissertation
Incremental Satisfiability Solving and its Applications

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 20/2014

Field of research Computer Science and Engineering

Manuscript submitted 1 November 2013 **Date of the defence** 14 March 2014

Permission to publish granted (date) 17 January 2014 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

The propositional logic satisfiability problem (SAT) is a computationally hard decision problem. Despite its theoretical hardness, decision procedures for solving instances of this problem have become surprisingly efficient in recent years. These procedures, known as SAT solvers, are able to solve large instances originating from real-life problem domains, such as artificial intelligence and formal verification. Such real-life applications often require solving several related instances of SAT. Therefore, modern solvers posses an incremental interface that allows the input of sequences of incrementally encoded instances of SAT. When solving these instances sequentially the solver can reuse some of the information it has gathered across related consecutive instances.

This dissertation contains six publications. The two focus areas of the combined work are incremental usage of SAT solvers, and the usage of parallelism in applications of SAT solvers. It is shown in this work that these two seemingly contradictory concepts form a natural combination. Moreover, this dissertations unifies, analyzes, and extends the results of the six publications, for example, by studying information propagation in incremental solvers through graphical visualizations.

The concrete contributions made by the work in this dissertation include, but are not limited to: Improvements to algorithms for MUS finding, the use of graphical visualizations to understand information propagation in incremental solvers, asynchronous incremental solving, and concurrent clause strengthening.

Keywords Incremental satisfiability solving, parallel satisfiability solving, applications of satisfiability solving

ISBN (printed) 978-952-60-5568-8 **ISBN (pdf)** 978-952-60-5569-5

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2014

Pages 218 **urn** http://urn.fi/URN:ISBN:978-952-60-5569-5

Contents

Contents 1

Preface 3

List of Publications 5

Author's Contribution 7

1. Introduction 9

1.1 Contributions of the publications 10

1.2 New contributions in this dissertation 11

2. Definitions 13

2.1 Incremental solver usage 16

2.2 Parallel SAT solving 18

2.3 Tools 18

3. Visualizing incremental solver behavior 21

3.1 The hyperactive variable visualization 22

3.2 The clause involvement visualization 25

3.3 The timed clause involvement visualization 30

4. Model Checking 31

4.1 Circuits 34

4.2 Properties 35

4.3 Bounded Model Checking 37

4.4 Completeness 39

4.5 Analyzing the solver usage 41

4.6 IC3 and PDR 47

4.7 The solver usage of IC3 and PDR 49

5. Finding Minimal Unsatisfiable Subsets **55**
5.1 Classical algorithms for MUS finding 56
5.2 Constructive algorithm using associated assignments 58
5.3 Model rotation 59
5.4 Weakening the termination condition 62
5.5 Blocked rotation edges 63
5.6 Proof of a conjecture by Belov et al. 65
5.7 Using the solver efficiently 68
5.8 Redundancy removal techniques 72

6. Asynchronous incremental solving using Tarmo **79**
6.1 Distribution modes 82
6.2 Conflict clause sharing 83
6.3 Interactive graphical visualizations 85
6.4 Applications 86

7. Cube and Conquer **87**
7.1 The weakness of search space splitting 88
7.2 Cube solving phase: Independent, incremental or parallel . 90

8. Concurrent Clause Strengthening **93**
8.1 The solver-reducer architecture 94
8.2 Employing concurrency versus parallelization 95
8.3 Applications and competitions 95

9. Conclusions **97**

Bibliography **99**

A. SMV model for Example 4.10 **109**

B. Errata for the publications **111**

Publications **113**

Preface

The defense of this dissertation is the final step of the doctoral research project which I conducted at the Department of Information and Computer Science at Aalto University. I wish to thank all my colleagues for the good time I had in the past years. The two department heads during these years, Prof. Pekka Orponen and Prof. Ilkka Niemelä, were always easily accessible, which contributed to the pleasant working atmosphere. The defense of this work will be facilitated by the Department of Computer Science and Engineering, to which I have been transferred while I was in the final stages of writing this dissertation. The most important source of funding for this research project has been the Academy of Finland. The personal grants I received from the Nokia Foundation and Wihurin Rahasto are also gratefully acknowledged.

I would like to thank my pre-examiners Prof. João Marques-Silva and Dr. Carsten Sinz for their professional evaluation of this dissertation. I thank Prof. Karem Sakallah for agreeing to act as an opponent for the defense of this work. I wish to thank my co-authors Dr. Hans van Maaren, Dr. Marijn Heule, Dr. Oliver Kullmann, and M.Sc. Matti Niemenmaa. I especially would like to thank my co-author Prof. Armin Biere, as he has motivated me tremendously on several occasions. In particular, he encouraged me to write what became Publication II, which was a crucial step in defining my own research direction. I thank my supervisor and co-author Assoc. Prof. Keijo Heljanko for all his help, first facilitating my life in Finland and then guiding me through this research project. Above all I wish to thank him for giving me the space and time I needed to find my own direction.

For comments and advice regarding my research Dr. Tommi Junttila was always available, and hence I would like to thank him for the positive influence he had on this work. The final professional contact I wish to

thank is Dr. Niklas Eén. Niklas and his wife Luige let me live and work for one month at their family home in Berkeley, California. This was a unique experience, which was very motivational and inspirational. It was simply wonderful to receive so much trust and generosity from one of the most important researchers in my field.

Outside the academic world I want to thank my friends at Teekkarien Autokerho, who form the basis of my social life in Finland. I thank Ossi Väänänen for helping me with the Finnish language in the dissertation release. I want to thank my parents, as they have always supported me regardless of which path I decided to take. I also wish to especially mention my deceased friend Dick, just because he would have liked that. I think we both considered our friendship as something special.

Finally, I would like to thank my fiancée Céline, who at the time of the defense of this dissertation will have become my wife. The love and attention she has given me have made my life so much more valuable. Her continuous support has made me grow in many ways, and it is hard to imagine how I would have finished this work without her.

Espoo, Finland, February 10, 2014,

Siert Wieringa

List of Publications

This dissertation consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Hans van Maaren and Siert Wieringa. Finding Guaranteed MUSes Fast. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Lecture Notes in Computer Science, volume 4996, pages 291-304. Guangzhou, China, May 2008.

II Siert Wieringa. On Incremental Satisfiability and Bounded Model Checking. In *Proceedings of the First International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS)*, CEUR workshop proceedings, volume 832, pages 46-54. Workshop affiliated to the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, Texas, November 2011.

III Marijn J.H. Heule and Oliver Kullmann and Siert Wieringa and Armin Biere. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Revised Selected Papers of the 7th international Haifa Verification Conference (HVC)*, Lecture Notes in Computer Science, volume 7261, pages 50-65. Haifa, Israel, December 2011, published 2012.

IV Siert Wieringa. Understanding, Improving and Parallelizing MUS Finding Using Model Rotation. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science, volume 7514, pages 672-687. Québec City, Canada, October 2012.

V Siert Wieringa and Keijo Heljanko. Asynchronous Multi-core Incremental SAT Solving. In *Proceedings of the 19th international conference on Tools and Algorithms for the Construction of Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, volume 7795, pages 139-153. Held as part of European Joint Conferences on Theory and Applications of Software (ETAPS), Rome, Italy, March 2013.

VI Siert Wieringa and Keijo Heljanko. Concurrent Clause Strengthening. In *Proceedings of the 16th international conference on Theory and Applications of Satisfiability Testing (SAT)*, Lecture Notes in Computer Science, volume 7962, pages 116-132. Helsinki, Finland, July 2013.

Author's Contribution

Publication I: "Finding Guaranteed MUSes Fast"

The author of this dissertation is responsible for the writing of Publication I, as well as for the development of the software described in it. The presented algorithm is derived from an original idea by Hans van Maaren.

Publication II: "On Incremental Satisfiability and Bounded Model Checking"

The author of this dissertation is solely responsible for Publication II.

Publication III: "Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads"

The author of this dissertation is responsible for the discussions on incremental SAT and "parallel solving of the cubes" in Publication III. Furthermore, the author performed extensive empirical evaluations of the presented technique, both during its development and for the results section presented in the publication.

Publication IV: "Understanding, Improving and Parallelizing MUS Finding Using Model Rotation"

The author of this dissertation is solely responsible for Publication IV.

Publication V: "Asynchronous Multi-core Incremental SAT Solving"

The author of this dissertation is responsible for the contents of Publication V. The second author of the publication, Keijo Heljanko, has provided comments to the manuscript, guidance, and supervision.

Publication VI: "Concurrent Clause Strengthening"

The author of this dissertation is responsible for the contents of Publication VI. The second author of the publication, Keijo Heljanko, has provided comments to the manuscript, guidance, and supervision.

1. Introduction

The dissertation consists of six publications and a unifying introduction, studying approaches to solving the *propositional satisfiability* problem. The two concepts that form the focus of this dissertation are *incremental solver usage* and *parallelism*. Propositional satisfiability, which is typically abbreviated SAT, is the problem of finding a satisfying truth assignment for a given propositional logic formula, or determining that no such assignment exists. This classifies the formula as respectively *satisfiable* or *unsatisfiable*. SAT is an important theoretical problem as it was the first problem ever to be proven NP-complete [Coo71].

Despite the theoretical hardness of SAT, current state-of-the-art decision procedures for SAT, known as *SAT solvers*, have become surprisingly efficient. Subsequently these solvers have found many industrial applications. Such applications are rarely limited to solving just one decision problem. Instead, a single application will typically solve a sequence of related problems. Modern SAT solvers handle such problem sequences through their *incremental SAT* interface [WKS01, ES03b]. Using this interface repeatedly loading common subformulas can be avoided. More importantly, it allows the solver to reuse information across several related consecutive problems. The resulting performance improvements make incremental SAT a crucial feature for modern SAT solvers in real-life applications.

Incremental solving can provide performance improvements, for example, for algorithms that view a SAT solver as an `NP`-oracle to which they perform repeated calls. An example of this type of usage is in algorithms for finding minimal unsatisfiable subsets [Mar10], which are discussed in Publication I and Publication IV. Other applications of incremental solvers include efficient implementations of abstraction refinement loops, guiding the search of a solver, and even handling queries to a symbolically

represented database.

As even the most modest personal computer is nowadays equipped with one or several multi-core processors it is logical and worthwhile to study the use of concurrency in applications of SAT solvers. One may view incremental usage of a solver as a means of solving several related formulas sequentially using one process, whereas parallel SAT solving is usually described as a method for solving one formula using multiple concurrent processes. From that point of view, using a solver incrementally and employing parallelism seem like exact opposites. However, if we take a slightly more high-level view it becomes clear that these two complimentary techniques form a natural combination. In many applications concurrency can be efficiently employed by performing several independent subtasks simultaneously (e.g. [SEMB11]). In between parallel solving of a single formula, and performing completely independent subtasks simultaneously, there is the option of combining incremental solving and parallelism, by solving several related problems concurrently, as we showed in Publication V.

1.1 Contributions of the publications

Publication I. Proposes a constructive algorithm for MUS finding. Although the algorithm itself can no longer compete with the current state-of-the-art algorithms it provided two significant contributions to the field of MUS finding. The first is an improvement of constructive MUS finding algorithms by the addition of a redundancy test. The second contribution is the use of satisfying assignments that are returned by the solver as a result of these redundancy tests, in order to reduce the total number of such tests required by the algorithm.

Publication II. Discusses *Bounded Model Checking*, a practical application of incremental SAT solvers, and the sequences of formulas this application generates. The publication discusses the difference between solving the formulas independently and solving them sequentially using an incremental solver, in terms of solver run time. A visualization of variable activity is proposed, which gives some insight into how information propagates across consecutive related formulas. These insights are related to the observed run times of the solver.

Publication III. Proposes *Cube and Conquer*, a method which aims to combine the strengths of SAT solvers of the look-ahead and CDCL type. A look-ahead solver is used to partition a SAT formula into tens of thousands of subformulas, each of which is solved using a CDCL solver. The method has been shown to perform well for several hard instances of SAT.

Publication IV. Discusses the excellent practical performance of a technique for improving MUS finding algorithms called *model rotation*, and aims to provide an explanation for this performance. It shows that for formulas which possess certain common properties model rotation is guaranteed to be successful. Furthermore, it proposes an algorithmic optimization of the technique, and discusses parallelization of MUS finding algorithms.

Publication V. Discusses parallel solving of incrementally encoded formula sequences. It proposes the *asynchronous interface*, a natural extension of the most commonly used incremental solver interface that allows efficient parallelization for applications using incremental solvers. This interface is implemented in the solver TARMO, which successfully participated in the Hardware Model Checking Competitions of 2011 and 2012.

Publication VI. Advocates performing additional reasoning in parallel with conventional single-threaded SAT solvers. It proposes one concrete instance of this general idea, called the *solver-reducer architecture*. In this architecture, a conventional CDCL solver is extended with a second computation thread, which is solely used to strengthen the clauses learned by the solver. This provides a simple and natural way to exploit the widely available multi-core hardware. The technique is empirically shown to provide a consistent run time reduction for solving unsatisfiable benchmarks. More impressively, the average performance is shown to improve with respect to the total amount of CPU time required.

1.2 New contributions in this dissertation

Besides the discussion of the existing work in the attached publications, this dissertation contains several new contributions. In Chapter 3 we propose a new visualization of incremental solver behavior, called the clause involvement visualization. This visualization is used in a new study of the behavior of solvers used by BMC algorithms provided in Chapter 4, which significantly extends the work of Publication II. Moreover, Chap-

ter 4 provides a discussion of the solver usage of the recent IC3 and PDR algorithms. Chapter 5 also includes several new elements, such as the discussion of undocumented features of an implementation of the algorithm from Publication I, and a discussion of redundancy removal techniques. Moreover, the work of Publication IV is extended with two new contributions. Namely, a proof that minimal unsatisfiable formulas may contain clauses that are unreachable by model rotation, and a proof of a conjecture from [BLM12].

2. Definitions

In this chapter we provide definitions that we will use throughout this document. We start by defining the notation we use for basic propositional logic concepts, followed by a description of several basic algorithmic concepts relating to incremental and parallel SAT solving. We conclude this chapter with a discussion of the tools used for the experiments that are discussed in this document.

A *literal* l is either a Boolean variable x or its negation $\neg x$. We define $\neg\neg l = l$, to represent that double negations cancel out. An *assignment* α is a set of literals such that if $l \in \alpha$ then $\neg l \notin \alpha$. If $l \in \alpha$ we say that literal l is assigned the value **true**. If $\neg l \in \alpha$ it is said that l is assigned the value **false**, or equivalently that l is *falsified*. If for some literal l neither l nor $\neg l$ is in the assignment α then l is *unassigned*. For an assignment α we denote by $\neg\alpha$ the negation of the assignment, which is defined as $\neg\alpha = \{\neg l \mid l \in \alpha\}$.

A *clause* c is a set of literals $c = \{l_0, l_1, \cdots, l_n\}$ representing the disjunction $\bigvee c = l_0 \vee l_1 \cdots \vee l_n$. Clause c is *satisfied* by assignment α if and only if c contains a literal assigned **true** by α, i.e. $c \cap \alpha \neq \emptyset$. A *cube* d is a set of literals $d = \{l_0, l_1, \cdots, l_n\}$ representing the conjunction $\bigwedge d = l_0 \wedge l_1 \cdots \wedge l_n$. Hence, cube d is *satisfied* by assignment α iff $d \subseteq \alpha$. A propositional logic formula is in *conjunctive normal form* (CNF) if it is formed as a conjunction of disjunctions, i.e. a set of clauses. A CNF formula $\mathcal{F}$ is satisfied by an assignment α if every clause in $\mathcal{F}$ is satisfied by α. If no such assignment exist then the CNF formula is *unsatisfiable*.

Conventional SAT solvers can only handle formulas represented in CNF. Therefore, throughout this work the word formula will always refer to a propositional logic formula in CNF form. The only exception is in the definition of several simple example formulas, which for clarity are not written in CNF. In such cases conversion to CNF is always implicitly assumed

as a part of the solving process.

For a clause c, let $Var(c)$ be the set of Boolean variables represented by the literals of c. For a formula $\mathcal{F}$, let $Var(\mathcal{F})$ be the set of all Boolean variables that appear in $\mathcal{F}$, i.e. $Var(\mathcal{F}) = \bigcup\{\, Var(c) \mid c \in \mathcal{F}\}$. An assignment α is *complete* for a formula $\mathcal{F}$ if no variable in $Var(\mathcal{F})$ is unassigned in α.

A clause consisting of exactly one literal is called a *unit clause*. The negation of a clause c should represent a logical constraint that forbids all assignments satisfying c. Hence, it is defined as $\neg c = \{\{\neg l\} \mid l \in c\}$, which is a set of unit clauses. Note the difference with the definition of the negation of an assignment, which is a set of literals.

The formula $\mathcal{F}$ *under the assignment* α is denoted $\mathcal{F}^\alpha$. It is defined as the result of removing all clauses satisfied by α from $\mathcal{F}$, followed by shrinking the remaining clauses by removing literals that are falsified by α. Formally:

$$\mathcal{F}^\alpha = \{\ c \setminus \neg\alpha \ \mid\ c \in \mathcal{F} \text{ and } c \cap \alpha = \emptyset\ \}.$$

Let $iup(\mathcal{F}, \alpha)$ be the assignment α that is the result of executing the following *iterative unit propagation* loop:

$$\textbf{while} \text{ there exists a unit clause } \{l\} \in \mathcal{F}^\alpha \textbf{ do } \alpha = \alpha \cup \{l\}.$$

Moreover we define $\mathcal{F}|_\alpha = \mathcal{F}^{iup(\mathcal{F},\alpha)}$, which is the result of simplifying formula $\mathcal{F}$ under assignment α by iterative unit propagation. If $\emptyset \in \mathcal{F}|_\alpha$ we say that assignment α is a *conflicting assignment*. If on the other hand $\mathcal{F}|_\alpha = \emptyset$ then assignment α satisfies $\mathcal{F}$.

The DPLL algorithm [DLL62] is the classical algorithm for determining the satisfiability of CNF formulas. It starts from the formula $\mathcal{F}$ and an empty assignment α, and alternates between iterative unit propagation and *branching decisions*. During a branching decision, or simply *decision*, the algorithm picks a *decision variable* x_d that is unassigned by α and assigns it to either **true** or **false**. Whenever iterative unit propagation leads to a conflict the algorithm backtracks to the last decision to which it had not backtracked before, and negates the assignment made at that decision. This backtracking search continuous until either all variables of $\mathcal{F}$ are assigned, or all branches of the search tree have been unsuccessfully explored. In the former case α satisfies $\mathcal{F}$, in the latter case $\mathcal{F}$ is unsatisfiable.

Most of the modern SAT solvers applied in practical applications are of the *Conflict Driven Clause Learning* (CDCL) type [MS96, MMZ+01]. Just like the basic DPLL procedure the search for a satisfying assignment pro-

ceeds by alternating between iterative unit propagation and branching decisions. The crucial difference is in what happens when a conflict is reached. In this case, a CDCL solver will analyze the sequence of decisions and implications that lead to the conflict. During this *conflict analysis* the solver derives a *conflict clause*, which is a clause implied by the input formula that gives a representation of the "cause" of the conflict. By including the conflict clause in the set of clauses on which iterative unit propagation is performed hitting another conflict with the same cause can be avoided. The database storing these clauses in the solver is called the *learnt clause database*, and we will often refer to the conflict clauses in this database as *learnt clauses*.

An important property of the most popular clause learning scheme for CDCL solvers, called *first unique implication point* (1-UIP) [MS96], is that each conflict clause contains exactly one literal that was falsified by the last decision or the subsequent unit propagation. This literal is called the *asserting literal*. After conflict analysis the CDCL solver must backtrack. Unlike the DPLL procedure CDCL solvers use non-chronological backtracking, which is driven by the conflict clauses. By definition all literals in a conflict clause are assigned the value **false** by assignment α when it is derived. After learning conflict clause c, the solver backtracks until the earliest decision at which all literals of c except the asserting literal l_a are assigned **false**. The literal l_a is then assigned the value **true**, as this is required to satisfy c. Subsequent unit propagation may yield a new conflict which is handled in the same way.

Variable State Independent Decaying Sum (VSIDS) [MMZ$^+$01] is an important heuristic used in modern CDCL solvers. It associates with every variable a value called the *activity* of the variable. Whenever a conflict clause is derived the activity of all variables involved in its derivation is increased. Periodically the activity of all variables is decreased. A modern CDCL solver will prefer branching on variables with a high activity. Note that those are the variables that have been most actively involved in recent conflict clause derivations. The *two watched literal scheme* was also proposed in [MMZ$^+$01]. It provides an efficient way of implementing iterative unit propagation, and remains a crucial engineering feature in today's SAT solvers.

2.1 Incremental solver usage

An early work considering incremental SAT solving is [Hoo93]. The main idea was to speed-up the solving of a single formula by repeatedly solving a growing subset of its constraints. Removal of constraints from the solver was not considered.

A general definition for the incremental satisfiability problem is given in [WKS01], where it is defined as solving each formula in a finite sequence of formulas. The transformation from a formula to its successor in the sequence is defined by two sets, a set of clauses to be added and a set of clauses to be removed. Although it is possible to implement a SAT solver that allows arbitrary removal of clauses between consecutive formulas, there is a complication in that when a clause is removed also all conflict clauses whose derivation depends on that clause must be removed. Maintaining sufficient information in the solver to achieve this has significant drawbacks on its performance and thus arbitrary clause removal is not implemented in any state-of-the-art solver.

Alternative solutions exists. For example, in the interface of the SAT solver ZCHAFF [MFM04], it is possible to assign clauses to groups, and those groups can be removed as a whole. The SMT-LIB standard [BST10] for SMT solver[1] input defines the *push- and pop-interface*. In this approach the subproblems are maintained on a stack and the solver aims to solve the union of the problems on that stack. The simplest and most commonly used interface for incremental SAT solvers however is the one defined in [ES03b] and first used in the solver MINISAT [ES03a]. This solver interface does not contain a function for removing clauses. Instead, a solver with this interface can determine the existence of satisfying assignments that include a specified set of *assumptions*. The interface is defined by the following two functions:

- `addClause(Clause clause)`
- `solve(Cube assumptions)`

Calling `solve` given assumptions cube d will make the solver determine the satisfiability of $\mathcal{F}^d$ rather than that of $\mathcal{F}$. Using this interface clause removal can be simulated as follows: Instead of adding clause c to the solver the clause $c \cup \{\neg x\}$ is added, where x is a variable that does not

[1] Solvers for *Satisfiability Modulo Theories* (SMT) allow input formulas that combine pure propositional logic with other logics.

occur in the original formula, called a *selector variable*. When the formula is solved under an assumptions cube d such that $x \in d$ the solver is forced to search for an assignment satisfying c, in order to satisfy $c \cup \{\neg x\}$. However, without the assumption x the solver can assign x to **false** to satisfy $c \cup \{\neg x\}$ without having to satisfy c. In fact, we can even add the unit clause $\{\neg x\}$ to make sure that x will remain falsified forever. Modern solvers such as MINISAT will in this case (eventually) delete the satisfied clause $c \cup \{\neg x\}$ from memory completely.

With the exception of [NR12], solving under n assumptions is implemented by forcing the first n branching decisions to assign the assumption literals **true**. If the solver reaches a conflict that requires it to undo any of these forced decisions then a *final conflict* clause is derived, and the solver reports the result "unsatisfiable under assumptions". The final conflict clause represents a subset of the assumptions that is sufficient to yield this result.

In this dissertation an input sequence for an incremental SAT solver will be considered as a sequence of *jobs* $\langle \phi_0, \phi_1, \cdots \rangle$. A job ϕ_i is characterized by a set of clauses $\texttt{CLS}(\phi_i)$ and a single cube $\texttt{assumps}(\phi_i)$. Each job ϕ_i induces a formula $\mathcal{F}(\phi_i)$ consisting of all its clauses and all clauses in previous jobs, and one unit clause for each literal in its cube of assumptions.

$$\mathcal{F}(\phi_i) = \underbrace{\left(\bigcup_{0 \le j \le i} \texttt{CLS}(\phi_j) \right)}_{\texttt{CLAUSES}(\phi_i)} \cup \left(\bigcup_{l \in \texttt{assumps}(\phi_i)} \{l\} \right).$$

Note that these definitions appear also in Publication V. They have been chosen to match solvers using the interface of [ES03b]. Calling `addClause(`c`)` for all $c \in \texttt{CLAUSES}(\phi_i)$ followed by a call to `solve(assumps(`ϕ_i`))` will make such solver solve $\mathcal{F}(\phi_i)$.

In the rest of this work "solving a job" refers to the process of determining the satisfiability of the CNF formula induced by that job. Also, we will often refer to job ϕ_i as simply job i, where i denotes the position of the job in the sequence counting from 0. Performing a *solver call* refers to calling the solver's `solve` function, and thus performing one solver call corresponds to solving one job. This terminology is used interchangeably.

SAT solvers can be used in one of two ways. Either they are used to solve a single formula which is stored on disk, typically in the *DIMACS file format*[2]. Or, they are integrated into an application that uses the

[2]See, e.g. rules of the SAT competitions: `http://www.satcompetition.org`

solver as part of some more high-level task. The latter type of usage has typically been considered to be the only way to exploit the incremental features of the solver. In [WNH09] we defined the *iCNF file format* for incremental SAT solving. This is a simple extension of the DIMACS format in which not only clauses, but also assumption sets can be represented. In this way, job sequences can be stored on disk, and subsequently used as benchmarks for testing the incremental features of SAT solvers.

2.2 Parallel SAT solving

Two major approaches for parallelizing SAT algorithms can be distinguished [HJN09]. The first is the classic divide-and-conquer approach, which aims to partition the formula to divide the total workload evenly over multiple SAT solver instances [BS96, SLB09, ZBH96]. The second approach is the *portfolio* approach [HJS09, XHHLB08, MSSS12]. Portfolio systems do not partition the input formula, but rather run multiple solvers in parallel each of which attempt to solve the same formula. The system finishes whenever the fastest solver is done. Both approaches can be extended with some form of learnt clause sharing between the solver threads (e.g. [AHJ+12]). Although other techniques have recently been developed (e.g. [HJN11] and Publication III) portfolio solvers have received the majority of the research attention in recent years. Some insight into the performance of these approaches is provided in [HM12]. The limited parallelizability of the proof system underlying modern SAT solvers is studied in [KSSS13].

2.3 Tools

Throughout this document we use MINISAT as the SAT solver for experiments. It was written by Niklas Eén and Niklas Sörensson [ES03a], and is a commonly used baseline in research on satisfiability solvers. Its implementation is easy to understand and extend, while still offering decent performance. We use version 2.2.0 which is the last version officially released by Niklas Sörensson. We did not modify the solver itself, although we modified its file parser to read the iCNF file format, and we added several datastructures and printing routines to obtain the extra information needed to draw the visualizations presented in this document.

The second tool we use throughout this document is AIGBMC, by Armin Biere. This is a simple implementation of a Bounded Model Checking (BMC) algorithm (see Chapter 4). Its input file format is the AIGER format, a representation of Boolean Circuits using only and-gates, inverters, and latches. Although AIGBMC has features for checking liveness properties, in this work we used it only for simple safety properties, i.e. invariants. The encoding of the BMC problem into SAT provided by AIGBMC for such properties is a simple initialized unrolling of the circuit, with a constraint forcing a violation of the invariant in the last timepoint, as is explained in Chapter 4 (see Example 4.8). We made several modifications to AIGBMC, and in this document will always refer to this modified version. The first modification we made was to replace SAT solver PICOSAT [Bie08] used in the original version by MINISAT. The second modification was to add an option which makes AIGBMC continue to encode and solve jobs when it has already found a counterexample, i.e. solved a job with result satisfiable. The third modification was to add an option to solve only one single job individually, without using the solver's incremental features. Links to all the tools used, including their extensions and modifications, are available in the online support material that can be found at: `http://www.siert.nl/thesis`.

3. Visualizing incremental solver behavior

Incremental SAT solvers are used in a diverse range of applications. In this document we analyze the behavior exposed by these solvers for several such applications. In order to unify our analysis we propose the use of graphical visualizations, starting with the *hyperactive variable visualization* proposed in Publication II. Other work on visualizations for SAT solvers focusses on the visualization of the input formula, as well as on visualization of steps performed within the DPLL-algorithm [Sin07]. The authors of [DZK13], aim to visualize the execution of a MUS finding algorithm which internally uses a SAT solver. Both [Sin07] and [DZK13] go beyond simply visualizing the algorithm and also allow the user to interact. In this way, the user can resolve non-determinism and manually make choices that are otherwise left to search heuristics. A special version of the ASP solver[1] CLASP [GKNS07], called CLAVIS, provides a means of logging information to create visualizations of problem structure, heuristic information, and execution steps.

The previously mentioned tools aim at visualizing as much information as possible about the execution of their algorithm and the content of their data structures. As such they can be useful to demonstrate how solvers work on small problems, but their application to studying the behavior on large scale problems is severely restricted. With billions of decisions, hundreds of thousands of conflicts, and gigabytes of data stored in the solver, visualizing everything is simply not an option.

As in this work we are particularly concerned with the behavior of SAT solvers used incrementally we focus our analysis on the propagation of information between jobs. When a SAT solver is used incrementally it reuses learnt clauses, as well as other information such as the activity of

[1]Answer Set Programming (ASP) is a form of declarative programming. ASP solvers and SAT solvers internally use many similar techniques.

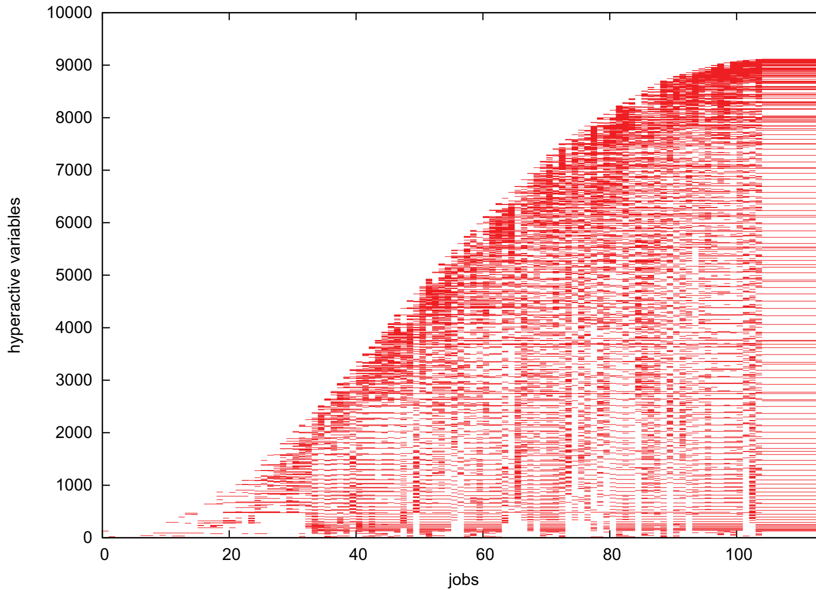


Figure 3.1. Hyperactive variable visualization for benchmark `bc57sensorsp2neg`.

variables and clauses, across jobs. The visualization of hyperactive variables proposed in Publication II illustrates how the activity of variables propagates across jobs.

3.1 The hyperactive variable visualization

Visualization of hyperactive variables, can provide some useful insights in behavior of incremental solvers using the VSIDS decision heuristic on instances from Bounded Model Checking (BMC) [BCCZ99]. BMC is described in Section 4.3 in more detail, but for now it suffices to understand BMC as a solver application which generates a sequence of jobs for an incremental solver.

To visualize the behavior of the solver we are interested in which variables are the most active, and especially in whether this activity remains high across several jobs. We consider a variable *hyperactive* if its activity is within the highest 2% of variables with non-zero activity. The *hyperactivity visualization* provides an illustration of the state of the hyperactive variables in the solver at a specific point in time. Examples of the hyperactivity visualization from Publication II are given in the Figures 3.1 and 3.2. The figures correspond to two benchmarks from the Hardware Model

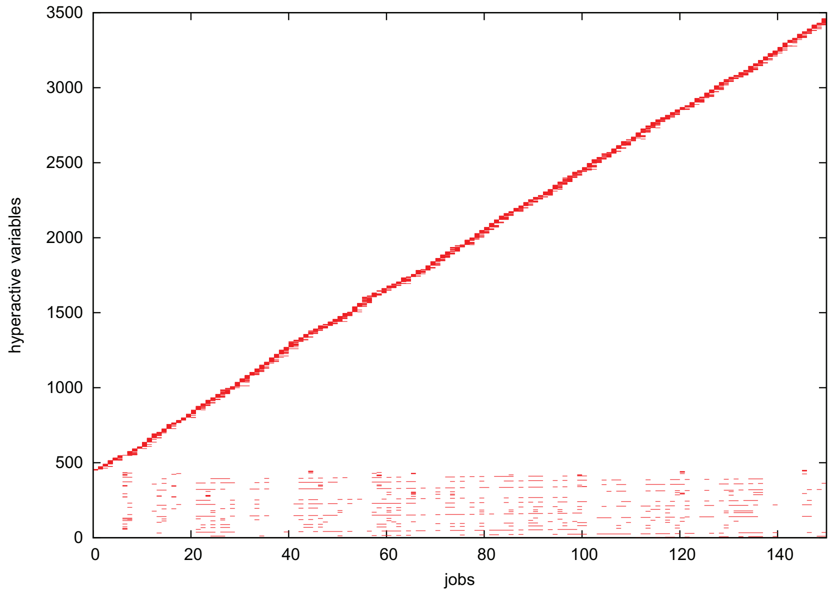


Figure 3.2. Hyperactive variable visualization for benchmark `eijk.S1238.S`.

Checking Competition 2007[2] (HWMCC'07), named `bc57sensorsp2neg` and `eijk.S1238.S`, respectively. For the work presented in [WNH09] these benchmarks were encoded as sequences of jobs for a SAT solver, represented in iCNF files, using the BMC encoding of [HJL05]. The data in the figures has been obtained by solving those two iCNF files.

All the variables that are hyperactive at least once are represented by an integer value on the y-axis of the graph. The variables are sorted on the y-axis by their index such that if we define $y(v)$ as the integer on the y-axis corresponding to the variable with index v then for any $v' > v$ we have $y(v') > y(v)$. If a variable v became hyperactive after solving job k, and lost its hyperactivity after solving job $k' > k$ then a horizontal line is drawn in the graph from job k to k' at the position $y(v)$ on the y-axis. In other words for all variables v and all job intervals $[k, k')$ on which v was hyperactive a line is drawn from $(k, y(v))$ to $(k', y(v))$. One may observe from the Figures 3.1 and 3.2 that both plots are empty in their top left corner. This means that variables with larger indices become active later. The reason for this is that the set of variables grows with every job, and each new variable is given an index larger than that of all old variables by the solver.

The hyperactive variable visualization can provide insight in the char-

[2] `http://fmv.jku.at/hwmcc07`

acteristics of the particular BMC instance that was being solved. The bottom right corner of Fig. 3.1 is almost completely filled by all the horizontal lines drawn on it. This means that many of the variables that become hyperactive stay hyperactive. As the sequence of jobs grows the solver keeps using variables that were already active before to derive new conflict clauses. In other words, the search behavior of the solver is *global* with respect to the set of all jobs. Another thing that may be noticed is that starting from job 104 all lines become horizontal, which means that the variable activities no longer change. For this particular benchmark the first 104 jobs (i.e. jobs 0 to 103) are unsatisfiable, after which all consecutive jobs are satisfiable. The horizontal lines in the figure illustrate that the solver has been able to extend the satisfying assignment for each job starting from job 104 into a satisfying assignment for the consecutive job, without reaching conflicts that caused the set of hyperactive variables to change. The easy extension of satisfying assignments to consecutive jobs is a property of BMC as a solver application. A satisfying assignment corresponds to a *counterexample*, and in practice we are not interested in extending such counterexamples over consecutive jobs. However, it illustrates that we can observe application specific properties using visualization of solver statistics.

The behavior illustrated for a different BMC problem in Fig. 3.2 is very different. With every job only a few variables are active. For this benchmark a counterexample does not exist, which implies that regardless of the length of the generated sequence all jobs in it are unsatisfiable. The visualization shows that the solver is able to prove the unsatisfiability of each job using only a very small number of recent variables. We say that the search behavior of the solver is *local* with respect to the last jobs in the sequence. When such behavior is observed one may expect the existence of a small *inductive invariant*, as we will explain in more detail in Chapter 4.

Note that the readability of the graphs, and in fact our ability to derive some behavioral insight from it, depends heavily on the continuous introduction of new variables for every job. This means that this visualization is not useful for applications in which the number of variables does not grow continuously with the number of jobs.

3.2 The clause involvement visualization

In this section we will propose a new type of visualization, called the *clause involvement visualization*. It provides an alternative illustration of the involvement of clauses in the solver's conflict clause derivation process. Examples are given in the Figures 3.3 to 3.6. Each figure consists of two separate plots with identical horizontal axes placed above each other. These plots illustrate the state of the solver at a specific point in time, i.e. after completing the solving of a specific job. The plots are filled with vertical bars, and the height of each bar corresponds to the number of clauses in a set. The clause sets correspond to specific jobs denoted on the horizontal axis. Unit clauses are considered as truth assignments to variables, and are thus not counted as elements of these sets. In each figure, the top plot corresponds to sets of problem clauses, whereas the bottom plot visualizes sets of learnt clauses.

The total height of a bar positioned in a problem clause plot (upper plot) represents the number of clauses that were added to the solver for that job, i.e. the height of the bar for job ϕ_i corresponds to $|\texttt{CLS}(\phi_i)|$. Hence, the total number of problem clauses ever introduced in the solver is the sum of the height of all the bars. In a learnt clause plot (lower plot) the height of a bar corresponds to the number of learnt clauses derived during the solving of the corresponding job, in other words, the number of conflicts the solver encountered solving that job. Each bar is divided into three partitions that represent the division of the set of clauses into three subsets, organized as follows:

$$\text{“Recently used”} \subseteq \text{“remains”} \subseteq \text{“total”}.$$

Note that due to the size of these sets not all three partitions are necessarily visible on each bar. The bottom partition of each bar is filled according to the style labeled "Recently used" in the legend. The height of this partition represents the number of clauses from this set that was used in a conflict clause derivation during the solving of the last job in the sequence.

Both problem- and learnt clauses may get deleted from the solver as a result of simplifications due to unit propagation. Moreover, learnt clauses may be removed as the solver attempts to keep the size of the learnt clause database under control. This clause deletion is visualized by the difference in height between the middle- and top partitions. On each bar, the height of the middle partition, labeled "SAT - remains" or "UNSAT - re-

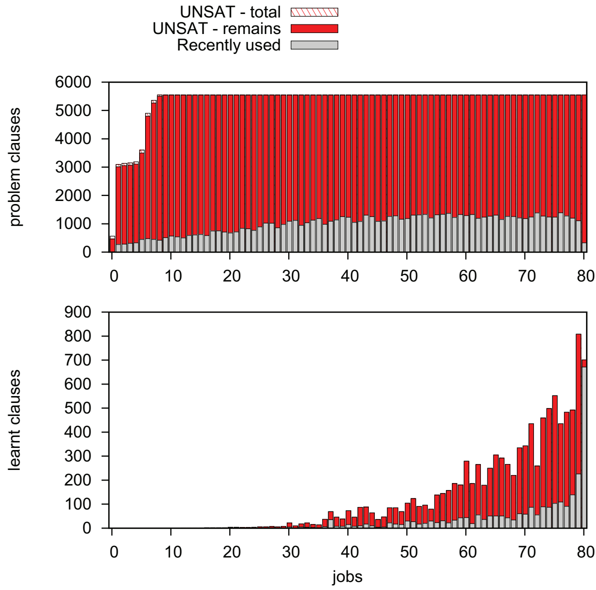


Figure 3.3. Clause involvement visualization, benchmark `bc57sensorsp2neg` and $k = 80$.

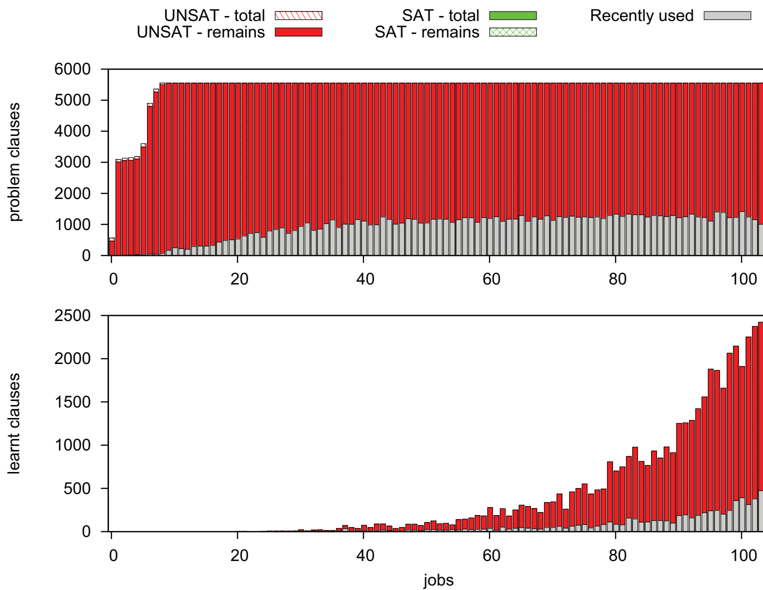


Figure 3.4. Clause involvement visualization, benchmark `bc57sensorsp2neg` and $k = 104$.

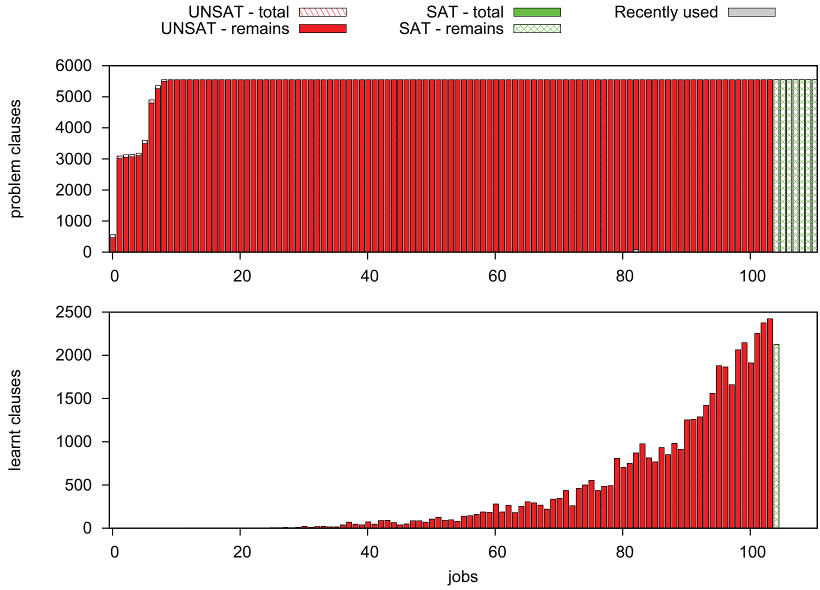


Figure 3.5. Clause involvement visualization, benchmark `bc57sensorsp2neg` and $k = 110$.

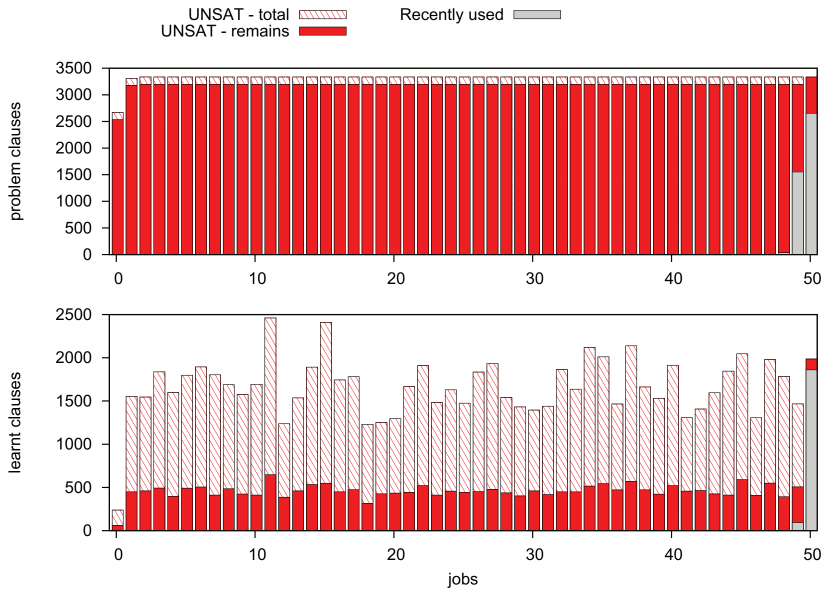


Figure 3.6. Clause involvement visualization, benchmark `eijk.S1238.S` and $k = 50$.

mains" in the legend, represents the number of clauses that still existed in the solver when the last job in the sequence was solved. The fill style of the middle- and top partitions depends on whether the corresponding job finished with result satisfiable or unsatisfiable.

Unlike the figures presented in Section 3.1 the data for the clause involvement figures in this chapter is not obtained by solving iCNF files, but by directly running AIGBMC on the original AIGER format benchmarks. The Figures 3.3 to 3.5 illustrate the state of the solver after solving three different prefixes of the sequence of jobs generated and solved by AIGBMC for HWMCC'07 benchmark `bc57sensorsp2neg`. Note that despite the different encoding the data visualized in the Figures 3.3 to 3.5 concern solving the same BMC problem as the one given in Fig. 3.1, and hence the first satisfiable job in this sequence is job 104.

In Fig. 3.3 the state of the solver just after finding job 80 unsatisfiable is illustrated. The clearly visible bottom partition on most of the bars illustrates that problem- and learnt clauses from many jobs are used in conflict clause derivations during the solving of job 80. The same behavior can be observed in Fig. 3.4, which illustrates the state of the solver just after finding job 104 satisfiable. In this way, these figures also visualize the global nature of the solver's search on this problem.

It was already observed in the discussion of Fig. 3.1 provided in Section 3.1, that once the solver finds a satisfying assignment for job 104 it can easily extend this assignment to a satisfying assignment for consecutive jobs. The same behavior can be seen in Fig. 3.5 which shows the state of the solver after solving job 110. One may observe from that figure that no new conflicts are encountered during the solving of the jobs 105 to 110. As a result, there are no clauses involved in conflict clause derivations during the solving of job 110, and thus the bottom partitions of all bars are empty. The conflict free jobs thus wiped out the history about the global search behavior of the earlier jobs from the picture. If this is desired, a clause involvement visualization that gives a more robust picture of the conflict history can be obtained by redefining what "Recently used" means. For example, it could be defined as a clause that was used during one of the last 1000 conflict clause derivations.

We already showed how global search behavior can be observed from the clause involvement visualization. Figure 3.6 visualizes the state of the solver after finding job 50 of HWMCC'07 benchmark `eijk.S1238.S` unsatisfiable using AIGBMC. Here, a very local search behavior is visible:

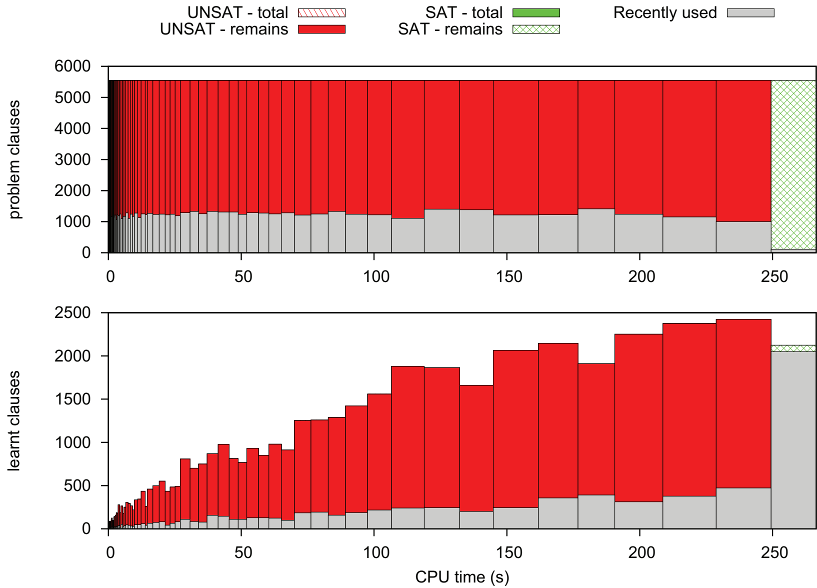


Figure 3.7. Timed clause involvement visualization, benchmark `bc57sensorsp2neg` and $k = 104$.

Only the most recently created clauses are used in conflict clause derivations during the solving of job 50. The clause involvement visualization can thus illustrate global or local search behavior for BMC problems, just like the hyperactive variable visualization. This is not surprising once one recalls that variable activity and recent conflict clause derivations are directly related: The more active a variable, the more it has been involved in recent conflict clause derivations. The advantage of the clause involvement visualization is, however, that it can also be used to illustrate the behavior of solvers in applications for which the set of variables does not continuously grow with every job.

To make observations about solver behavior, we typically used videos providing an animation of the clause involvement visualizations for consecutive jobs, rather than static pictures. We developed an extension of MINISAT which displays the clause involvement visualization while the solver is running, continuously updating it to match the state of the solver[3].

3.3 The timed clause involvement visualization

The clause involvement visualization provides clear pictures for solver applications in which the solving of each job requires a substantial amount of work, or at least generates a substantial number of conflicts. However, incremental solvers are also commonly used in applications where only a small fraction of the jobs are actually causing a non-negligible amount of work for the solver. In such cases, the clause involvement visualization will not provide readable pictures. A solution is to set the width of each bar based on the amount of CPU time spend solving the corresponding job. The result is the *timed clause involvement visualization*. An example is provided in Fig. 3.7, which is a timed version of Fig. 3.4.

[3]This extension of MINISAT, as well as examples of these animated visualizations, are available from: `http://www.siert.nl/thesis`

4. Model Checking

The most successful industrial application of SAT solvers is arguably in the area of formal verification. We will discuss some basic concepts relating to the verification of finite state systems. The discussion is meant to enable our discussion of SAT solver usage in formal verification, and it is by no means a complete overview of verification techniques. The provided solver usage discussion uses the visualizations proposed in Chapter 3, and can be seen as a major extension of the work performed in Publication II. It also relates directly to Publication V, which considers verification applications as important targets for parallelization.

Model checking [CGP01] is a popular formal verification technique. In model checking, one is given a formal model of a system and asked to prove or disprove a property of this model. The properties are often specified in a *temporal logic* [Pnu77], and *Kripke structures* are a commonly used formalism for finite state systems.

Definition 4.1 (Kripke structure). *A Kripke structure is a tuple* $M = (S, I, T, L)$ *where* S *is a finite set of states,* $I \subseteq S$ *is the set of initial states,* $T \subseteq S \times S$ *is the transition relation, and* $L : S \rightarrow 2^{AP}$ *is a function that labels each state with the subset of the atomic propositions* AP *that hold in that state. Moreover, the transition relation* T *is such that for all* $s \in S$ *there exists some* $s' \in S$ *such that* $(s, s') \in T$*.*

Example 4.2. *We will define a Kripke structure that represents a simple traffic light controller, which has only one possible operation sequence that is depicted in Fig. 4.1. The states of the Kripke structure are labelled with atomic propositions from* $AP = \{red, orange, green\}$*, representing the lights that are lit in that state of the traffic light controller. The Kripke*

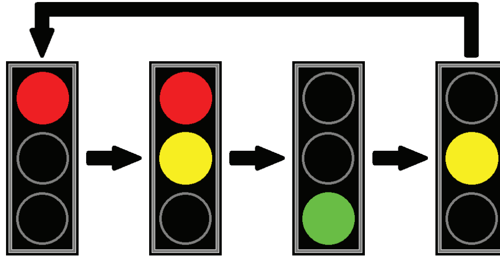


Figure 4.1. Traffic light operation sequence.

structure $M = (S, I, T, L)$ *is completely defined as follows:*

$$\begin{aligned} \text{Let} \quad & S &= \{s_0, s_1, s_2, s_3\}, \\ \text{and} \quad & I &= \{s_0\}, \\ \text{and} \quad & T &= \{(s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_0)\}, \\ \text{and} \quad & L(s_0) &= \{red\}, \\ \text{and} \quad & L(s_1) &= \{red, orange\}, \\ \text{and} \quad & L(s_2) &= \{green\}, \\ \text{and} \quad & L(s_3) &= \{orange\}. \end{aligned}$$

Kripke structures can be represented as graphs, and for our example Kripke structure this representation is given in Fig. 4.2. The states of a Kripke structure can be given a bit-vector representation. As we have four states we need at least two bits to represent the state of this Kripke structure. For this example we chose a minimal representation that operates as a two-bit binary counter:

state	*bit-vector representation*	*symbolic representation*
s_0	00	$s_0^{\#}(x_0, x_1) = \neg x_1 \wedge \neg x_0$
s_1	01	$s_1^{\#}(x_0, x_1) = \neg x_1 \wedge x_0$
s_2	10	$s_2^{\#}(x_0, x_1) = x_1 \wedge \neg x_0$
s_3	11	$s_3^{\#}(x_0, x_1) = x_1 \wedge x_0$

Now that we have shown by example how the states of a Kripke structure can be given a bit vector representation, we can encode the states and the transition relation symbolically, using propositional logic formulas. For Example 4.2 the symbolic representations $s^{\#}$ for all states s of the Kripke structure are given alongside the bit-vector representations. Observe that these are simply conjunctions of literals that force the *state variables* $x_0, \cdots, x_{n-1}$ to attain values representing the bit-vector representation of the state.

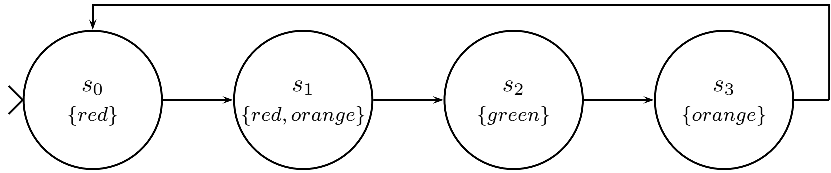


Figure 4.2. The Kripke structure defined in Example 4.2 as a graph.

Definition 4.3 (Symbolically representing a set of states). *For any set of states $X \subseteq S$ we will denote its symbolic representation as $X^{\#}$, a formula over a set of state variables $\{x_0, \cdots, x_{n-1}\}$, such that:*

If and only if $s \in X$ then

$$\begin{array}{ll} s^{\#}(x_0, \cdots, x_{n-1}) & \wedge \\ X^{\#}(x_0, \cdots, x_{n-1}) & \textit{is satisfiable.} \end{array}$$

Definition 4.4 (Symbolic transition relation). *The transition relation T of a Kripke structure is a set of pairs of states. It can therefore be represented symbolically by a formula $T^{\#}$ over two sets of state variables, such that:*

If and only if $(s, s') \in T$ then

$$\begin{array}{ll} s^{\#}(x_0, \cdots, x_{n-1}) & \wedge \\ T^{\#}(x_0, \cdots, x_{n-1},\ x'_0, \cdots, x'_{n-1}) & \wedge \\ s'^{\#}(x'_0, \cdots, x'_{n-1}) & \textit{is satisfiable.} \end{array}$$

Model checking using symbolic representations of systems is called *symbolic model checking* [McM93]. Originally, manipulations on *Binary Decision Diagrams* (BDDs) were used for symbolic model checking, but in recent years SAT based techniques have become very popular [BCCZ99, SSS00, McM03, SB11].

Example 4.5. *To complete the symbolic representation defined in Example 4.2, let us first define for each of the three colors the set of states in which that color is included in the labelling, e.g.* $Red = \{s \mid red \in L(s)\} = \{s_0, s_1\}$. *The symbolic representation can now be completed by defining*

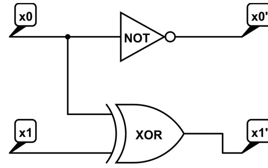


Figure 4.3. Transition relation of the traffic light controller as a circuit.

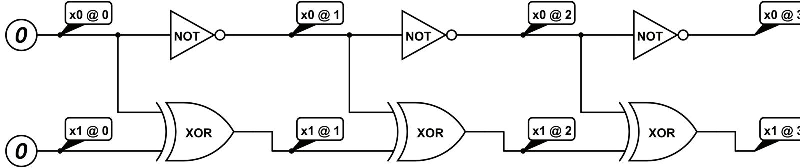


Figure 4.4. Unrolling of the transition relation of Fig. 4.3.

the symbolic representation of all sets involved:

$$
\begin{array}{llll}
\textit{Let} & I^{\#}(x_1, x_0) & = s_0^{\#}(x_1, x_0) & = \neg x_1 \wedge \neg x_0, \\
\textit{and} & Red^{\#}(x_1, x_0) & = s_0^{\#}(x_1, x_0) \vee s_1^{\#}(x_1, x_0) & = \neg x_1, \\
\textit{and} & Orange^{\#}(x_1, x_0) & = s_1^{\#}(x_1, x_0) \vee s_3^{\#}(x_1, x_0) & = x_0, \\
\textit{and} & Green^{\#}(x_1, x_0) & = s_2^{\#}(x_1, x_0) & = x_1 \wedge \neg x_0, \\
\textit{and} & T^{\#}(x_1, x_0, x_1', x_0') & = (\, s_0^{\#}(x_1, x_0) \wedge s_1^{\#}(x_1', x_0')\,) & \vee \\
 & & (\, s_1^{\#}(x_1, x_0) \wedge s_2^{\#}(x_1', x_0')\,) & \vee \\
 & & (\, s_2^{\#}(x_1, x_0) \wedge s_3^{\#}(x_1', x_0')\,) & \vee \\
 & & (\, s_3^{\#}(x_1, x_0) \wedge s_0^{\#}(x_1', x_0')\,) & = \\
 & & (x_0' \leftrightarrow \neg x_0) \wedge (x_1' \leftrightarrow x_0 \texttt{ xor } x_1). &
\end{array}
$$

4.1 Circuits

In Fig. 4.3[1] the symbolically represented transition relation $T^{\#}$, defined in Example 4.5, is presented as a Boolean circuit. To evaluate the state of the system after k timepoints we *unroll* the transition relation. This means that we simply create k cascaded copies of the transition relation circuit, constraining the state bits of the first copy by a circuit representing the initial state formula. The unrolling of the transition relation of our example is shown in Fig. 4.4, where the circles containing the value 0 on the left represent the initial state constraint, which forces the state bits at timepoint zero to **false**. Our example system has only exactly one

[1]Figures 4.3 to 4.5 were made using the online circuit editor CircuitLab: `http://www.circuitlab.com`

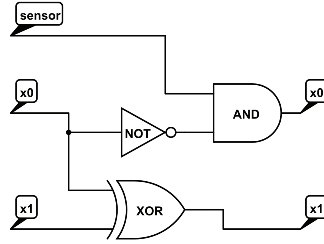


Figure 4.5. Transition relation of a traffic light controller with sensor input.

initial state, and each state has exactly one successor. In other words, this system has no inputs. As a result, for any unrolling of this circuit the value of all variables follows directly from the unrolled transition relation. We will now extend our example to include an input signal.

Example 4.6. *Assume that our traffic light controller from Example 4.2 is extended with an external sensor that detects whether a car arrives at a traffic light somewhere on the crossing. Let us modify the controller such that if the controller is in state s_0 (red light only) or s_2 (green light) then it will remain in that state until a car passes the sensor. This adds non-determinism in the modified transition relation T of the Kripke structure, where the states s_0 and s_2 now each have two possible successor states.*

$$T = \{ (s_0, s_0), (s_0, s_1), (s_1, s_2), (s_2, s_2), (s_2, s_3), (s_3, s_0) \}.$$

*Let us denote by i_{sensor} the state of the sensor input signal, where the input **true** means that a car is passing the sensor. The new transition relation must enforce $x'_0 = \neg x_0 \wedge i_{sensor}$. This transition relation is given as a circuit in Fig. 4.5. Note that if we unroll this circuit, then there will be a "loose input wire" for each timepoint, that corresponds to the state of the input sensor at that timepoint.*

4.2 Properties

Once we have decided on a formal model for our system we may want to verify some properties of this model. Such properties are typically formalized in a temporal logic [Pnu77], but in this document we will use only informal descriptions. Two basic types of properties exist. The first type of properties are the *safety properties*. A safety property is a property that is such that if it does not hold, then there exists a finite execution path of the system that violates the property. Properties of this type in-

tuitively state that "something bad will never happen". An example of a safety property for a traffic light controller would be "the green and the red lamp are never lit at the same time". If the property holds then any state in which the green and red lamp are both lit must be unreachable from the initial states of the Kripke structure representing the traffic light controller. Otherwise, the *counterexample*, e.g. the execution path that illustrates that the property does not hold, is a finite path that starts in an initial state and ends in a state in which the green and red lamp are both lit.

The second type of properties are called *liveness properties*. Such properties, intuitively speaking, state that "something good will eventually happen". For example, for a traffic light controller a natural liveness property would be "for all execution paths of the system, if the red lamp is on, then there is a state in the future in which the green lamp is on". The counterexample against a liveness property is always an infinite execution path. This is because this type of property requires some event to happen in the future, but it places no bound on how far in the future this event may happen. As we only consider finite state systems any infinite execution must contain a loop. Hence, a counterexample against the property is an execution path in which at some state the red light is on, after which the execution path eventually reaches a loop without the green light ever being lit along this path.

Note that the traffic light controller of Example 4.6 does not satisfy the stated liveness property. A counterexample against the property is the infinite execution that starts in the initial state s_0, and remains there forever. In terms of cars and traffic lights this corresponds to a case where the light is red and no car ever passes, hence the light will never turn green. Although this is a valid counterexample against the liveness property, it is probably not considered as a bug in the controller[2]. To avoid this type of counterexamples we may add a *fairness constraint* to the problem description for our model checker. A fairness constraint is a logic constraint that must be satisfied infinitely often in every infinite execution of the system that the model checker may consider. A natural fairness constraint for our traffic light controller is "a car passes the sensor", as we are only interested in infinite executions in which cars keep arriving.

[2]Real-life traffic light controllers are obviously much more complex. Built-in timers may ensure fairness in such systems.

4.3 Bounded Model Checking

Bounded Model Checking (BMC) [BCCZ99] was the first popular SAT based model checking technique, and it remains popular today. In BMC we restrict ourselves to testing the existence of a counterexample of a bounded length k using a SAT solver. First, we must represent the initialized unrolling of the transition relation over k-timepoints in the solver. This is straightforward, one simply creates $k+1$ copies of the state bits, then constrains the first copy using the initial state constraint, and every consecutive copy using the transition relation. Any satisfying assignment to such a formula corresponds directly to an execution path that is valid in the system model.

Example 4.7. *An unrolling over three timepoints of the circuit in Fig. 4.5 corresponds to the following formula. Each of the four copies of the states bits x_0 and x_1 correspond to one of the timepoints 0 to 3, as follows:*

$$\begin{array}{ll} I^{\#}(x_{1@0}, x_{0@0}) & \wedge \\ T^{\#}(x_{1@0}, x_{0@0}, x_{1@1}, x_{0@1}) & \wedge \\ T^{\#}(x_{1@1}, x_{0@1}, x_{1@2}, x_{0@2}) & \wedge \\ T^{\#}(x_{1@2}, x_{0@2}, x_{1@3}, x_{0@3}). & \end{array}$$

This formula expands to:

$$\begin{array}{llll} (& \neg x_{0@0} \wedge \neg x_{1@0} &) & \wedge \\ (& x_{0@1} \leftrightarrow \neg x_{0@0} \wedge i_{sensor@0} &) & \wedge \\ (& x_{1@1} \leftrightarrow x_{0@0} \texttt{ xor } x_{1@0} &) & \wedge \\ (& x_{0@2} \leftrightarrow \neg x_{0@1} \wedge i_{sensor@1} &) & \wedge \\ (& x_{1@2} \leftrightarrow x_{0@1} \texttt{ xor } x_{1@1} &) & \wedge \\ (& x_{0@3} \leftrightarrow \neg x_{0@2} \wedge i_{sensor@2} &) & \wedge \\ (& x_{1@3} \leftrightarrow x_{0@2} \texttt{ xor } x_{1@2} &). & \end{array}$$

To test whether a path of length k exists that violates a safety property, we simply add a constraint that states that the safety property P is violated at timepoint k. Clearly, this formula will be satisfiable if and only if an execution violating the property of length k exists.

Example 4.8. *Let us extend the set of constraints defined in Example 4.7 to require an execution path that violates the safety property "the green light is off" at timepoint 3. The property can be represented symbolically by $P^{\#}(x_1, x_0) = \neg Green^{\#}(x_1, x_0)$. We add the violation of this safety prop-*

erty at timepoint 3 *as a constraint, i.e. we add the following constraint:*

$$\neg P^{\#}(x_{1@3}, x_{0@3}) \quad = \quad \neg(\,\neg Green^{\#}(x_1@3, x_0@3)\,) \quad = \quad x_{1@3} \wedge \neg x_{0@3}.$$

Adding this constraint to the constraints of Example 4.7 yields a satisfiable formula, because an execution in which the green light is lit at timepoint 3 *exists. An example of such an execution is the path* $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_2$*, which is the path that the system will traverse if* $i_{sensor@0}$ *is assigned* ***true****, and* $i_{sensor@2}$ *is assigned* ***false****.*

Typically BMC algorithms start from bound $k = 0$, and as long as the solver returns unsatisfiable for the encoded set of constraints this bound is increased by one. The constraint that P is falsified in the last state is the only constraint that depends on the value of k, hence this constraint must be removed when the bound is increased[3]. If we want to be able to start from k larger than zero, or increment k by more than one at a time, we may want to encode our formula such that it is satisfiable if a counterexample of length at most k, rather than exactly k, exists. As we illustrated in Publication V this type of encoding facilitates efficient parallelization of BMC algorithms. The first method to obtain this type of semantics is to modify the encoding. This can be done by including the definition of the violated safety property for every timepoint, and then adding a constraint that forces the disjunction of all of those to hold. Observe that this disjunction will still be a constraint that must be removed when k is increased.

The second method, which we used in Publication V, is to use the standard "exactly k" encoding, but to modify the system and state invariant such that once the modified system reaches a bad state it stays in a bad state forever. For this purpose we created the tool AIGMOD[4].

Definition 4.9 (Operation of AIGMOD). *For a system with* n *state bits, with good states represented symbolically by* $P^{\#}$*, initial states by* $I^{\#}$*, and the transition relation by* $T^{\#}$*, let the modified system with* $n+1$ *state bits*

[3]Recall that clause removal for a solver using the assumptions interface can be simulated by using selector variables.
[4]Available from: http://www.siert.nl/thesis

be defined as follows:

$$\begin{array}{lll} \textit{Let} \;\; P^{\#}_{mod}(x_0,\cdots,x_n) & = & \neg x_n, \\ \textit{and} \;\; I^{\#}_{mod}(x_0,\cdots,x_n) & = & I^{\#}(x_0,\cdots,x_{n-1}) \wedge \\ & & (\, x_n \leftrightarrow \neg P^{\#}(x_0,\cdots,x_{n-1})\,), \\ \textit{and} \;\; T^{\#}_{mod}(x_0,\cdots,x_n,x'_0,\cdots,x'_n) & = & T^{\#}(x_0,\cdots,x_{n-1},x'_0,\cdots,x'_{n-1}) \wedge \\ & & (\, x'_n \leftrightarrow (\, x_n \vee \neg P^{\#}(x'_0,\cdots,x'_{n-1})\,)\,). \end{array}$$

*Observe that the extra state bit x_n is assigned **true** at any timepoint in which the original system is in a bad state. Moreover, whenever x_n is assigned **true** it remains **true** forever. Hence, the assignment of **true** to x_n represents that the original system has traversed a bad state.*

Bounded Model Checking can also be used for finding counterexamples against liveness properties, using e.g. its original encoding [BCCZ99], or the more advanced the encoding of [HJL05]. The basis is the same k times unrolled transition relation and initial state formula. However, the way in which violation of the property is encoded is different. Most importantly, the constraints must enforce that any satisfying execution path contains a loop, as this is the only way to represent an infinite path using a sequence of k states.

4.4 Completeness

Bounded model checking is a powerful technique to find counterexamples. In theory it can also be used to prove properties, but in practice this ability is very weak [BCCZ99, BHJ$^+$06, Bie09]. In this section we will discuss some techniques aimed at complete model checking using SAT solvers. In this discussion we will limit ourselves to the verification of simple safety properties, i.e. *invariants*.

The *diameter* of a Kripke structure is the length of the longest shortest path between any two states in the state set S [BCCZ99]. If we prove that no execution violating the invariant exists for length equal to the diameter of the Kripke structure, then we have proven that no counterexample exists at all. This is because if there exists a counterexample, then the shortest counterexample uses a path from the initial state to the bad state that is at most as long as the diameter. Unfortunately, computing the diameter of a Kripke structure that is represented symbolically is difficult [Bie09]. However, any upper bound on the diameter can be used as a *completeness threshold* (CT) for BMC [BCCZ99].

A trivial upper bound on the diameter of the Kripke structure is the number of states in it, but for any real-life system this is far too large to be useful as a CT. The length of the longest loop-free path starting from any initial state is another upper bound on the diameter of the system. It is also known as the *forward radius*. Observe that any shortest counterexample will start in an initial state, and traverse only good states until it reaches the final bad state. Hence, the length of the longest loop-free path through good states that ends in a bad state is also an upper bound on the diameter of the system. This completeness threshold is known as the *backward radius*, and it is used by a complete SAT based model checking technique called *k-induction* or *temporal induction* [SSS00, ES03b].

Just like for normal BMC, k-induction operates using a growing bound k. To perform k-induction for bound k, we first perform BMC for that bound. This is called the *base step*. If a counterexample is found then the safety property is violated. If no counterexample is found, we ask a SAT solver whether there exists a path through good states that ends in a bad state of length k This is called the *induction step*. If no such path exist, then k is at least as large as the backward radius, and thus no counterexample against the property can exist. This technique is not complete, as there may be a path leading to a bad state with an infinite prefix of good states. In order to make k-induction a complete model checking technique we need to add constraints that force the backward path to be a loop-free path. The simplest way to achieve this is to forbid every pair of state variable copies to assume identical values, however this leads to a number of extra constraints that is quadratic in k. Although solutions requiring fewer constraints exists (e.g. [KS03]) such constraint sets tend to be harder to solve. Here, incremental solvers can play an invaluable role. It was observed in [ES03b] that we can simply leave out all of the loop-free path constraints, request the solver for a path of length k leading to the bad state, and only if the path contains a loop encode a constraint forbidding that specific loop before trying again. Typically, only a few of such *refinement* iterations are required before a loop-free path is found. Solving such an incomplete set of constraints (known as an *abstraction* of a problem), and adding constraints only as long as we do not obtain an answer to the original problem is known as *Counter Example Guided Abstraction Refinement* (CEGAR).

Other techniques for complete SAT-based symbolic model checking exists. Amongst the most successful are techniques based on *interpolation*

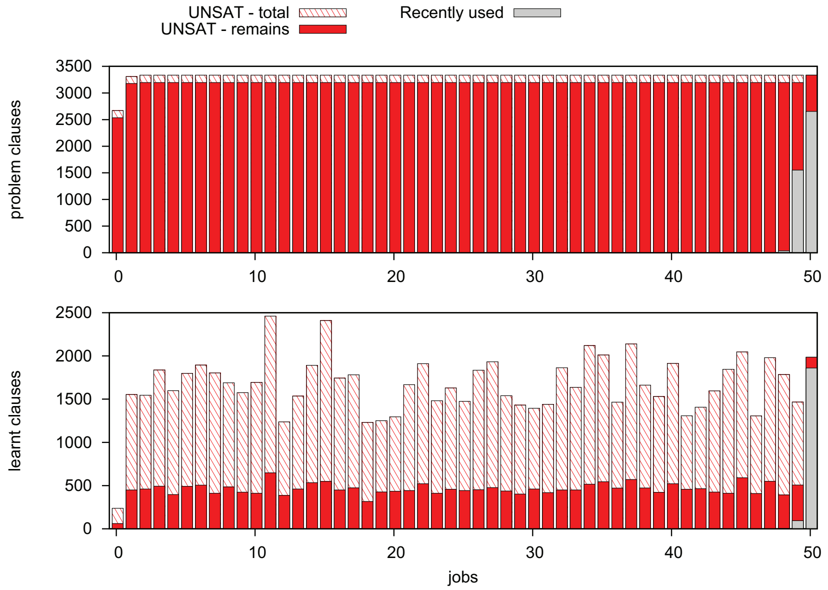


Figure 4.6. Clause involvement visualization for benchmark `eijk.S1238.S` and $k = 50$.

[Cra57], first proposed in [McM03]. A recent and powerful SAT-based symbolic model checking technique is the IC3 algorithm [Bra11], which will be discussed in the Sections 4.6 and 4.7.

4.5 Analyzing the solver usage

In Publication II we discussed the observation that there exist job sequences from BMC, for which it takes less time to solve all jobs from zero up to any arbitrary k using the solver's incremental features, than it takes to solve only job k without those features. We observed that all benchmarks with this property expose a very local behavior in the incremental solver, in the sense that to prove the last job unsatisfiable conflict clauses are generated using only the immediately preceding jobs. One example of such a job sequence used in Publication II was corresponding to the encoding of HWMCC'07 benchmark[5] `eijk.S1238.S`. The clause involvement visualization resulting from solving that benchmark directly using AIGBMC up to the rather arbitrarily chosen bound $k = 50$ is given in Fig. 4.6. Note that this same figure already appeared in Chapter 3 as Fig. 3.6. The behavior exposed by this benchmark both in terms of locality and in terms of run time behavior is extreme, but not uncommon. It is caused by the existence of a small *inductive invariant* in the benchmark,

[5] `http://fmv.jku.at/hwmcc07`

in other words a small proof by induction of the property reflected in the benchmark. To clarify this, let us consider a simple example system.

Example 4.10. *Consider a system that consists of a* 3*-bit counter* x*, and a* 4*-bit counter* y*. Let the only initial state constraint be that the counter* y *has value zero. Note that this system has* $2^7 = 128$ *states, of which* $2^3 = 8$ *are initial states. The transition relation is defined as follows: If the current state value of* x *is its maximum value* 7*, then the next state value of both counters is zero. Otherwise, the next state value of both counters is their respective current state value incremented by one.*

Now let us verify the property that the value of counter y *always remains smaller than* 8*. In other words, the most significant bit of the counter* y *is assigned* ***false*** *throughout any initialized execution of the system. Observe that this property holds, as* y *counts the number of times* x *is incremented before it reaches zero, and* x *is a* 3*-bit counter.*

Any state where $y \geq 8$ *is a bad state. Consider a state in which counter* x *has value* x_b *and counter* y *has value* y_b *with* $y_b \geq 8$*. The longest path through good states leading to this bad state is of length exactly* x_b*, because in the first state of that path the counter* x *will have value zero and counter* y *will have a value larger than zero. The transition relation does not allow any transitions into such a state. The backward radius of this model is* 8*, because this is the largest possible value for* x_b*.*

This system and its property are given as an SMV [McM93] model in Appendix A. The SMV model was converted to an AIGER file using a tool called SMVTOAIG[6]*. The clause involvement figures where subsequently obtained by running* AIGBMC*. In Fig. 4.7 the inductive behavior can be easily observed, as all conflict clauses are derived from the last set of jobs. The figure shows the behavior for* $k = 30$*, but any choice of* k *that is substantially larger than* 8*, say starting from* 12*, would be suitable to illustrate the behavior we discuss here. Observe that the last* 8 *jobs are the most heavily used in the conflict clause derivations that lead to the unsatisfiable result for job* 30*. Some problem clauses from jobs* $20 - 22$ *were also used, but this is simply a result of the solver's heuristically guided search process. Note that a proof using only the last* 8 *jobs for any* $k \geq 8$ *is guaranteed to exist for this example system. The key to this fact is the incremental usage of the solver: Whenever it starts to solve job* k*, it has already proven that no counterexample of length* $k-1$ *exists, i.e. there are no bad states amongst the first* $k-1$ *states. The transition relation*

[6]This tool is included in the AIGER 1.9 toolset: `http://fmv.jku.at/aiger`

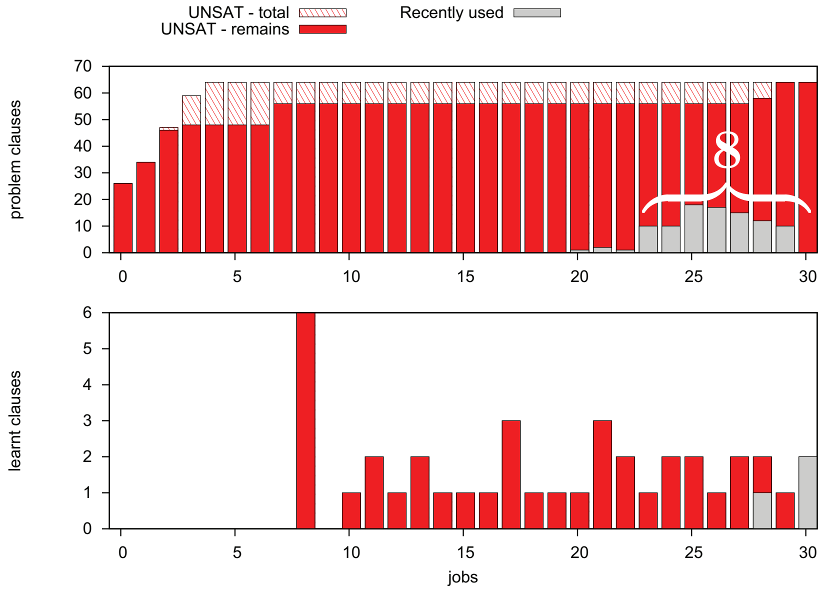


Figure 4.7. Clause involvement visualization for the BMC problem described in Example 4.10 solved using AIGBMC up to $k = 30$.

over 8 *states as represented by the problem clauses of the last* 8 *jobs, in conjunction with the constraint that all states are good except the last which is bad, is unsatisfiable, because the backward radius of the system is* 8*.*

For benchmark `eijk.S1238.S`, the extremely local behavior observable in Fig. 4.6 can also be explained by its backward radius, which is only 2. This means that the property in this benchmark can easily be proven using k-induction. Studying the behavior of BMC without completeness checks on this benchmark is nevertheless interesting from the point of view of studying incremental solver behavior.

As mentioned before, for the encoding of this benchmark used in Publication II, solving all jobs sequentially up to any k can be achieved faster than solving just job k. The direct encoding using AIGBMC does not have this property, it instead generates sequences for which each of the individual jobs is very easy, also for large k. However, the encoding used in Publication II has the property that the satisfiability of job k corresponds to the existence of a counterexample of length *at most* k. We can enforce this same semantics for each job generated by AIGBMC by first applying AIGMOD to the benchmark, in other words by modifying the benchmark according to Definition 4.9. Note that in general this type of change has several practical benefits, most notably allowing efficient parallelization,

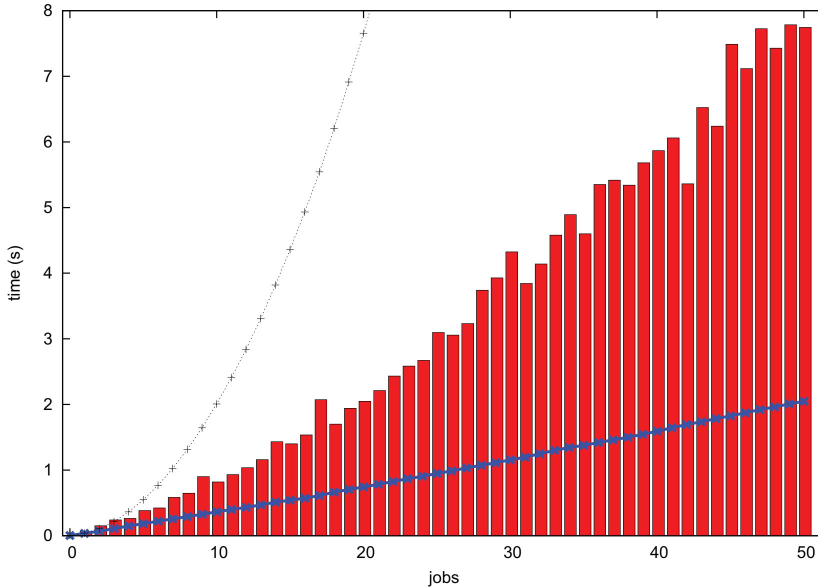


Figure 4.8. Run time behavior for benchmark `eijk.S1238.S` after manipulation with AIGMOD.

as discussed in Publication V. The change has no effect whatsoever on the behavior of AIGBMC when it uses its solver in the conventional incremental way, and thus it will yield a clause involvement visualization identical to Fig. 4.6. This is because whenever the incremental solver proceeds to job k it has already proven that there exist no counterexample of length $k-1$, and hence the modification by AIGMOD does not lead to any extra work for the incremental solver. However, solving the jobs generated by AIGBMC independently becomes much harder, as the solver must prove that no bad state is reached at any timepoint. As a result, we obtain the run time behavior also observed in Publication II, which we illustrate in Fig. 4.8. The height of a bar in this figure denotes the amount of time spend solving the corresponding job independently, in other words without using any incremental features. This data was obtained from 50 independent runs of AIGBMC which, as discussed in Section 2.3, has been modified to allow solving only a single job at a time. The thick curve illustrates the behavior of AIGBMC using it in the conventional incremental fashion, reporting its total run time each time it proceeded to the next formula in the sequence. The dotted curve is meant to further emphasize the poor performance of non-incremental solving, by illustrating the cumulative run time of solving all jobs sequentially and independently. Note that the superior performance of solving the whole sequence incrementally just to

achieve solving the last job is the result of guiding the solver in a way that is natural for the problem that it is solving.

By illustrating how we can observe properties of BMC problems in the clause involvement visualization, we hope to have already given some insight into how visualization can play a role in understanding behavior of solver applications. In the previously discussed case, we knew that a small inductive invariant existed and used this as an argument to explain the observations. However, such observations may be used in the opposite direction. If a solver consistently derives conflicts from a small number of recent jobs in a BMC problem sequence, then the existence of an inductive invariant is likely. Hence, if this observation is made, either manually or automatically, it could be used to direct effort towards finding this inductive invariant.

Although all kinds of properties of the problem and its encoding may be observed from the clause involvement visualization, one should not forget that it represents a snapshot of the involvement in clause derivations during the solving of one particular job. To make observations about solver behavior, it is preferable to use animated visualizations, i.e. videos of clause involvement visualizations for consecutive jobs, rather than static pictures. We give an example in which looking at more than one picture can avoid drawing an incorrect conclusion[7]. Looking at Fig. 4.9, the clause involvement visualization for HWMCC'11 benchmark[8] `pdtvisgigamax2` and $k = 57$, one may expect that this is a model checking problem with a small backward radius. There exist other individual values of k, both smaller and larger than 57, for which one could be easily tempted to draw the same conclusion. However, the visualization for $k = 58$ given in Fig. 4.10 tells a different story, as the solver used problem clauses from all jobs in conflict clause derivations for job 58. Using other model checking algorithms we have been able to determine that the backward radius of this system is at least 50, and possibly much larger. Nevertheless, when considering initialized paths of this system the incrementally generated proofs of unsatisfiability are apparently small for many k. This may still provide useful insight, and could still be used as guidance towards finding an inductive invariant. In this particular case a small inductive invariant does exist, and a representation of it as a 7 clause CNF formula can

[7]Clause involvement videos for the benchmarks discussed in this Section are available from: `http://www.siert.nl/thesis`

[8]`http://fmv.jku.at/hwmcc11`

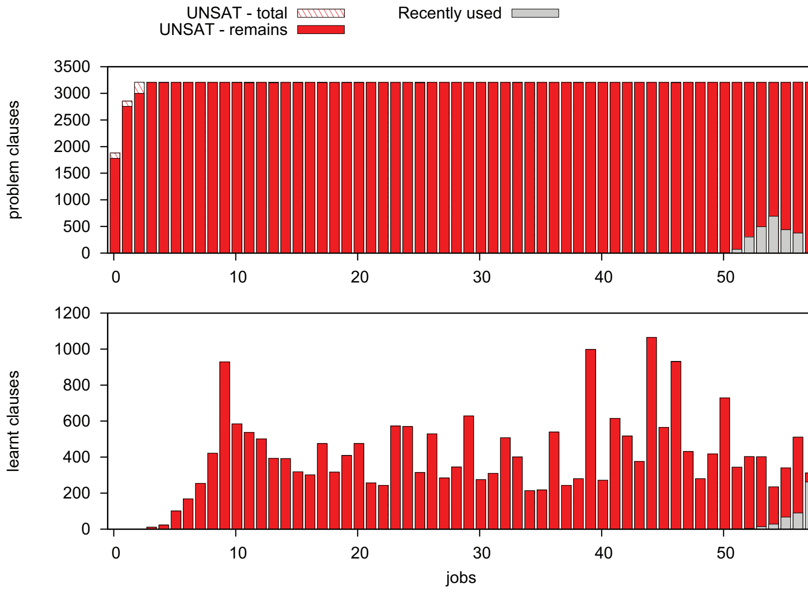


Figure 4.9. Clause involvement for HWMCC'11 benchmark `pdtvisgigamax2` and $k = 57$.

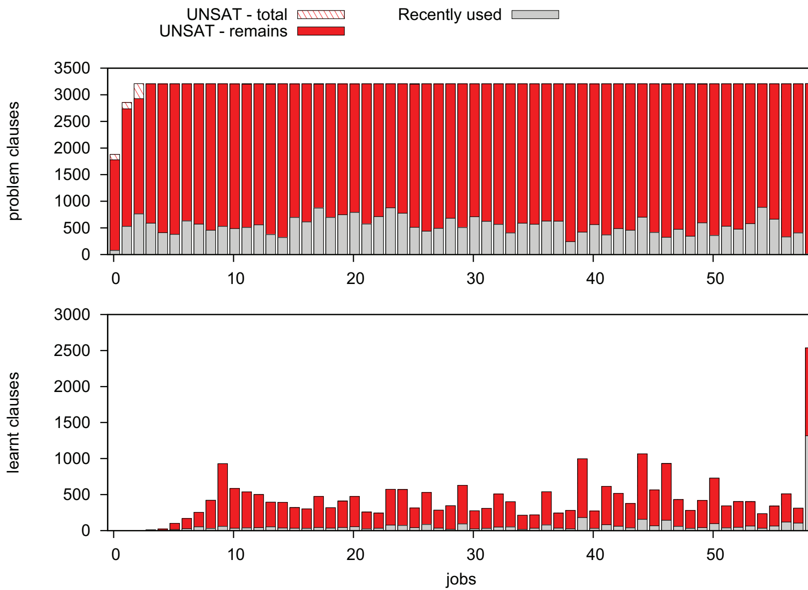


Figure 4.10. Clause involvement for HWMCC'11 benchmark `pdtvisgigamax2` and $k = 58$.

be found in a fraction of a second by TREBUCHET, a tool that will be discussed in the next section.

4.6 IC3 and PDR

A powerful and novel model checking algorithm called IC3 was recently introduced[9] [Bra11]. It currently represents the state-of-the-art in complete symbolic model checking algorithms. In [EMB11] an alternative description of IC3 is given, which is named *Property Driven Reachability* (PDR). There are some technical differences between IC3 and PDR. For example, IC3 requires testing that no counterexamples of length zero or one exist before starting the algorithm, whereas PDR does not require such special case handling. The work on PDR moreover advocates the use of *ternary simulation* to improve constraint strengthening steps used by the IC3 algorithm. On a conceptual level both algorithms are nevertheless identical.

We provide a basic high-level description of the algorithm, which is directly based on the explanation found in [KJN12]. A proof of a property P generated by IC3 is a formula $Proof^{\#}$ that symbolically represents a state set $Proof$ for which all of the following properties hold:

(a) All initial states are included in $Proof$, i.e. $I \subseteq Proof$.

(b) If $s \in Proof$ and $(s, s') \in T$ then $s' \in Proof$.

(c) P holds for all states in $Proof$, i.e. $Proof \subseteq P$.

$Proof$ is an over-approximation of the set of states reachable from the initial state. It forms an inductive proof that P holds in every initialized execution of the system, where property (a) is the base case and (b) is the induction step. To create such a proof, the IC3 algorithm builds a sequence of formulas $F_0^{\#} \cdots F_k^{\#}$. Each of these symbolically represents a state set F_i, called a *frame*, satisfying all of the following properties:

(i) $I \subseteq F_0$.

(ii) $F_i \subseteq F_{i-1}$ for $i > 0$.

(iii) P holds for all states in F_i.

(iv) If $s \in F_{i-1}$ and $(s, s') \in T$ then $s' \in F_i$ for $i > 0$.

[9]IC3 stands for *Incremental Construction of Inductive Clauses for Indubitable Correctness*.

The basic strategy employed by the IC3 algorithm is to add constraints to the frames in a way that maintains the properties (i)-(iv) until for all $s \in F_k$ and $(s, s') \in T$ it holds that P holds in s'. In other words, it refines the over-approximation of the reachable set of states until all successor states of the states in the last frame satisfy the property P. Once this happens, a new frame F_{k+1} such that $F^{\#}_{k+1} = P^{\#}$ is created, and k is incremented. The algorithm terminates once $F^{\#}_i = F^{\#}_{i+1}$ for some i and then provides $F^{\#}_i$ as the proof. If this happens the frame properties (i) and (ii) imply that the proof satisfies property (a). The frame property (iv) and the termination condition $F^{\#}_i = F_{i+1}$ imply that (b) holds, and property (iii) implies property (c) of the proof.

IC3 uses a SAT solver to ensure that the conditions (i)-(iv) are maintained throughout its execution. There are two types of solver calls. The first type of calls are those performed in the outer loop of the algorithm, to check whether there is a state not satisfying property P that is reachable from a state in the last frame F_k. This is a formula of the form displayed in the left column Table 4.1. If this formula is unsatisfiable a new frame is created. If, however, the formula is satisfiable, then there exists a state in the last frame from which a bad state is reachable. The algorithm now enters a loop that attempts to *block* this bad state, in other words, a loop that tries to improve the reachable state set approximations provided by the frames in such a way that properties (i)-(iv) are maintained. This is achieved by repeatedly calling the SAT solver to check the existence of a predecessor $s \in F_{i-1}$ that leads to a state $s_b \in F_i$ that must be blocked, where $(s, s_b) \in T$ and $s \neq s_b$. This is represented by formulas of the second type, displayed in the right column of Table 4.1. If the property P does not hold then the algorithm will fail to block some path to a bad state, which proves the existence of a counterexample.

$\exists s \in F_k$ s.t. $(s, s') \in T$ and $s' \notin P$	$\exists s \in F_i$ s.t. $s \neq s_b$ and $(s, s_b) \in T$
$\leftrightarrow$	$\leftrightarrow$
$F^{\#}_k(x_n, \cdots, x_0) \quad \wedge$	$F^{\#}_i(x_n, \cdots, x_0) \quad \wedge$
$T^{\#}(x_n, \cdots, x_0, x'_n, \cdots, x'_0) \quad \wedge$	$\neg s^{\#}_b(x_n, \cdots, x_0) \quad \wedge$
$\neg P^{\#}(x'_n, \cdots, x'_0)$	$T^{\#}(x_n, \cdots, x_0, x'_n, \cdots, x'_0) \quad \wedge$
	$s^{\#}_b(x'_n, \cdots, x'_0)$

Table 4.1. The form of the two types of formulas the IC3 algorithm uses a SAT solver for.

4.7 The solver usage of IC3 and PDR

The inventor of IC3 stated in [Bra12] that the incremental SAT solver usage of IC3 is different from that of other solver applications. We took a look at the solver usage of the PDR implementation TREBUCHET found in Niklas Eén's model checking tool collection ZZ[10]. This implementation uses a SAT solver incrementally, but it also uses more than one instance of that solver per run of the algorithm. The default solver used by TREBUCHET is a recent version of MINISAT by Niklas Sörensson. The clause involvement visualization does not result in readable pictures for this application, as its solver usage is atypical, with a large number of jobs, and a small number of conflicts.

The Figures 4.11 and 4.12 show how the average number of conflicts divided by the number of solver calls progresses over time. Each line in the plot denotes a separate solver instance. Remarkably enough, the number of conflicts per solver call is always very small, often even below one. Observe also that, for these two benchmarks, no single solver instance is used for more than a couple of seconds. We generated similar plots for all HWMCC'12 benchmarks[11] that TREBUCHET could solve within 5 minutes, and found that such low numbers of conflicts are typical behavior. As a compact summary of that experiment we provide Table 4.2. The table is sorted by the column "conflicts per call". The values given in the last row of the table, for benchmark `6s126`, differ significantly from the other entries. The reason is that for this benchmark a counterexample of length zero exists (i.e. $\neg P \cap I \neq \emptyset$).

The experiment motivates future work on how to exploit IC3's atypical solver behavior. It seems like using a conflict driven SAT solver for problems that constitute very few conflicts may not be the optimal choice. Alan Mishchenko, developer of the model checking environment ABC [BM10], tried an alternative.

Quote 4.11 (Alan Mishchenko, private communication). *Some time ago I tried to use a simplistic circuit-based solver, which is geared to small incremental low-conflict or no-conflict SAT problems. This solver made a big difference in the efficient implementation of SAT-based signal correspondence whose run time was improved more than 10x compared to the previous implementation, making it applicable to sequential circuits with*

[10] `http://bitbucket.org/niklaseen`

[11] `http://fmv.jku.at/hwmcc12`

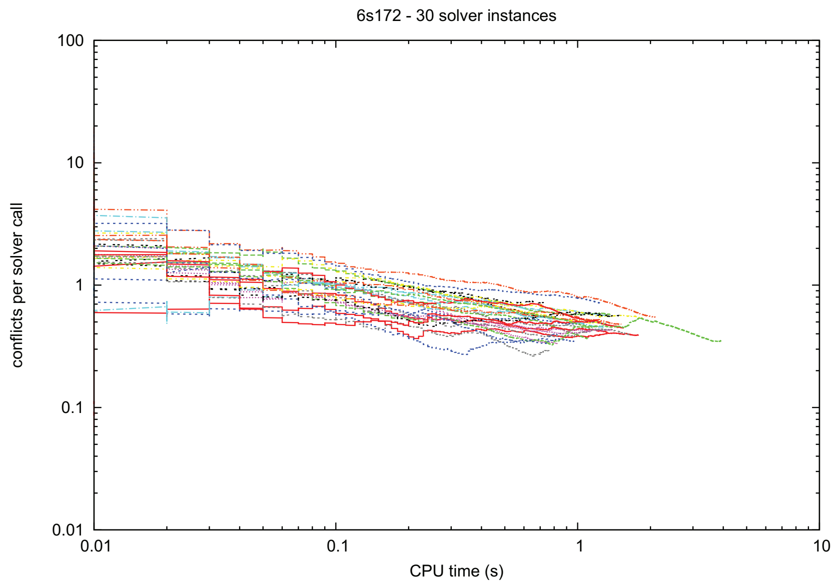


Figure 4.11. Conflicts per solver call for a satisfiable HWMCC'12 benchmark.

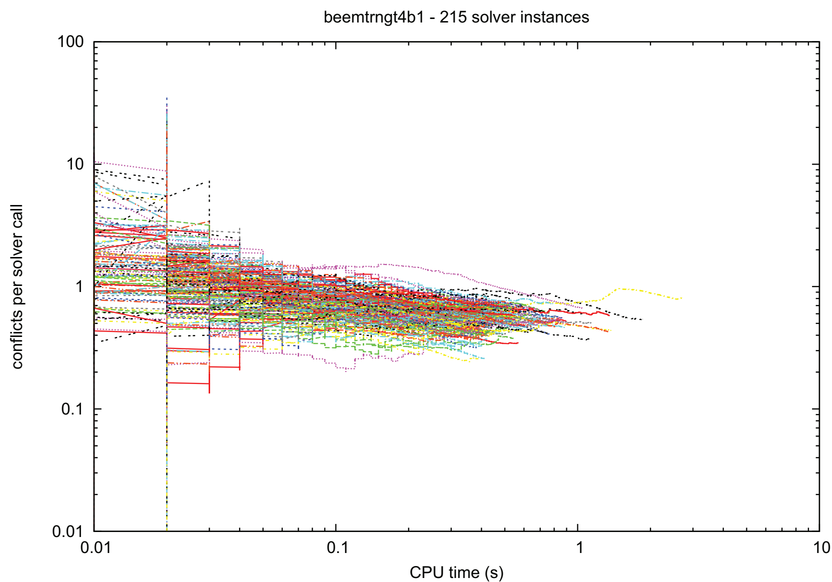


Figure 4.12. Conflicts per solver call for an unsatisfiable HWMCC'12 benchmark.

more than 1M AIG nodes. However, the main difference of using SAT in IC3/PDR, is that in addition to the circuit, there is a substantial number of CNF clauses constraining the values of flop output variables. Therefore, a pure circuit-based solver without efficient support for two-literal watching needed to perform the clause-based BCP would not be effective in this application.[12]

It seems that for the majority of the work performed the only solver features IC3 and PDR require are fast iterative unit propagation, and the ability to compute a final conflict clause. However, as final conflict clause computation requires almost all the bookkeeping needed for full conflict clause analysis it is not clear how to design a solver that can be faster by focusing on these two features alone[13].

It would be interesting to study IC3 and PDR as solver applications in more detail. A possible direction would be to classify the jobs generated by this algorithm based on their theoretical or empirical hardness. This can be done considering these jobs as individual problems, or as elements of the job sequence in which they appear. Clearly, jobs that are easy for a solver that has already solved a sequence of related jobs are not necessarily easy as individual problems. We performed the following experiment to illustrate that it is the incremental usage of the solver that enables solving a large fraction of the jobs without reaching any conflicts.

Experiment 4.12. *We modified* TREBUCHET *to print the clauses and assumptions sets that it sends to its SAT solver also into a set of iCNF files, one file for each solver instance. The* 30 *iCNF files generated for benchmark* `6s172`*, also used for Fig. 4.11, contained in total* 99233 *jobs. We gave the* 30 *files one by one to* MINISAT*, which solved all jobs in each of the files incrementally, leading to a total of* 46950 *conflicts, an average of half a conflict per job. We subsequently solved the* 99233 *jobs independently using the same solver without using any incremental features. The resulting sum of conflicts was* 1672105*, which is more than* 35 *times larger, and corresponds to an average of* 16.8 *conflicts per job.*

The 215 *iCNF files generated for* `beemtrngt4b1`*, also used for Fig. 4.12, contained in total* 367220 *jobs. Incremental solving results in* 202954 *conflicts (*0.6 *per job). Independent solving of all jobs results in* 2373878 *conflicts in total (*6.5 *per job). Clearly, for the two cases studied here, incre-*

---

[12]Flop output variables are the same as system state bits. Clause-based Boolean Constraint Propagation (BCP) is the same as iterative unit propagation.
[13]This insight was provided by Niklas Eén in a private communication.

mental usage is what enables solving a large fraction of the jobs without reaching any conflicts.

Many of the jobs generated by the IC3 algorithm correspond to satisfiable formulas, as this corresponds to all cases where the state sets represented by the frames must be further refined. As the number of conflicts is on average very small, the satisfying assignments of consecutive jobs are apparently close together in the search space. This property is exactly what is exploited by a technique called *model rotation*, for another incremental solver application called *MUS finding* (see Chapter 5). This technique prunes away substantial amounts of relatively easy work from the solver. The research into, and possible development of, a similar technique for IC3 is an ambitious target for future work.

benchmark	solver instances	solver calls	conflicts	conflicts per call	max. conflicts in one call
6s102	3	13151	3002	0.23	15
6s157	1	273	68	0.25	1
6s159	2	1588	390	0.25	5
6s162	240	174116	52280	0.30	25
6s51	9	93452	28911	0.31	19
6s194	31	199156	72595	0.36	16
beemldelec4b1	1	1201	485	0.40	46
6s43	22	109087	46642	0.43	75
beemfwt1b1	7	96327	42570	0.44	168
beempgmprot1b1	12	110280	50124	0.45	128
6s172	30	99233	46950	0.47	45
6s164	647	575192	279651	0.49	167
bob05	2	10780	5399	0.50	24
bob1u05cu	2	10780	5399	0.50	24
beemgear2b1	48	83007	42125	0.51	26
beembrptwo4b1	27	47389	25785	0.54	35
beemmcs1f1	11	8969	4875	0.54	22
beemlmprt5f1	38	41811	22791	0.55	16
beemtrngt2b1	55	72745	39869	0.55	28
beemtrngt4b1	215	367220	202954	0.55	35
beemfish4f1	226	188972	113495	0.60	24
6s34	64	197903	122575	0.62	21
beemlann2f1	37	37417	25232	0.67	13
beemtrngt3f1	586	506412	344180	0.68	25
beemlptna5f1	7	32688	22750	0.70	24
6s109	113	195247	139491	0.71	50
6s108	4	22776	16713	0.73	55

— table continues on the next page —

benchmark	solver instances	solver calls	conflicts	conflicts per call	max. conflicts in one call
bobsmoci	298	897107	667288	0.74	712
beemexit5f1	65	100170	75679	0.76	41
beemsnpse4f1	11	22055	17362	0.79	35
bobsmi2c	43	113225	89359	0.79	40
beembrp1f1	1175	1291474	1097954	0.85	28
beemrether6b1	1	1470	1270	0.86	58
beemtlphn5f1	260	695364	617253	0.89	258
beemhanoi1b1	197	453390	420086	0.93	124
pdtpmsns3	465	841812	813971	0.97	394
pdtpmssfeistel	66	180266	193179	1.07	543
beemptrsn1b1	225	239484	266603	1.11	66
beembkry1b1	46	41454	47897	1.16	56
beemndhm2f2	21	157640	189897	1.20	2083
beemldfilt4b1	304	535316	685360	1.28	111
beemrshr2b1	3	38207	51712	1.35	433
beemmcs4b1	563	703701	1000462	1.42	293
6s6	83	146992	214201	1.46	241
beemloyd2b1	6	8290	12402	1.50	45
eijkbs3330	55	68514	135582	1.98	1313
beemlmprt1b1	103	113713	236450	2.08	224
6s132	78	64206	135747	2.11	1928
beemmsmie1b1	108	123919	264616	2.14	586
6s173	67	178647	535369	3.00	222
beemfrogs1b1	78	290266	1255448	4.33	593
beemlifts8b1	1	2457	11143	4.54	332
neclaftp3002	1	374	1833	4.90	162
pdtpmsviper	7	63418	372162	5.87	8092
beemcoll1b1	1	8367	50754	6.07	3176
6s120	1	1169	15955	13.65	2163
bobsmrisc	3	85159	1246113	14.63	44768
bjrb07amba10andenv	1	7033	125837	17.89	5233
bobaesdinvdmit	3	97804	1936190	19.80	64302
beemprng1b1	1	603	13323	22.09	818
beempgsol5b1	2	5409	158305	29.27	12235
6s126	1	1	43709	43709.00	43709

Table 4.2. Statistics for HWMCC'12 benchmarks solved within 5 minutes by TREBUCHET.

5. Finding Minimal Unsatisfiable Subsets

This chapter discusses algorithms for finding minimal unsatisfiable subsets. It provides an extended discussion of the results published in Publication I and Publication IV. This includes the discussion of several previously undocumented features of an implementation of the algorithm presented in Publication I. Moreover, we provide several new theoretical results related to the work performed in Publication IV.

An unsatisfiable formula is *minimal unsatisfiable*, if removing any of its clauses makes it satisfiable. A *Minimal Unsatisfiable Subset* (MUS) of a formula is a subset of clauses of the formula that is minimal unsatisfiable. A clause whose removal from an unsatisfiable formula makes the formula satisfiable is called a *critical clause*. Here, we provide a generalized definition of criticality which can also be applied with respect to satisfiable formulas.

Definition 5.1 (Critical clause). *A clause c is critical*[1] *with respect to formula $\mathcal{F}$ if and only if the following equivalent claims hold:*

- $\mathcal{F} \setminus \{c\}$ *does not imply* c.
- $\mathcal{F} \setminus \{c\}$ *has a satisfying assignment not satisfying* c.
- $(\,\mathcal{F} \setminus \{c\}\,) \cup \neg c$ *is satisfiable, where* $\neg c = \{\,\{\neg l\} \mid l \in c\,\}$.

Throughout this chapter we will simply write that a clause is critical, without mentioning of the formula with respect to which it is critical if this is clear from the context Observe that a tautological clause, i.e. a clause c containing $l \in c$ and $\neg l \in c$ for some literal l, can never be critical. In the following discussion we therefore assume, without loss of generality, that formulas do not contain tautological clauses. A formula is

[1] Alternative names found in the literature are *transition clause* (e.g. [GMP08, ML11]) and *necessary clause* (e.g. [KLM06]).

Algorithm 1 Basic destructive MUS finding algorithm

Given: An unsatisfiable formula $\mathcal{F}$.

1. $M = \emptyset$
2. `while` $\mathcal{F} \neq M$
3.   `pick` a clause $c_i \in \mathcal{F} \setminus M$
4.   `if` $\mathcal{F} \setminus \{c_i\}$ is **satisfiable** `then`
5. $M = M \cup \{c_i\}$
6.   `else`
7. $\mathcal{F} = \mathcal{F} \setminus \{c_i\}$
8. `return` $\mathcal{F}$

irredundant [Lib05] if all of its clauses are critical. A formula is minimal unsatisfiable if and only if it is unsatisfiable and irredundant.

The complexity class DP contains all languages L such that $L = L_1 \cap L_2$ where $L_1 \in$ NP and $L_2 \in$ coNP. Testing irredundancy of a formula is an NP-complete problem [Lib05], whereas testing whether a formula is unsatisfiable is a coNP-complete problem. Hence, testing whether a formula is minimal unsatisfiable is in the complexity class DP. In fact, it is DP-complete [PW88].

5.1 Classical algorithms for MUS finding

Most well known algorithms for MUS finding are based on repeated calls to a SAT solver [Mar10]. Such algorithms have been categorized as *constructive*, *destructive* or *dichotomic* [GMP08]. All these algorithms are based on iteratively identifying critical clauses. Constructive algorithms start from an empty formula and add clauses until the formula becomes unsatisfiable. When this happens, the last clause that was added is critical with respect to the constructed formula. This was first observed in [dSNP88]. Destructive algorithms, on the contrary, start from the full formula and remove clauses until the formula becomes satisfiable. When this happens, the last clause removed is critical with respect to the formula it is removed from. The dichotomic approach identifies critical clauses using a binary search over the set of clauses in the input formula [HLSB06].

Possibly the simplest algorithm for MUS finding is the classical destructive algorithm, for which pseudocode is given in Alg. 1. It starts from the complete formula $\mathcal{F}$ and an empty set M. The set M represents the set

Algorithm 2 Basic constructive MUS finding algorithm

Given: An unsatisfiable formula $\mathcal{F}$.

```
1.  M = ∅
2.  while F ≠ M
3.      R = M
4.      L = ∅
5.      while R is satisfiable
6.          pick a clause c_i ∈ F \ R
7.          R = R ∪ {c_i}
8.          L = {c_i}
9.      F = R
10.     M = M ∪ L
11. return F
```

of clauses that the algorithm has proven critical. In every iteration the algorithm tests for a clause c_i that is still in $\mathcal{F}$ and not yet in M whether it is critical with respect to the clauses in $\mathcal{F}$. If it is, then it is added to M, otherwise it is removed from $\mathcal{F}$. This continues until $\mathcal{F}$ and M are equal, which means the algorithm has found a MUS of $\mathcal{F}$.

A basic constructive MUS finding algorithm is given in pseudocode in Alg. 2. The algorithm repeatedly identifies one critical clause by starting from a satisfiable formula R, and adding clauses to it until it becomes unsatisfiable. Any clauses that have not been added to R once it becomes unsatisfiable can naturally be removed from $\mathcal{F}$. The set L is keeping track of the last clause added to R, and thus throughout the algorithms execution always contains at most one clause. When R becomes unsatisfiable the clause in L is critical.

The basic destructive algorithm Alg. 1 performs at most m calls to the SAT solver, where $m = |\mathcal{F}|$. The number of SAT solver calls the constructive algorithm presented in Alg. 2 performs is bounded by $m \times k$ where k is the size of the largest MUS in $\mathcal{F}$. However, this does not mean that constructive algorithms are not interesting. First of all, it was shown in [ML11] that a more elaborate constructive algorithm requiring only $|\mathcal{F}|$ solver calls does exist. Secondly, the basic constructive algorithm Alg. 2 performs a lot of calls to the solver for problems that are severely underconstrained, and thus easy to solve. All solver calls inside one round are performed in a naturally incremental fashion, starting from a sim-

Algorithm 3 Constructive MUS finding algorithm from Publication I

Given: An unsatisfiable formula $\mathcal{F}$.

1. $M = \emptyset$
2. `while` $\mathcal{F} \neq M$
3. $\quad R = M$
4. $\quad P = \emptyset$
5. $\quad$ `while` $\mathcal{F} \neq R$
6. $\quad\quad$ `pick` a clause $c_i \in \mathcal{F} \setminus R$
7. $\quad\quad$ `if` $R \cup \neg c_i$ is **satisfiable** `then`
8. $\quad\quad\quad \alpha_i =$ a complete assignment satisfying $R \cup \neg c_i$
9. $\quad\quad\quad R = R \cup \{c_i\}$
10. $\quad\quad\quad P = \{c_j \mid c_j \in P$ and α_j satisfies $c_i\} \cup \{c_i\}$
11. $\quad\quad$ `else` $\mathcal{F} = \mathcal{F} \setminus \{c_i\}$
12. $\quad M = M \cup P$
13. `return` $\mathcal{F}$

ple problem and gradually extending it. Thus one round of the algorithm is arguably simply an organized way of solving one problem, rather than a loop solving k independent problems. We will continue this discussion in Section 5.7.

5.2 Constructive algorithm using associated assignments

In Publication I a constructive algorithm for MUS finding was presented. In Alg. 3 we give pseudocode for this algorithm. This presentation of the algorithm is different from the original presentation in Publication I, but nevertheless matches the original implementation of the algorithm. The publication made two significant contributions.

The first improvement of Alg. 3 with respect to Alg. 2 is that a clause is only added to the growing set R if it is critical with respect to R. This is achieved by testing the satisfiability of $R \cup \neg c_i$ instead of the satisfiability of R, before adding c_i. The idea is that for a clause to be critical with respect to a MUS of $\mathcal{F}$ it has to be critical with respect to all subsets of that MUS.

The second improvement is in the construction of the set P of *potentially critical clauses*. An invariant of this algorithm is that all clauses in the set P are critical with respect to R, and hence they are “potentially crit-

ical" with respect to a MUS of $\mathcal{F}$. The invariant is maintained using the satisfying assignments returned by the SAT solver.

Definition 5.2 (assoc). *An associated assignment (assoc) for a clause $c \in \mathcal{F}$ is a complete assignment α for the formula $\mathcal{F}$ that satisfies the formula $\mathcal{F} \setminus \{c\}$ and does not satisfy c.*

Note that by definition the existence of an assoc for a clause $c \in \mathcal{F}$ implies that c is critical for $\mathcal{F}$. In Alg. 3 on Line 8 an assoc, i.e. an assignment associated with c_i proving its criticality with respect to R, is stored. Because c_i is critical with respect to R it is also added to P, on Line 10. However, the addition of c_i to R may mean that some clauses that were already in P are no longer critical. Thus, on Line 10 only those clauses in P are maintained for which the previously stored assoc satisfies the newly added clause, i.e. is still an assoc for the new set R. The use of truth assignments to witness criticality apparently was an inspiration for the development of *model rotation* [ML11], which is used in the most successful MUS finders today, and which is discussed in detail in Section 5.3.

The implementation of this algorithm used for the performance evaluation presented in Publication I, was named MINIUNSAT, after the solver MINISAT it is based on. At the SAT competition[2] of 2011 there was a special track for MUS finders, in which an updated version of this MUS finder named MINIUNSAT2 competed. It contained several undocumented improvements compared to the original algorithm.

First of all, we observed that for correctness and termination it is sufficient to initialize R on Line 3 to any $R \subset \mathcal{F}$ such that $M \subseteq R$. This means that the number of SAT solver calls per round can be limited to a constant s_{max} by initializing R as follows:

Let	$R = M$	if $\lvert\mathcal{F}\rvert - \lvert M\rvert \leq s_{max}$,
and	$R \subset \mathcal{F}$ s.t. $M \subseteq R$ and $\lvert\mathcal{F}\rvert - \lvert R\rvert = s_{max}$	if $\lvert\mathcal{F}\rvert - \lvert M\rvert > s_{max}$.

During the competition MINIUNSAT2 was using this type of initialization with constant $s_{max} = 100$. The second undocumented feature is a redundancy check that we will discuss in Section 5.8.

5.3 Model rotation

In [ML11] a technique called *model rotation* was introduced. Shortly after the original publication it was improved to *recursive model rotation*

[2] http://www.satcompetition.org

Algorithm 4 Recursive model rotation

Given:

- A formula $\mathcal{F}$.
- A set $M \subseteq \mathcal{F}$ of critical clauses.
- A clause $c_i \in M$.
- An assoc $\alpha_i \in A(c_i, \mathcal{F})$.

subroutine `modelRotate`$(\mathcal{F}, M, c_i, \alpha_i)$

1. `for` $l \in c_i$ `do`
2. $\quad \alpha_j =$ `rotate`$(\alpha_i, \neg l)$
3. $\quad$ `if` (exactly one clause $c_j \in \mathcal{F}$ is not satisfied by α_j and $c_j \notin M$) `then`
4. $\quad\quad M = M \cup \{c_j\}$
5. $\quad\quad$ `modelRotate`$(\mathcal{F}, M, c_j, \alpha_j)$

[BM11], which has become a standard technique for MUS finders. Model rotation provides another way of benefiting from the satisfying assignments returned by the solver used for MUS finding.

Definition 5.3 (Set of assocs). *Let $A(c, \mathcal{F})$ be the set of all assocs for $c \in \mathcal{F}$.*

Clearly, for any formula $\mathcal{F}$ and clause $c \in \mathcal{F}$ a single assoc $\alpha \in A(c, \mathcal{F})$ or prove that $A(c, \mathcal{F}) = \emptyset$ can be obtained by testing the satisfiability of the formula $(\mathcal{F} \setminus \{c\}) \cup \neg c$ using a SAT solver. Model rotation is an algorithm that given an assoc for a clause attempts to find an assoc for another clause by negating a single literal in the assoc.

Definition 5.4 (Rotation function). *Let* `rotate`(α, l) *be a function that negates literal l in assignment α, i.e.:* `rotate`$(\alpha, l) = (\alpha \setminus \{l\}) \cup \{\neg l\}$.

The pseudocode for the recursive model rotation algorithm is shown in Alg. 4. The algorithm can be used as a subroutine in any MUS finding algorithm, whenever a new critical clause is discovered. For example, to use model rotation inside the basic destructive algorithm[3], it should be executed after every addition of a clause c_i to M on Line 5 of Alg. 1. Observe that the required assoc α_i is the satisfying assignment the solver found when it solved the formula $\mathcal{F} \setminus \{c_i\}$, where $\mathcal{F}$ was known to be unsatisfiable. In Publication IV we viewed model rotation as an algorithm that

[3]In e.g. [ML11, BLM12, MJB13] the combination of Alg. 1 with Alg. 4 is named the *hybrid algorithm*.

traverses a graph with one vertex for each clause, called the *flip graph*.

Definition 5.5 (Flip graph). *For a formula $\mathcal{F}$ the flip graph $G_{\mathcal{F}} = (V, E)$ is a graph which has a vertex for every clause, i.e. $V = \mathcal{F}$. Each edge $(c_i, c_j) \in E$ is labelled with the set of literals $L(c_i, c_j)$ such that:*

$$L(c_i, c_j) \quad = \quad \{l \mid l \in c_i \text{ and } \neg l \in c_j\}.$$

The set of edges E of the flip graph is defined by $(c_i, c_j) \in E$ if and only if $L(c_i, c_j) \neq \emptyset$.

Even though $(c_i, c_j) \in E$ if and only if $(c_j, c_i) \in E$ in this work the flip graph is considered to be a directed graph. This is useful for defining the *rotation edges*. The undirected version of this graph is sometimes referred to as the *resolution graph* (e.g. [Sin07]), and will be further discussed in Section 5.8.

Definition 5.6 (Rotation edges). *Given a formula $\mathcal{F}$, let the sets of possible rotation edges*[4] *E_P, and guaranteed rotation edges E_G be defined as:*

$$\begin{aligned} E_P &= \{(c_i, c_j) \mid \quad c_i, c_j \in \mathcal{F} \text{ and } |L(c_i, c_j)| = 1\}, \text{ and:} \\ E_G &= \{(c_i, c_j) \mid \quad c_i, c_j \in \mathcal{F} \text{ and } |L(c_i, c_j)| = 1 \text{ and for all } c_k \in \mathcal{F} \\ &\qquad\qquad\qquad \text{such that } c_k \neq c_j \text{ it holds that } L(c_i, c_j) \neq L(c_i, c_k)\}. \end{aligned}$$

In Fig. 5.1 the flip graph for an example formula $\mathcal{F}_{fig5.1}$ is given. Because there are no two clauses $c_i, c_j \in \mathcal{F}_{fig5.1}$ such that $|L(c_i, c_j)| > 1$ it holds that the set of possible rotation edges E_P is equal to the set of all edges E in the flip graph. However, only the solid edges in the figure belong to the set of guaranteed rotation edges E_G. The dotted edges are not in the set E_G because the two outgoing edges from vertex c_1 have the same label $L(c_1, c_2) = L(c_1, c_3) = \{x\}$. In Publication IV we prove the following theorem:

Theorem 5.7. *Let $\mathcal{F}$ be an unsatisfiable formula and E_G the set of guaranteed rotation edges it induces. If $(c_i, c_j) \in E_G$ then for any assoc $\alpha_i \in A(c_i, \mathcal{F})$ an assignment $\alpha_j = \texttt{rotate}(\alpha_i, \neg l)$ such that $L(c_i, c_j) = \{l\}$ is an assoc $\alpha_j \in A(c_j, \mathcal{F})$.*

This theorem implies that if we find an assoc for a clause then model rotation is guaranteed to find an assoc for all clauses that are reachable

[4]Note that the set E_P corresponds to all pairs of clauses (c_i, c_j) on which resolution $c_i \otimes c_j$ can be performed without creating a tautology.

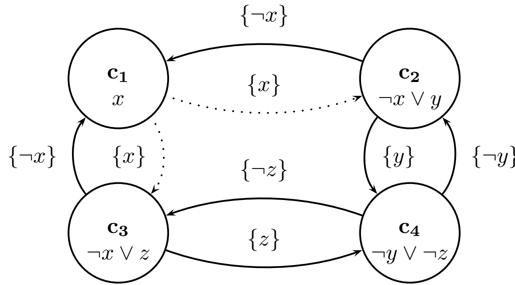


Figure 5.1. The *flip graph* for the formula $\mathcal{F}_{fig5.1} = \{\{x\}, \{\neg x, y\}, \{\neg x, z\}, \{\neg y, \neg z\}\}$.

from that clause over edges in E_G. It is shown in Publication IV that formulas which are commonly used for benchmarking MUS finding algorithms contain large numbers of guaranteed rotation edges. Using this observation we computed, for a set of formulas typically used for benchmarking MUS finders, an upper bound on the minimum number of calls to a SAT solver needed by the destructive algorithm Alg. 1 when it is using Alg. 4 as a subroutine. This number is typically much smaller than the number of clauses in the input formula. We used this observation to argue about the strength of model rotation.

Example 5.8. *From Theorem 5.7 it follows that an assoc exists for every clause for which there is a path over edges in E_G from a clause for which an assoc exists. Thus for formula $\mathcal{F}_{fig5.1}$ presented in Fig. 5.1 obtaining any assoc $\alpha \in A(c_i, \mathcal{F}_{fig5.1})$ for $i \in \{2, 3, 4\}$ is sufficient to determine that all clauses in the formula are critical. Obtaining an assoc for clause c_1 may however be less effective. Note that:*

$$A(c_1, \mathcal{F}_{fig5.1}) = \{\, \{\neg x, \neg y, \neg z\}, \{\neg x, \neg y, z\}, \{\neg x, y, \neg z\} \,\}.$$

Although by replacing $\neg x$ by x the second and third assoc in this set can be rotated into a valid assoc for c_2 and c_3 respectively, no negation of a single literal will make the first assoc into a valid assoc for any other clause in the formula $\mathcal{F}_{fig5.1}$.

5.4 Weakening the termination condition

In Publication IV we present an algorithmic improvement for the recursive model rotation algorithm Alg. 4. We show an example for a formula with nine clauses in which starting from an assoc for one clause we find an assoc for five other clauses. We then showed that if the algorithm would

have been modified by removing the condition $c \notin M$ from Line 3 it would have found an assoc also for the three other clauses. The problem is that after removing this condition the algorithm is no longer guaranteed to terminate, as the algorithm may traverse a cycle in the flip graph. In other words, to guarantee termination we must weaken the condition in such a way that a vertex may be visited several times, but not infinitely often.

Visiting the same vertex twice with the same assoc is not useful, so a simple weakened termination condition would be to check when we reach a critical clause whether we already reached this clause using the same assoc. However, this requires storing a potentially exponential number of assocs with every critical clause. In [BLM12] it was suggested to store a limited number rd ("rotation depth") of assocs with each clause identified as critical, and to allow rd distinct traversals of the same critical clause. The authors did not manage to achieve a performance improvement using this technique. Our solution requires less memory, does not require choosing any constants, and provides a consistent performance improvement.

Observe that visiting a critical clause multiple times may be useful because rotation may yield different results when started from a different assoc. Therefore, our intention was to define a termination condition that would enforce that different traversals of the same clause are on different paths. Our solution is given in pseudocode in Alg. 5. It associates with every literal l in every clause c a Boolean value $seen(c, l)$. Whenever model rotation is called from the MUS finder the value $seen(c, l)$ is set to **false** for all clauses and all literals. If the subsequent recursive calls to model rotation find an assoc for a clause c after rotation of literal l and $seen(c, l)$ is **false** then $seen(c, l)$ is set to **true** before traversing c. If on the other hand $seen(c, l)$ is already **true** then clause c is not traversed.

5.5 Blocked rotation edges

In Publication IV we defined a subset of possible rotation edges $E_G \subseteq E_P$ on which rotation is guaranteed to succeed. Here, we discuss a new result[5] regarding the possible existence of edges in E_P on which rotation

[5] The content of the Sections 5.5 and 5.6 has not been previously published. It does appear in a technical report [Wie13] that was made publicly available before the completion of this dissertation.

Algorithm 5 Improved recursive model rotation

Given:

- A formula $\mathcal{F}$.
- A set $M \subseteq \mathcal{F}$ of critical clauses.
- A clause $c_i \in M$.
- An assoc $\alpha_i \in A(c_i, \mathcal{F})$.

---

subroutine `improvedModelRotate`$(\mathcal{F}, M, c_i, \alpha_i)$

I. `for` all $c \in \mathcal{F}$ and all $l \in c$ `do` $seen(l, c) =$ **false**

II. `improvedMRRec`$(\mathcal{F}, M, c_i, \alpha_i)$

subroutine `improvedMRRec`$(\mathcal{F}, M, c_i, \alpha_i)$

1. `for` $l \in c_i$ `do`
2. $\quad \alpha_j = \texttt{rotate}(\alpha_i, \neg l)$
3. $\quad$ `if` (exactly one clause $c_j \in \mathcal{F}$ is not satisfied by α_j and $seen(c_j, \neg l) =$ **false**) `then`
4. $\quad\quad seen(c_j, \neg l) =$ **true**
5. $\quad\quad M = M \cup \{c_j\}$
6. $\quad\quad$ `improvedMRRec`$(\mathcal{F}, M, c_j, \alpha_j)$

---

is guaranteed to fail.

Definition 5.9 (Blocked rotation edge). *An edge $(c_i, c_j) \in E_P$ is blocked if for all $\alpha_i \in A(c_i, \mathcal{F})$ we have $\texttt{rotate}(\alpha_i, \neg l) \notin A(c_j, \mathcal{F})$, where l is the literal such that $L(c_i, c_j) = \{l\}$.*

Corollary 5.10. *If and only if $(c_i, c_j) \in E_P$ is a blocked edge then $(c_j, c_i) \in E_P$ is a blocked edge.*

Naturally, an edge $(c_i, c_j) \in E_P$ is a blocked edge if either $A(c_i, \mathcal{F}) = \emptyset$ or $A(c_j, \mathcal{F}) = \emptyset$. However, we will show that blocked edges may also exist between two critical clauses.

Lemma 5.11. *Let $\mathcal{F}$ be a formula, and $c_i, c_j \in \mathcal{F}$ a pair of clauses such that $L(c_i, c_j) = \{l\}$. If for some literal $l' \neq l$ it holds that $\mathcal{F} \setminus \{c_i, c_j\} \models l \leftrightarrow l'$ then the edge $(c_i, c_j) \in E_P$ is blocked.*

Proof. *For all $\alpha_i \in A(c_i, \mathcal{F})$ it holds that $\neg l \in \alpha_i$ and α_i satisfies $\mathcal{F} \setminus \{c_i, c_j\}$, thus $\neg l' \in \alpha_i$ holds. But then any assignment $\texttt{rotate}(\alpha_i, \neg l)$ contains l and $\neg l'$ and therefore does not satisfy $\mathcal{F} \setminus \{c_i, c_j\}$. It follows that no such assignment can be an assoc for c_j.*

Lemma 5.11 can be generalized, for example, by replacing the literal l' with any formula $\mathcal{Q}$ such that l does not occur in $\mathcal{Q}$ and $\mathcal{F}\backslash\{c_i, c_j\} \models l \leftrightarrow \mathcal{Q}$. Hence, the lemma provides a sufficient condition for blocking the edge between two critical clauses c_i and c_j, but this is not a necessary condition.

An interesting observation is that we can create an irredundant formula $\mathcal{F}$ with a clause $c_i \in \mathcal{F}$ such that for all $c_j \in \mathcal{F}$ all edges $(c_i, c_j) \in E_P$ are blocked. This means that for this formula model rotation starting at c_i can never find an assoc for any other clause, neither can model rotation starting from any other clause result in an assoc for clause c_i.

Example 5.12. *Consider the following irredundant satisfiable formula $\mathcal{F}$:*

$$\begin{aligned} \mathcal{F} = \{ \quad & c_0 = x \vee y, \\ & c_1 = p \vee \neg x, \quad c_2 = \neg p \vee x, \\ & c_3 = q \vee \neg x, \quad c_4 = \neg q \vee x, \\ & c_5 = r \vee \neg y, \quad c_6 = \neg r \vee y, \\ & c_7 = s \vee \neg y, \quad c_8 = \neg s \vee y \quad \}. \end{aligned}$$

Note that this formula represents four equivalences $p \leftrightarrow x$, $q \leftrightarrow x$, $r \leftrightarrow y$, and $s \leftrightarrow y$. Together, these make sure that for all $c \in \mathcal{F}$ it holds that the edge $(c_0, c) \in E_P$ is blocked. The formula can be made minimal unsatisfiable without breaking this property, for example by adding one clause for each one of the three satisfying assignments of the formula. This yields the following minimal unsatisfiable formula $\mathcal{F}'$:

$$\begin{aligned} \mathcal{F}' = \mathcal{F} \cup \{ \quad & c_9 = p \vee q \vee \neg r \vee \neg s \vee x \vee \neg y, \\ & c_{10} = \neg p \vee \neg q \vee r \vee s \vee \neg x \vee y, \\ & c_{11} = \neg p \vee \neg q \vee \neg r \vee \neg s \vee \neg x \vee \neg y \quad \}. \end{aligned}$$

5.6 Proof of a conjecture by Belov et al.

In [BLM12] a conjecture is presented that we prove here. The conjecture states a property of the *rotation graph*, which was defined alongside the conjecture. Here we state an equivalent definition for the rotation graph using slightly different notation.

Definition 5.13 (Rotation graph). *Let $\mathcal{F}$ be an unsatisfiable formula, and let $Unsat(\mathcal{F}, \alpha)$ be the set of clauses in $\mathcal{F}$ not satisfied by assignment α, i.e. $Unsat(\mathcal{F}, \alpha) = \{c \mid c \in \mathcal{F} \text{ and } c \cap \alpha = \emptyset\}$. The rotation graph $\mathcal{R}_{\mathcal{F}} = (V_R, E_R)$ is a directed graph which has a vertex for each complete*

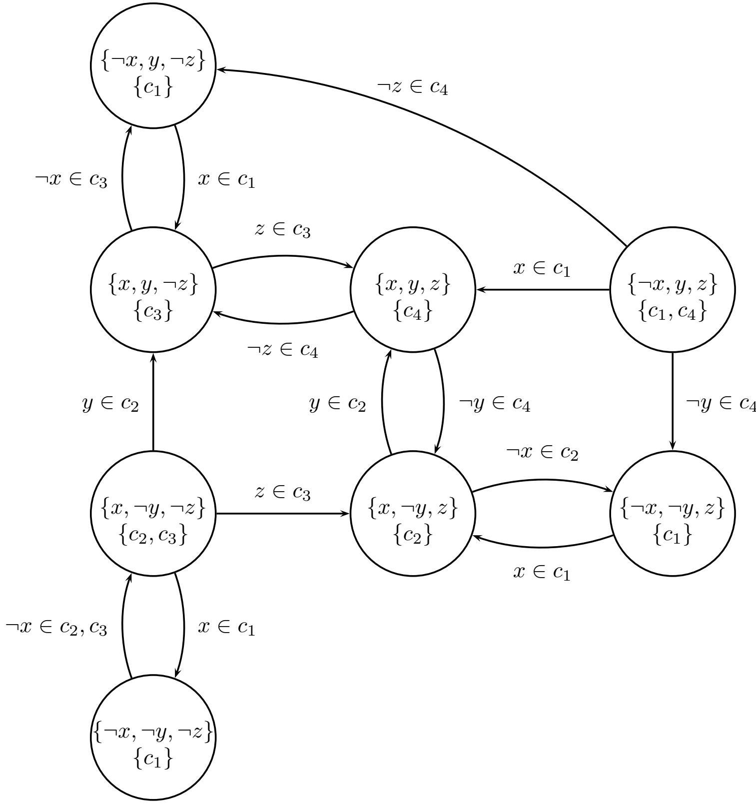


Figure 5.2. The rotation graph for the formula $\mathcal{F}_{fig5.1}$.

assignment to the variables of $\mathcal{F}$. *There exists an edge* $(\alpha, \alpha') \in E_R$ *if* $\alpha' = \texttt{rotate}(\alpha, \neg l)$ *for some literal* $l \in \bigcup Unsat(\mathcal{F}, \alpha)$.

Example 5.14. *Figure 5.2 provides the rotation graph for our example formula* $\mathcal{F}_{fig5.1}$. *All vertices are labeled with a complete assignment* α *to the variables* x, y *and* z, *and with the set of clauses* $Unsat(\mathcal{F}_{fig5.1}, \alpha)$. *For clarity each edge in the graph is labeled with the literal that justifies its existence.*

A *witness assignment*, as mentioned in the following quote, is exactly the same as an assoc.

Quote 5.15 (Conjecture from [BLM12]). *Let* $\mathcal{F}$ *be a minimally unsatisfiable formula, and let* $\mathcal{R}_\mathcal{F}$ *be the rotation graph of* $\mathcal{F}$. *Then, there exists a witness assignment* v *such that the traversal of* $\mathcal{R}_\mathcal{F}$ *starting from* v *visits at least one witness assignment for each clause* $c \in \mathcal{F}$.

The possible existence of critical clauses that are connected only through blocked edges in the flip graph, as in Example 5.12, does not disprove this

conjecture. This is because the traversal of the rotation graph as defined here may pass through assignments α for which $|Unsat(\mathcal{F}, \alpha)| > 1$, i.e. it may perform rotation through assignments that are not an assoc for any clause. The intuition behind the following lemma is that from any complete assignment we can always traverse an edge in the rotation graph which brings us strictly closer to a chosen destination assoc.

Lemma 5.16. *Let $\mathcal{F}$ be an unsatisfiable formula, and let α_j be an assoc for some clause $c_j \in \mathcal{F}$, i.e. $\alpha_j \in A(c_j, \mathcal{F})$. Moreover, let α_i be a complete assignment which is not an assoc for c_j, i.e. $\alpha_i \notin A(c_j, \mathcal{F})$. Then there exists a literal $l \in \bigcup Unsat(\mathcal{F}, \alpha_i)$ such that $l \in D$ where $D = \alpha_j \setminus \alpha_i$.*

Proof. *Let $c_i \in Unsat(\mathcal{F}, \alpha_i)$ such that $c_i \neq c_j$. Such a clause must exists because $Unsat(\mathcal{F}, \alpha_i)$ is both non-empty (as $\mathcal{F}$ is unsatisfiable), and not equal to $\{c_j\}$ (as α_i is not an assoc for c_j). As α_j satisfies c_i while α_i does not, it must hold for some $l \in c_i$ that $l \in \alpha_j$ and $l \notin \alpha_i$, hence $l \in D$.*

Lemma 5.17. *Let $\mathcal{F}$ be an unsatisfiable formula, let $c_j \in F$ be a critical clause, and let α_i be a complete assignment for to the variables of $\mathcal{F}$. There exists a path in the rotation graph starting from the vertex corresponding to assignment α_i to an assoc $\alpha_j \in A(c_j, \mathcal{F})$.*

Proof. *We will show how to construct a rotation path starting from α_i that is guaranteed to end in an assoc for c_j. For some $\alpha_j \in A(c_j, \mathcal{F})$ let $D = \alpha_j \setminus \alpha_i$. The path begins at the vertex corresponding to assignment $\alpha = \alpha_i$. The path is completed when we reach an assignment α that is an assoc for c_j. By combining Definition 5.13 and Lemma 5.16 we may observe that if α is not an assoc for c_j then there exists a literal $l \in D$ such that $(\alpha, \alpha') \in E_R$ for $\alpha' = \texttt{rotate}(\alpha, \neg l)$. Hence, the path can proceed from α to α', after which l can be removed from D. At α' we repeat the previous, i.e. either we find that α' is an assoc for c_j or we compute the next step in the path. As one element is removed from D in every step the path is guaranteed to end in an assoc for c_j.*

Lemma 5.17 states that starting from any complete assignment there exists a path to an assoc for any arbitrary critical clause. Hence, the conjecture in Quote 5.15 must hold. In fact, we can even strengthen the conjecture to the following corollary.

Corollary 5.18. *Let $\mathcal{F}$ be an unsatisfiable formula, and let $\mathcal{R}_{\mathcal{F}}$ be the rotation graph of $\mathcal{F}$. Starting from any complete assignment to the variables*

of $\mathcal{F}$ (any vertex in V_R), there exists a path in $\mathcal{R}_{\mathcal{F}}$ that visits an assoc for every clause $c \in \mathcal{F}$ such that $A(c, \mathcal{F}) \neq \emptyset$.

Clearly, a variant of model rotation that may traverse all edges in the rotation graph (called unrestricted EMR in [BLM12]) can reach an assoc for any critical clause in the input formula, starting from any complete assignment. Unfortunately this result does not provide much insight in the strength of conventional model rotation.

5.7 Using the solver efficiently

A powerful way to improve the implementation of MUS finding algorithms is by using a *proof-logging* SAT solver, i.e. a solver which can provide a proof of unsatisfiability. The set of clauses used in such a proof is by definition always an unsatisfiable subset of the solver's input formula. Hence, if we use a proof-logging solver for Alg. 1 then on Line 7 of the pseudocode we can delete from $\mathcal{F}$ any clause that is not used in the proof provided by the solver. Although the power of such techniques has been reconfirmed recently [NRS13], the approach has not been particularly popular. Its disadvantages are that storing proofs may require a large amount of memory, and that state-of-the-art solvers with proof-logging features have not been widely available. This may be changing in the near future as recently there has been an interest in generating compact unsatisfiability proofs [HHW13], and implementing proof-logging has been actively encouraged by the SAT competition[2] of 2013.

A commonly used implementation of the destructive algorithm given in Alg. 1 uses an incremental SAT solver without proof-logging features, by making use of selector variables. For each clause $c_i \in \mathcal{F}$ a selector variable s_i is created in the solver. Recall that selector variables, discussed in Chapter 2, are auxiliary variables that do not occur anywhere in the original formula $\mathcal{F}$. Instead of loading the formula $\mathcal{F}$ the set of clauses $\{c_i \vee \neg s_i \mid c_i \in \mathcal{F}\}$ is loaded in the solver. The solver can now be used to test the satisfiability of any subformula of the original formula $\mathcal{F}$ by solving under a set of assumptions that contains literal s_i for each clause c_i that we wish to include during the test. The final conflict returned by the solver can be used to delete more than one clause per iteration, i.e. on Line 7 of the pseudocode we can delete any clause c_i for which selector s_i does not occur in the final conflict. This is called *clause set refinement* [ML11], and it is crucial for the performance of this algorithm. An impor-

tant optimization of the implementation is to add the unit clause $\{s_i\}$ to the solver for every clause c_i we add to M, and the unit clause $\{\neg s_i\}$ for any clause c_i that we remove from $\mathcal{F}$.

As already discussed in Section 5.1, MUS finding algorithms may differ in the number of calls they make to the SAT solver that they use internally. Dichotomic algorithms [Jun04, GMP08] are particularly interesting from a theoretical point of view, as they require only $O(k \times log(m))$ calls to a solver. Unfortunately these algorithms do not perform well in practice [BLM12] as they fail to efficiently exploit incremental SAT solving technology. A recent dichotomic-style algorithm called the *progression algorithm* [MJB13] also requires only $O(k \times log(m))$ solver calls to compute a MUS, and has been shown to have excellent performance in practice.

Just looking at the number of SAT solver calls an algorithm will perform does not give much information about the practical performance of the algorithm. Its not how often we make the solver do work, but how much work the solver performs in total that counts. The constructive algorithms of Alg. 2 and Alg. 3 perform at most k calls to a SAT solver *per round*, however all solver calls in one round form a natural way of introducing the problem $\mathcal{F}$ to the solver one clause at a time.

Experiment 5.19. *The set of application benchmarks from the SAT competition*[2] *in 2011 contained* 65 *unsatisfiable benchmarks that could be solved using the solver* MINISAT 2.2.0 *within ten minutes on our hardware. We performed two experiments with this set of unsatisfiable benchmarks. A memory limit of* $7.5GB$ *was employed.*

The first experiment simulates the first round of a constructive MUS finding algorithm by adding the clauses from the input file one clause at a time, and running the solver after every clause addition. As can be seen from Fig. 5.3 using this approach 53 *of the* 65 *benchmarks were still found unsatisfiable within ten minutes. The memory limit was never exceeded. Note that we used one solver call per clause, and that this type of repeated addition of clauses corresponds to the type of incremental solving of a single problem envisioned in [Hoo93].*

For the second experiment we added a selector variable to every clause in the 65 *formulas, and then solved each of those formulas using a single solver call under the set of assumptions that requires all selector variables to attain the value* ***false****. In this experiment the solver returned the answer unsatisfiable for only* 37 *formulas. For the remaining* 28 *formulas the solver failed because the time limit was exceeded in* 18 *cases, and*

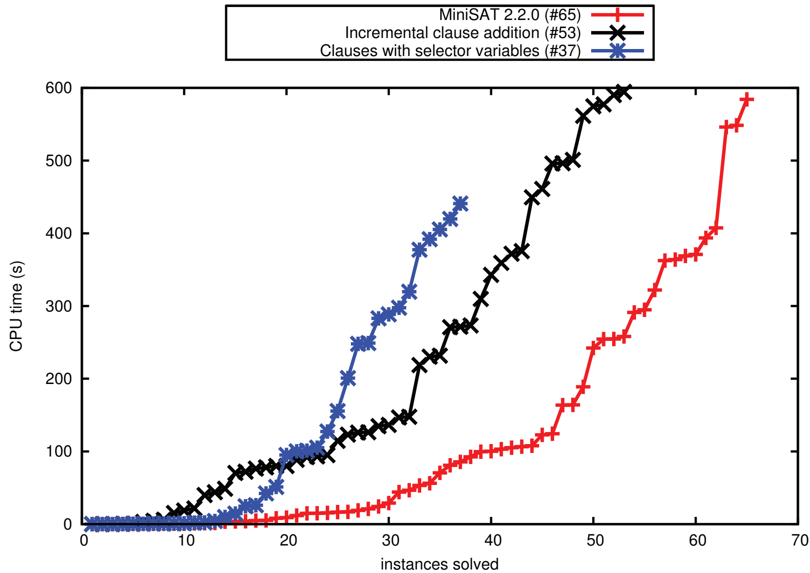


Figure 5.3. Cactus plot for Experiment 5.19.

because the memory limit was exceeded in 10 *cases.*

One of the problems with the selector variable approach is that it significantly harms the solvers ability to propagate and learn. If all problem clauses have a selector variable then every learnt clause will contain the assumption literals for all the clauses used in its derivation. This increases the memory usage and slows down the unit propagation procedure. This problem was addressed in [LB13], where the authors propose to shorten such learnt clauses by using auxiliary variables as shorthands for disjunctions of assumption literals. It should be noted that, unlike the other two approaches, the selector variable based implementation provides a simple way of extracting unsatisfiable cores through clause set refinement. For this reason the performance comparison in Experiment 5.19 is arguably not completely fair.

Model rotation can substantially reduce the number of solver calls required by a MUS finding algorithm. It exploits the close proximity of satisfying assignments, in terms of Hamming distance, of consecutive satisfiable solver calls performed by such algorithms. As a result, the solver calls that are avoided using model rotation are usually not those corresponding to hard jobs. Figures 5.4 and 5.5 show the timed clause involvement visualization for the learnt clauses alone, for an execution of the

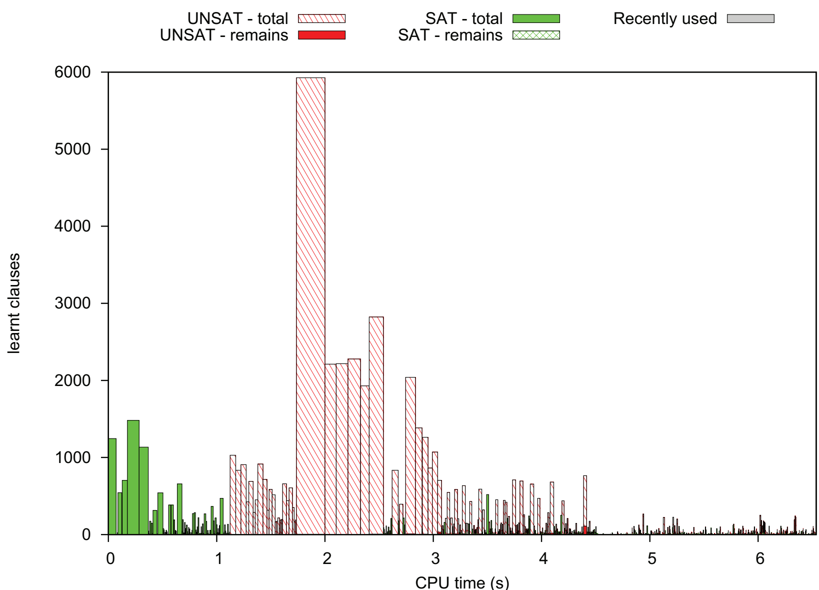


Figure 5.4. Timed clause involvement visualization for Alg. 1, given benchmark `barrel5` from [BCCZ99].

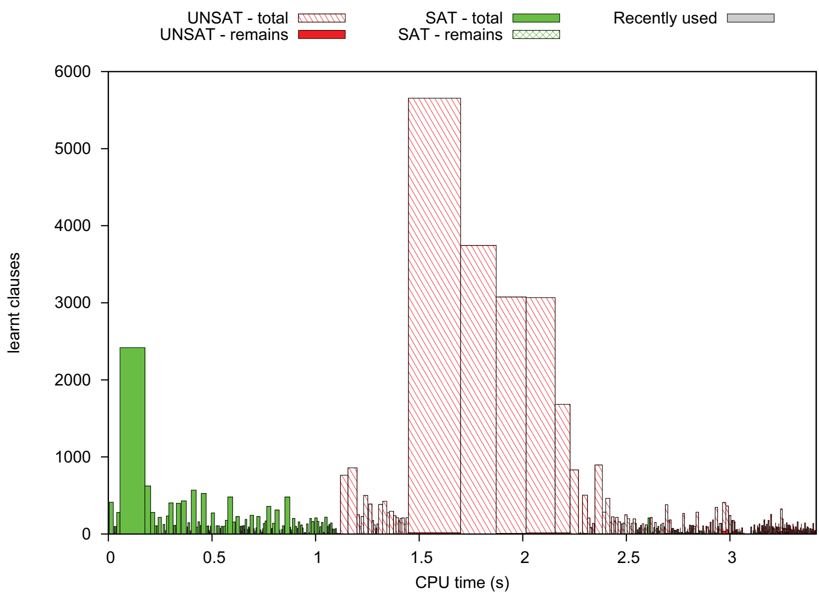


Figure 5.5. Timed clause involvement visualization for Alg. 1 using rotation Alg. 5, given benchmark `barrel5` from [BCCZ99].

destructive algorithm[6] Alg. 1, with and without the use of model rotation algorithm Alg. 5. The clause involvement plots for the problem clauses have been left out because these would have been empty. This is because the first job was very easy in these two cases, and thus the corresponding problem clause set bar would have been too narrow to see. After the first job no problem clauses are added to the solver[7], hence there would not be any bars visible for other jobs either. Observe that the two figures illustrate a very similar solver behavior, but the scale of the horizontal axes are different. Without model rotation 5383 solver calls were performed, whereas with model rotation only 2953 solver calls were necessary. The reduction of the number of easy satisfiable solver calls can be seen from the figures, as Fig. 5.5 seems like a condensed version of Fig. 5.4, where all the whitespace caused by "invisibly narrow" bars has been removed. The other thing that may be observed is that there is no visible evidence of clause reuse. This is not surprising as the learnt clauses on average contain many selector variables, and are thus providing logically weak and "job local" constraints.

5.8 Redundancy removal techniques

In this section we will discuss various forms of redundancy removal that can be implemented in MUS finding algorithms. In Section 5.3 we defined the *flip graph*, and already mentioned the existence of its undirected and unlabeled version called the *resolution graph*, which we will now formally define.

Definition 5.20 (Resolution graph). *Let the resolution graph $G^R_{\mathcal{F}} = (V, E)$ of a formula $\mathcal{F}$ be the undirected graph with vertices $V = \mathcal{F}$ and edges $(c_i, c_j) \in E$ if and only if for some $l \in c_i$ we have $\neg l \in c_j$.*

Resolution can only be performed between any two clauses that are connected in the resolution graph. By definition every clause in a minimal unsatisfiable formula $\mathcal{F}$ is used in any resolution refutation of $\mathcal{F}$, thus the resolution graph of any minimal unsatisfiable formula $\mathcal{F}$ is connected. This justifies the following redundancy removal technique that was im-

[6] The MUS finder used to generate these figures is a simple selector variable based implementation built on top of MINISAT. It is available from: `http://www.siert.nl/thesis`

[7] Except unit clauses, but those are not visualized.

plemented in MINIUNSAT2.

Definition 5.21 (Resolution graph based redundancy removal). *Given a clause c that is critical with respect to the unsatisfiable formula $\mathcal{F}$, remove any clause $c' \in \mathcal{F}$ for which there exists no path from c to c' in the resolution graph.*

Note that this check can be performed by any MUS finding algorithm once it has determined at least one critical clause. However, destructive algorithms implemented using clause set refinement or using core extraction from the resolution refutation, will perform this same reduction automatically as a side-effect of unsatisfiable SAT solver calls. As a result, these type of redundancy removal techniques tend to have little effect on the average performance of such algorithms. It is nevertheless not hard to construct a motivating example.

Example 5.22. *Let $\mathcal{F}_{mu}$ be a minimal unsatisfiable formula, which is constructed such that a satisfying assignment for any proper subformula of $\mathcal{F}_{mu}$ can be found in a fraction of the time it takes to prove that $\mathcal{F}_{mu}$ is unsatisfiable. Instances of the pigeon hole problem are an example of such formulas. Let $\mathcal{F}_{sat}$ be an easy to solve satisfiable formula that has no variables in common with $\mathcal{F}_{mu}$, i.e. $Var(\mathcal{F}_{sat}) \cap Var(\mathcal{F}_{mu}) = \emptyset$. Let $\mathcal{F} = \mathcal{F}_{mu} \cup \mathcal{F}_{sat}$ be the input of a destructive MUS finder, such as Alg. 1 or Alg. 4. It would be easy for these algorithms to prove that $\mathcal{F}_{mu}$ is irredundant, as this would require solving only easy satisfiable problems. However, as the algorithms rely on solving unsatisfiable problems to identify redundancy, they must perform the difficult task of proving $\mathcal{F}$ unsatisfiable to reduce $\mathcal{F}$ to $\mathcal{F}_{mu}$.*

Using the redundancy removal technique from Def. 5.21 this can be avoided. If on input $\mathcal{F}$ the algorithm is lucky (or clever) enough to perform the first solver call for $\mathcal{F} \setminus \{c\}$ where $c \in \mathcal{F}_{mu}$, then it will easily find that c is critical. None of the clauses of $\mathcal{F}_{sat}$ are connected to c in the resolution graph, and hence all of $\mathcal{F}_{sat}$ can be removed. Consequently, the algorithm can complete without ever having to prove any formula unsatisfiable.

This redundancy check can be generalized, making it stronger. It is well known that any clause that contains a pure literal, i.e. a literal whose negation does not exist in the formula, can always be removed from an unsatisfiable formula without rendering the formula satisfiable [DLL62]. The reason is that such a clause can always be satisfied by assigning the pure literal to **true**, and this has no implications for any clause of the

formula that does not contain that pure literal. *Autarkies* can be seen as a generalization of the pure literal rule over multiple literals.

Definition 5.23 (Autarky [MS85]). *An autarky, or autark assignment, is an assignment that satisfies all clauses it touches, i.e. all clauses that contain a variable which is assigned **true** or **false** by the assignment.*

Corollary 5.24. *An assignment α is an autarky for a formula $\mathcal{F}$ if and only if for all $l \in \alpha$ and all $c \in \mathcal{F}$ such that $\neg l \in c$ it holds that $\alpha \cap c \neq \emptyset$.*

The empty assignment α is a *trivial autarky* for any formula. Clauses of an unsatisfiable formula that are satisfied by an autarky can not be used in any resolution refutation of the formula [KLM06]. Hence, a MUS finder may delete clauses that are satisfied by an autarky [BK09]. Searching for maximal autarkies in order to trim unsatisfiable clause sets has been suggested in [KLM06] and explored in [LS08]. However, finding a maximal autarky is more difficult than finding a single MUS. The authors of [LS08] argue that autarky detection can nevertheless be a useful preprocessing step for algorithms that perform a more difficult task such as finding all MUSes [LS05] or finding the smallest MUS. Autarky detection for redundancy removal is also discussed in [ML11, BLM12]. The interactive MUS finder of [DZK13] implements an algorithm for redundancy removal by autarky identification.

A proof of correctness for the redundancy removal technique of Def. 5.21 can be given in terms of the existence of an autarky for the parts of the resolution graph that are disconnect from the critical clause.

Lemma 5.25. *Given a clause c_i that is critical with respect to the unsatisfiable formula $\mathcal{F}$. Let $U \subseteq \mathcal{F}$ be the set of disconnected clauses removed by the redundancy removal technique defined in Def. 5.21. There exists an autarky α for formula $\mathcal{F}$ that satisfies all clauses in U.*

Proof. *Let $\alpha_i \in A(c_i, \mathcal{F})$, i.e. α_i is a satisfying assignment for $\mathcal{F} \setminus \{c_i\}$. Clearly α_i satisfies all clauses in U, but it is not an autarky for $\mathcal{F}$ as it touches c_i but does not satisfy it. Let α be obtained by removing from α_i all literals whose negation appears in the clauses of $\mathcal{F} \setminus U$, i.e. $\alpha = \alpha_i \setminus \{\neg l \mid c_j \in (\mathcal{F} \setminus U) \text{ and } l \in c_j\}$. We will prove that α satisfies all clauses in U. To obtain a contradiction assume that α does not satisfy some clause $c_k \in U$. Because α_i satisfies c_k there is some $l \in c_k$ such that $l \in \alpha_i$. If $l \notin \alpha$ then there must exists some $c_j \in (\mathcal{F} \setminus U)$ such that $\neg l \in c_j$. But if $\neg l \in c_j$ and $l \in c_k$ then there exists an edge between c_j and c_k in the resolution graph,*

Algorithm 6 Reducing an assignment to an autarky

Given: A formula $\mathcal{F}$ and an assignment α.

1. `while` exists $l \in \alpha$ and $c \in \mathcal{F}$ such that $\neg l \in c$ and $c \cap \alpha = \emptyset$
2. $\quad \alpha = \alpha \setminus \{l\}$
3. `return` α

which contradicts the definition of U. Hence, α satisfies all of U. Also, α touches no clauses in $\mathcal{F} \setminus U$ except for possible clauses containing pure literals, which it satisfies. Hence, α is an autarky for $\mathcal{F}$ that satisfies all clauses in U.

The proof of Lemma 5.25 shows that a subset of the assoc for a critical clause is satisfying all clauses that are disconnected from the critical clause in the resolution graph. Instead of performing the reachability check on the resolution graph we may use Alg. 6 to reduce an assoc into the maximum autarky that is a subset of that assoc, and then remove all clauses satisfied by the assoc.

Lemma 5.26. *Algorithm 6 returns the maximum autarky that is a subset of the assignment it is given as input.*

Proof. *Clearly, the assignment α returned by Alg. 6 satisfies all clauses it touches, hence it is an autarky. It is also maximal, because the algorithm starts from a complete assignment and only deletes literals that are touching unsatisfied clauses.*

Definition 5.27 (Redundancy removal based on Alg. 6). *Given a clause c_i that is critical with respect to the unsatisfiable formula $\mathcal{F}$, and an assoc $\alpha_i \in A(c_i, \mathcal{F})$. Remove from $\mathcal{F}$ any clause satisfied by the autarky $\alpha \subseteq \alpha_i$ obtained as output from Alg. 6 given input α_i.*

Lemma 5.25 proves that this new redundancy removal technique is at least as strong as the redundancy check of Def. 5.21. It is not hard to see that it is in fact strictly stronger, as the autarky may satisfy more clauses than just those disconnected from any critical clause in the resolution graph. Another redundancy check that may be performed is *blocked clause elimination* [Kul99].

Definition 5.28 (Blocked clause [Kul99]). *A clause $c \in \mathcal{F}$ is blocked with respect to literal l if for all $c' \in \mathcal{F}$ such that $\neg l \in c'$ the result of performing resolution $c \otimes c'$ is tautological.*

Corollary 5.29. *A clause $c \in \mathcal{F}$ is blocked with respect to literal l if for all $c' \in \mathcal{F}$ it holds that $L(c, c') \neq \{l\}$.*

Definition 5.30 (Blocked clause elimination). *Redundancy removal by blocked clause elimination is the result of iteratively removing blocked clauses from the formula until none remain.*

The authors of [BJM13] found by experimental evaluation that removing blocked clauses from the input formula did not significantly affect the average performance of their MUS finder. Our MUS finder TARMOMUS, which is discussed in Publication V, maintains the set of edges E_P and thus, because of Corollary 5.29, can easily perform blocked clause elimination on every intermediate formula between the input formula and the output MUS. Our evaluations are consistent with [BJM13] and show that although this repeated blocked clause elimination can be useful for some benchmarks, it is harmful for others.

Although blocked clauses are redundant they may appear in resolution proofs, and these proofs may be considerably shorter than those not using any blocked clauses[8]. As the MUS finder must eventually discover an unsatisfiable subformula that contains no blocked clauses one would expect that their removal can only affect the performance positively, especially as their removal is not guaranteed to happen as a side effect of clause set refinement or unsatisfiable core extraction. However, this is not the case, as even to a MUS finder redundant clauses may be beneficial. For example, redundant clauses can help to avoid assocs that are suboptimal starting points for model rotation. In Example 5.8 it was shown that clause $c_0 \in \mathcal{F}_{fig5.1}$ has an assoc which can not be rotated. This assoc can be avoided by adding the redundant clause $x \vee y \vee z$ to the formula. This type of redundancy addition can be generalized, although one must be cautious when adding redundant clauses that are not redundant with respect to all MUSes of the formula in the solver.

Another form of redundancy addition is inspired by the constructive algorithm of Publication I, and its use in destructive algorithms was proposed in [ML11]. Instead of testing $\mathcal{F} \setminus \{c\}$ for a formula $\mathcal{F}$ that is known to be unsatisfiable, an algorithm may test $(\mathcal{F} \setminus \{c\}) \cup \neg c$. Because $\mathcal{F}$ is unsatisfiable the latter formula is satisfiable if and only if the former is, but as it is more constrained it may be easier to solve. A major downside

[8]The original motivation for the definition of blocked clauses was to generalize the characteristics of clauses created by the *extension rule* of the *extended resolution proof system* [Kul99].

of all redundancy addition techniques is that they will yield clause-set refinement and proof extraction techniques useless when the solver constructs a resolution refutation using the added redundant clauses. It was suggested in [ML11, BLM12] that these are the cases where redundancy checking techniques such as autarky detection could be beneficial.

6. Asynchronous incremental solving using Tarmo

In [WNH09] we introduced TARMO, which at that time was only envisioned to be a special purpose parallel solver for bounded model checking. Two different version of TARMO competed successfully in the Hardware Model Checking Competitions of 2011 and 2012. The competing versions can be seen as parallelizations of the simple BMC algorithm implementation AIGBMC, which was described in the Chapters 2 and 4. TARMO was also discussed in Publication III. In Publication V we generalized the ideas of TARMO, making explicit the notion of *asynchronous* incremental SAT. This is a simple concept that allows combining incremental SAT and parallelism in an application specific manner, as we will discuss in this chapter.

The original version of TARMO, as discussed in [WNH09], was motivated by a particular run time profile often observed for job sequences originating from BMC. As discussed in Chapter 4, BMC problems can always be encoded such that rather than testing the length of a counterexample of exactly k steps, we check the existence of a counterexample of at most k steps. Any sequence of jobs that can be generated using such a BMC encoding will consist either only of unsatisfiable jobs, or start with a finite prefix of unsatisfiable jobs, followed by only satisfiable jobs. It has been observed for the latter type of sequence, that amongst the first satisfiable jobs there is often a job which is significantly easier to solve independently than any of the preceding jobs. Figure 6.1 is an illustration of this behavior taken from Publication II, for the job sequence generated for HWMCC'07 benchmark `bc57sensorsp2neg`.

To explain how to read the illustration given in Fig. 6.1, let us recall that a similar figure already appeared in Chapter 4 as Fig. 4.8. The height of a bar in these two figures denotes the run time of a solver used to solve only the formula induced by that job, without using incremental solving.

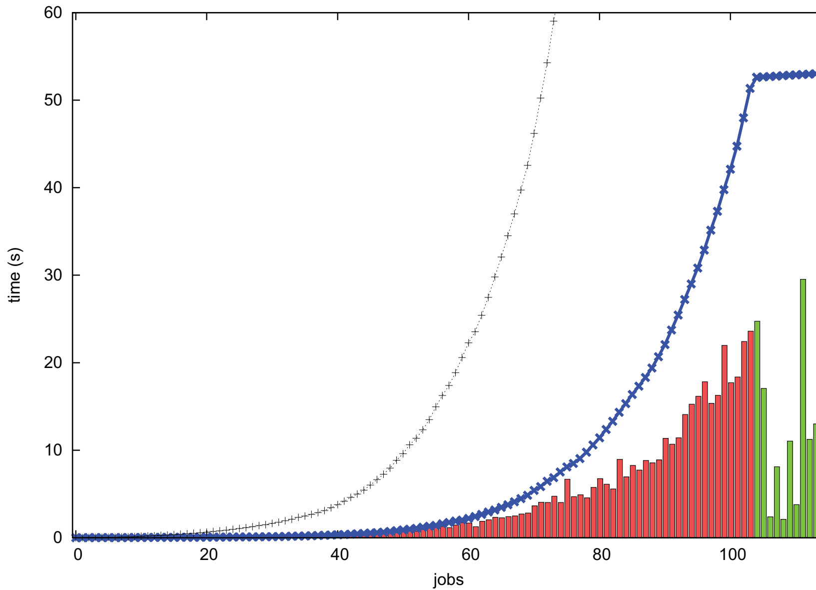


Figure 6.1. Run time behavior for benchmark `bc57sensorsp2neg`.

The thick curve illustrates the behavior of the same solver used incrementally on the sequence of jobs, reporting its total run time each time it proceeded to the next job in the sequence. The dotted curve illustrates the cumulative run time of solving all jobs sequentially and independently.

Note that the top of the bars is always below the thick curve. In other words, unlike the example in Chapter 4 solving one problem k independently consistently takes less time than solving the problems from 0 to k incrementally. Clearly, if we are just interested in proving the existence of a counterexample, rather than finding the minimal counterexample, then simply solving the easy satisfiable job 106 independently would be a fast way to obtain that result. However, this requires knowing in advance at which index an easy satisfiable job resides. A similar run time profile with easy satisfiable jobs following hard unsatisfiable ones was observed for a different application, called automated planning, in [RHN06]. That work proposed a solution in which a set of unsolved jobs from the sequence are solved concurrently, in the hope that the process "leaps over" the difficult jobs, by solving an easy satisfiable job that is several jobs ahead of the last solved unsatisfiable one. However, [RHN06] did not consider the use of incremental solvers, which are crucial for the performance of BMC algorithms. Note that even in cases such as depicted in Fig. 6.1 the incremental solver provides unmatched robust behavior. By this we mean that

if we want to try to solve only the easy jobs individually, then there is not a lot of room for unlucky guesses as to where those easy jobs are before the incremental solver becomes a better choice. The asynchronous solver interface of Publication V provides a way to submit a job sequence to the solver, while leaving the solver the freedom to solve jobs concurrently or out-of-order.

Recall from Chapter 2 that each job in a conventional incremental solver is representing the formula that is made from all problem clauses of that job and all preceding jobs, in conjunction with a set of assumptions that is specific to that job.

$$\mathcal{F}(\phi_i) = \underbrace{\left(\bigcup_{0 \le j \le i} \mathtt{CLS}(\phi_j)\right)}_{\mathtt{CLAUSES}(\phi_i)} \cup \left(\bigcup_{l \in \mathtt{assumps}(\phi_i)} \{l\}\right).$$

The clauses are entered in the solver using the `addClause` function, whereas the assumptions cube that completes a job is passed through the `solve` function. Note that the `addClause` and `solve` functions are part of the interface of a SAT solver, and they control the execution of this particular computer program. The `solve` function is *blocking*, in the sense that the call to this function will not return to the calling application until the SAT solver determines the satisfiability of the job. Hence, as long as a job is not solved the application using the solver can not start to add the clauses for consecutive jobs.

The main idea proposed in Publication V is to extend the solver's interface with a non-blocking version of the `solve` function called `addCube`. Without enforcing the blocking semantics it is possible to think of the solver as a reactive system. The system is given jobs as input and as output it reports the result of solving those jobs. The communication between the application and the solver is *asynchronous*: The application may proceed to submit more jobs while the solver has not yet reported the result for a previously submitted job. Moreover, the results may be reported by the solver out-of-order with respect to the order of the jobs in the input sequence.

6.1 Distribution modes

The asynchronous interface provides a way of giving a sequence of jobs to an incremental solver, in which different strategies for solving jobs from that sequence may be implemented. TARMO is a multi-core solver for incremental SAT problems, which provides the conventional "synchronous" incremental solver interface, as well as the asynchronous interface. Each individual solver thread used inside TARMO is a copy of the solver MINISAT. TARMO divides the sequence of jobs it is given over the available solver threads. We call a strategy for dividing the available jobs over the available solver threads a *distribution mode*. The simplest distribution mode simply gives all jobs to all solver threads. In this way, we obtain a portfolio of incremental solvers. This distribution mode will be referred to as distribution mode `multiconv`, for *multiple conventional*.

Another natural strategy for parallelization within these settings would be to define a distribution mode such that no two solver threads ever work on the same job concurrently. A simple distribution mode that satisfies this constraint is the one in which each solver thread is given the first job from the sequence that has not yet been given to another solver thread. This strategy was named `multijob` in Publication III and Publication V[1] In general, any distribution mode that satisfies the following condition can be easily implemented in TARMO:

Condition 6.1. *A solver thread that has previously worked on job ϕ_i may only be given a job ϕ_j such that* $\texttt{CLAUSES}(\phi_i) \subseteq \texttt{CLAUSES}(\phi_j)$.

The reason this condition must be satisfied is that each of the solver threads is a copy of MINISAT, and thus once clauses have been added to these solver threads they can no longer be removed. Tarmo can use any one of the individual solver threads to solve any subsequence of jobs, and by Condition 6.1 it is also allowed to solve two jobs out of order using the same solver thread if those two jobs share the same clause set.

The third and final distribution mode that is available by default in TARMO is called `multijob+`. It is the same as the `multijob` distribution mode, except that if no new jobs are available then a solver thread will be given the oldest unsolved job that satisfies Condition 6.1, if such a job exists. Note that such an older unsolved job is always a job that another solver thread is also already trying to solve. In this way `multijob+` pro-

[1] In the original BMC-focussed publication [WNH09] it was named `multibound`.

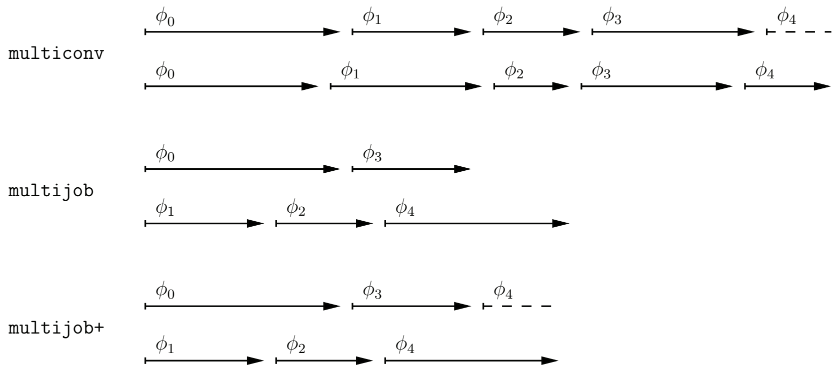


Figure 6.2. Example of all standard distribution modes in TARMO.

vides the distribution mode `multijob` with a fall-back to distribution mode `multiconv`, to prevent idle solvers.

Example 6.2. *In Fig. 6.2 the parallel solving of* 5 *jobs using two solver threads is shown for each one of the three discussed distribution modes. The length of the arrows indicates the time it takes to solve the job. For this example, the lengths have been hand picked and do not correspond to an actual experimental result. In the example the run time for job* ϕ_4 *was chosen smaller for distribution mode* `multiconv` *than for the other two distribution modes. This was done because it corresponds to behavior that will be often seen in practice: Incremental SAT solvers improve performance by reusing information across jobs. Hence, the downside of the* `multijob` *strategy is that each solver thread individually solves less jobs, and thus also gathers less information. In the example, for the* `multijob` *strategy, the job* ϕ_3 *has not been solved by the second solver, which may mean that it misses some information that would have made it solve job* ϕ_4 *faster. Conflict clause clause sharing between solver threads can help to reduce this effect, and will be discussed in Section 6.2.*

The execution of job ϕ_4 *by the first solver thread is drawn using a dotted line for distribution modes* `multiconv` *and* `multijob+`*, because it is interrupted once the second solver thread finishes this last job in the sequence.*

6.2 Conflict clause sharing

Sharing of conflict clauses between solver threads is an important building block in any parallel solver (e.g. [AHJ+12]). Typically this is per-

formed in parallel solvers for single formulas, in other words, between solver threads that all solve the same job, or partitions of the same job. In TARMO multiple solver threads may be solving different jobs concurrently. Hence, care must be taken when employing sharing of learnt clauses between those solver threads. Note that in general a clause c derived while solving a job ϕ_i can be used in the solving process of any job ϕ_j such that $\texttt{CLAUSES}(\phi_i) \subseteq \texttt{CLAUSES}(\phi_j)$.

To achieve correct clause sharing with low overhead the database in TARMO is organized as a set of queues. There is one queue for each unique clause set, i.e. one queue $q(\phi_i)$ for each job ϕ_i such that $\texttt{CLS}(\phi_i) \neq \emptyset$. For jobs ϕ_j such that $\texttt{CLS}(\phi_j) = \emptyset$ we have $q(\phi_j) = q(\phi_i)$ for the largest i such that $i < j$ and $\texttt{CLS}(\phi_i) \neq \emptyset$. If a solver thread wants to share a learnt clause it derived while working at job ϕ_i it pushes it in the corresponding queue $q(\phi_i)$. A solver thread that is solving job ϕ_j can safely introduce any clause that it can find in the queues $q(\phi_i)$ for all $i \leq j$ to its learnt clause database.

For all solver threads s, and all queues q, there exists a pointer $p(s, q)$ that points to the last element in queue q that solver s has seen. If solver s wants to read from queue q it reads the clauses starting from $p(s, q)$, and then updates the pointer to indicate it has now seen all clauses in q. If solver s wants to write to queue q, it must first read all clauses from q that it has not yet read. After this, s writes the new clauses to queue q and updates the pointer. By this simple mechanism each solver thread reads every clause only once, and never reads the clauses it has itself provided to the database. There is no mechanism to avoid duplicates, i.e. the same clause occurring multiple times in the database. Clauses that have been read by all solver threads are deleted from memory.

Example 6.3. *Consider an execution of* TARMO *with two solver threads* s_0 *and* s_1*, using a shared clause database that is in the state given in Fig. 6.3. Assume that solver thread* s_0 *is solving job* ϕ_{21} *and solver thread* s_1 *is solving job* ϕ_{22}*. Both solver threads have already read all clauses that are in queue* $q(\phi_{20})$*. At some point solver thread* s_1 *has accessed queue* $q(\phi_{21})$*, but since then* s_0 *has been writing four new clauses to that queue which have not yet been read by* s_1*. Solver thread* s_0 *has not read any of the clauses from* $q(\phi_{22})$*, and it is also not allowed to do so as long as it is still trying to solve job* ϕ_{21}*.*

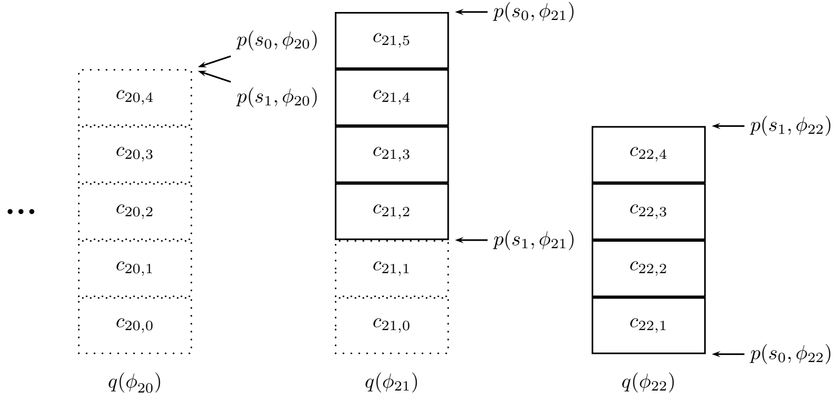


Figure 6.3. Example state of the clause database in TARMO.

6.3 Interactive graphical visualizations

The version of TARMO that was developed along with Publication V provides a graphical interface, to visualize its internal activity. This can be used to study the effect of different distribution modes, and it also proved useful for improving the performance of the tool during its development. The graphical interface of TARMO consists of two windows, one of which visualizes which jobs are run on which solver threads, and a second which visualizes the content of the shared clause database. The visualization is dynamic, in other words it is continuously updated to match the current solver state.

These dynamic visualizations are not as useful when presented statically as a picture, but we nevertheless give one example in Fig. 6.4. The example concerns the state of TARMO after 7.5 seconds of solving benchmark `bc57sensorsp2neg` using four solver threads and distribution mode `multijob`. The top window illustrates the solving history of the four solver threads using four horizontal bars with different colors, similar to the way this was done using arrows in Fig. 6.2. The colors correspond to the result of solving the job on that solver thread, i.e. satisfiable, unsatisfiable or not finished. Further information displayed on the top window concerns the loading of jobs in the solver. The bottom window visualizes the content of the queues of the shared clause database. When used dynamically, it is possible to see from this window how many clauses there are in each queue, how many clauses remain queued in memory, and which clauses have been seen by which individual solver thread.

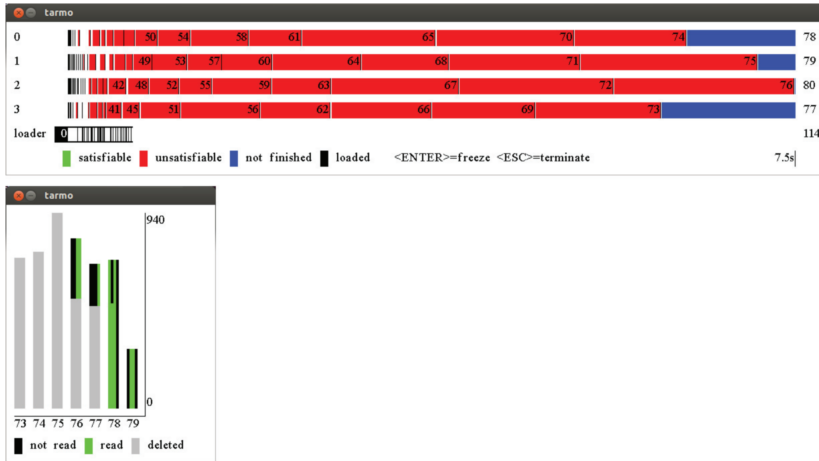


Figure 6.4. Graphical interface of TARMO.

6.4 Applications

Because of the available synchronous interface TARMO can be used to replace any conventional incremental solver, in particular, it can be used as a drop-in replacement for MINISAT. Because only one job can be specified at a time through the synchronous interface the operation of TARMO is in such case limited to the behavior provided by the `multiconv` distribution mode. We showed in Publication V that this can already yield substantial speed-ups for real applications. In many cases further improvements are possible by using the asynchronous interface to create an application specific parallelization. This is illustrated in Publication V for BMC, MUS finding, and a combination of BMC with an implementation of the IC3 algorithm. The application of TARMO in a technique called *Cube and Conquer* is proposed in Publication III and discussed in Chapter 7.

7. Cube and Conquer

To finish first, first you have to finish.
- Juan Manual Fangio

In Publication III a technique called *Cube and Conquer* (C&C) was introduced that aims at solving hard instances of the satisfiability problem. The technique uses a *look-ahead solver* [HvM09], to determine a partitioning of a formula into tens-of-thousands, or even millions of pieces. Each of these partitions is defined by a conjunction of literals, and hence called a cube. Solving of the independent cubes can be performed using a conventional CDCL solver, in either a sequential or parallel fashion. A recent extension called *Concurrent Cube and Conquer* executes the look-ahead and CDCL solver concurrently [vdTHB12]. A state-of-the-art implementation of Concurrent C&C can be found in the solver TREENGELING [Bie13]. The discussion in this chapter is limited to the original version presented in Publication III.

The motivation for this work came from Oliver Kullmann, who generated problems that he could solve by using an ad-hoc combination of a look-ahead solver and a CDCL solver much faster than using either approach independently. Like CDCL solvers, look-ahead solvers are an extension of the classical DPLL procedure [DLL62]. A *look-ahead* on a variable x determines the heuristic quality of x as a decision variable by computing the reduced formulas $\mathcal{F}|_{\{x\}}$ and $\mathcal{F}|_{\{\neg x\}}$. If the computation of the reduced formula for either the literal x or the literal $\neg x$ leads to a conflict then the literal is called a *failed literal*, and it is assigned **false**. A typical look-ahead procedure weighs the clauses that are shortened in length by the assignment, but that do not become satisfied, i.e. the clauses in $\mathcal{F}|_{\{x\}} \setminus \mathcal{F}$ and $\mathcal{F}|_{\{\neg x\}} \setminus \mathcal{F}$. Typically, the variables whose assignment in both polarities cause a large reduction are considered the best decision variables.

The idea behind C&C is that the computationally expensive decision heuristic of the look-ahead solver can be used to create a partitioning on

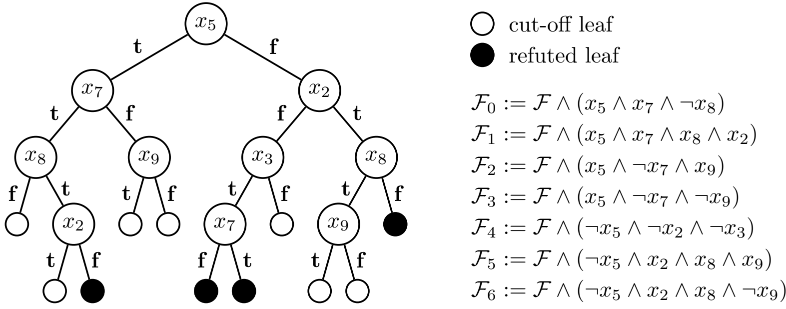


Figure 7.1. Example partitioning.

important variables, after which the CDCL solver can solve the partitions. Use of a look-ahead solver for partitioning had been suggested before in [HJN10]. This earlier work considered splitting the formula into dozens of partitions for parallel solving. C&C on the other hand partitions the formula into thousands or sometimes even millions of pieces, and aims at combining the strong features of both solver architectures. The idea is that the look-ahead solver can break the "hard combinatorial core" of a difficult problem, after which the conflict driven solver is used to solve the localized partitions. For some problems this was shown to outperform either one of the independent approaches, even if the cubes are solved sequentially.

Figure 7.1 illustrates an example partitioning. It shows a binary search tree as explored by the look-ahead solver, and the resulting partitions of the input formula $\mathcal{F}$. Each internal node in the tree corresponds to a decision variable, and arcs labeled "**t**" denote the assignment **true** to the variable, whereas "**f**" denotes the assignment **false**. There are two types of leaf nodes, those corresponding to branches that were refuted because of a conflict, and those for which the *cut-off heuristic* decided that the partition should be solved by a CDCL solver. Observe that there is one partition for each cut-off leaf. Several cut-off heuristics are discussed in [HJN10] and Publication III.

7.1 The weakness of search space splitting

From the point of view of proof complexity theory, splitting a formula into pieces and solving the pieces independently is always a bad idea. For unsatisfiable formulas partitioning can only increase the length of the shortest resolution refutation. This is because a solver that is given the com-

plete formula can chose to refute one of the partitions at a time, whereas solving each of the partitions independently may require creating a refutation for the same subformula multiple times. This is a problem with search space splitting techniques in general [HJN09]. If a formula has two MUSes, and we split the formula such that both partitions contain one MUS, then solving both partitions requires finding a resolution refutation for both MUSes, whereas to prove the whole formula unsatisfiable only one such refutation would have to be found. On the other hand, if we split an unsatisfiable formula on a variable that is outside any MUS, then we end up with two partitions as hard as the original one. This is not just a theoretical problem, but also happens in practice.

Experiment 7.1. *Let $\mathcal{F}_{ph11}$ and $\mathcal{F}_{ph20}$ denote the two unsatisfiable formulas, which represent the pigeon hole problem for 11 and 20 pigeons, respectively. Let the two formulas be generated over disjoint sets of variables, i.e. $Var(\mathcal{F}_{ph11}) \cap Var(\mathcal{F}_{ph20}) = \emptyset$. For this experiment, the satisfiable formula $\mathcal{F}'_{ph20}$ was obtained by removing one arbitrary clause c from $\mathcal{F}_{ph20}$, i.e. $\mathcal{F}'_{ph20} = \mathcal{F}_{ph20} \setminus \{c\}$. Subsequently, the formula $\mathcal{F}$ was obtained by merging $\mathcal{F}_{ph11}$ and $\mathcal{F}'_{ph20}$ into one formula*[1] $\mathcal{F} = \mathcal{F}_{ph11} \cup \mathcal{F}'_{ph20}$.

Clearly, the formula $\mathcal{F}$ is unsatisfiable, and the shortest refutation of $\mathcal{F}$ is equal to the shortest refutation of $\mathcal{F}_{ph11}$. However, given $\mathcal{F}$ as an input to the modified look-ahead solver MARCH_CC *presented in Publication III resulted in 51940 cubes, of which only 7 contained a variable from $Var(\mathcal{F}_{ph11})$. This is due to the heuristic used in the look-ahead solver, which considers variables from $Var(\mathcal{F}_{20})$ as the most important because their assignment causes the largest reduction of $\mathcal{F}$.*

Solving these partitions independently clearly means performing a lot of unnecessary work. When solving $\mathcal{F}$ under cubes d for which $\mathcal{F}'_{ph20}|_d$ is unsatisfiable the solver must create a refutation for one of the unsatisfiable cores $\mathcal{F}_{ph11}$ or $\mathcal{F}'_{ph20}|_d$. There are also thousands of cubes d for which $\mathcal{F}'_{ph20}|_d$ is satisfiable[2]*. To independently solve any one of these partitions the solver must find a refutation for $\mathcal{F}_{ph11}$. This is clearly not an improvement comparing to solving just the original formula, which requires finding the refutation for $\mathcal{F}_{ph11}$ only once.*

Experiment 7.1 considers a carefully constructed bad case example, but it is not simply underlining a theoretical argument. This same behav-

[1]This example formula is available from: `http://www.siert.nl/thesis`
[2]The existence of several thousands such cubes has been confirmed by experiment, the exact number has not been determined.

ior can occur for any unsatisfiable formula with a significant amount of redundant clauses. Redundancy in unsatisfiable formulas is common in real-life problems as illustrated, for example, by the number of MUSes found in recent SAT competition benchmarks [LM13]. Arguably, even this particular example, a conjunction of multiple pigeon hole-like cores, is the encoding of a practical problem: An instance of the pigeon hole problem can represent the impossibility of routing $n + 1$ wires through n channels in an FPGA[3] [ARMS03].

From the start of this project the look-ahead solver was meant to be used to split the hard core of a problem. One could say that if a problem contains a large amount of redundancy, then it clearly is not just a hard core, and thus C&C is simply not the right technique to solve it. The original idea always included using a formula simplifier, before running the look-ahead solver, to remove redundancy and hopefully obtain a single core. In Publication III the simplifier integrated in the solver LINGELING[4] was used for this purpose. For the bad case constructed in Experiment 7.1, this has little effect, as the simplifier does not manage to remove many redundant clauses from this formula. As a result, the total number of cubes generated for the simplified formula is only marginally smaller (51724), while the number of cubes that include a variable from $Var(\mathcal{F}_{ph11})$ remains equally small (7).

7.2 Cube solving phase: Independent, incremental or parallel

In Section 7.1 we ignored the fact that the cubes do not have to be solved completely independently. If an incremental solver is used the information sharing between the solving of the consecutive cubes, then generating the same refutation over and over again should be avoidable. C&C can use the incremental solver by simply placing all clauses in the solver at once, and then submitting the cubes as sets of assumptions. In the early stages of the development of C&C, Marijn Heule discovered that an efficient way to use an incremental solver in this application was through the solver MINISAT with the modifications to read the iCNF file format that we proposed in [WNH09].

Example 7.2. *An incremental job sequence used to represent a C&C partitioning of formula $\mathcal{F}$ will have $\texttt{CLS}(\phi_0) = \mathcal{F}$ and $\texttt{CLS}(\phi_i) = \emptyset$ for all $i > 0$.*

[3] Field-programmable Gate Array, a type of programmable logic device.
[4] `http://fmv.jku.at/lingeling`

We can complete the definition of the 7 *jobs based on the partitions displayed in Fig. 7.1 as follows:*

$$\begin{aligned}
\textit{Let} \quad & \texttt{assumps}(\phi_0) = \{x_5, x_7, \neg x_8\}, \\
\textit{and} \quad & \texttt{assumps}(\phi_1) = \{x_5, x_7, x_8, x_2\}, \\
\textit{and} \quad & \texttt{assumps}(\phi_2) = \{x_5, \neg x_7, x_9\}, \\
\textit{and} \quad & \texttt{assumps}(\phi_3) = \{x_5, \neg x_7, \neg x_9\}, \\
\textit{and} \quad & \texttt{assumps}(\phi_4) = \{\neg x_5, \neg x_2, \neg x_3\}, \\
\textit{and} \quad & \texttt{assumps}(\phi_5) = \{\neg x_5, x_2, x_8, x_9\}, \\
\textit{and} \quad & \texttt{assumps}(\phi_6) = \{\neg x_5, x_2, x_8, \neg x_9\}.
\end{aligned}$$

For a formula like the one constructed for Experiment 7.1, the first cube d for which $\mathcal{F}'_{ph20}|_d$ is satisfiable will require the solver to prove the unsatisfiability of $\mathcal{F}_{ph11}$. Because there are no clauses that link the variables of subformula $\mathcal{F}_{ph11}$ and subformula $\mathcal{F}'_{ph20}$ the solver's final conflict will be the empty clause if cube d contained no variables from $Var(\mathcal{F}_{11})$. Once the solver has derived an empty final conflict clause it clearly does not need to proceed to solve any future jobs.

Although information sharing between the solving processes for different cubes can be beneficial and even crucial, it may be seen as a means of compensating for a failed partitioning. An ideal partitioning function will provide such localized partitions that information sharing between the solver process is unnecessary, allowing trivial parallelization by solving multiple cubes concurrently. As sharing remains desirable in practice, TARMO, with its asynchronous interface discussed in Chapter 6, provides exactly the right features for parallelizing the solving of the cubes.

Although Publication III discusses the use of TARMO for the technique, it was not used for the empirical evaluation in that Publication. TARMO was at the time outperformed by a solver called ILINGELING, a version of the solver LINGELING, modified to read iCNF files especially for this application. The version of TARMO that was available at the time used an excessive amount of memory for C&C, because it was not implemented to deal with tens-of-thousands of small jobs. The new implementation of TARMO discussed in Publication V was developed with this experience in mind, as a result its performance for C&C was greatly improved. More importantly, these were crucial steps towards making TARMO a true multipurpose tool.

As was shown in Publication III, C&C works well for some hard benchmarks. Even if solving a hard problem by splitting it in many easy pieces

is not the most elaborate strategy, it is a strategy that can enable eventually solving problems that would otherwise be too large to handle. Intuitively speaking, it can prevent a solver from "choking", instead allowing it to continuously make progress by solving small partitions one by one. This line of thought is the reason for the quote, which originates from car racing, at the beginning of this chapter.

The recent work on Concurrent C&C [vdTHB12], where partitions are dynamically partitioned further when they are found to be difficult for the CDCL solver, is a natural extension of this technique. Although this technique has not matured yet, it does have the potential to become one of several powerful complementary SAT solving techniques. This is underlined by the excellent performance of the Concurrent C&C implementation TREENGELING in the most recent SAT competition[5].

[5] http://www.satcompetition.org/2013

8. Concurrent Clause Strengthening

This chapter discusses a technique called *Concurrent Clause Strengthening*, which was recently proposed in Publication VI. This technique provides a novel way of exploiting the availability of multi-core hardware in SAT solvers.

Although a lot of research effort has been invested in the development of parallel SAT solvers (e.g. [BS96, ZBH96, HJS09, HJN11]), their performance remains relatively modest. Limitations of the two most common approaches to parallel solving, the *portfolio* and *search space splitting* approaches, have been identified [HJN09]. Limitations to the parallelizability of resolution as a proof system are discussed in [KSSS13]. These results do not imply that applications of SAT solvers can not yet fully benefit from the availability of multi-core and multi-processor hardware. As we have noted before, independent concurrent execution of subtasks is common practice in, for example, industrial applications of model checking [SEMB11]. Asynchronous incremental satisfiability, as proposed in Publication V and discussed in Chapter 6, provides an intermediate level between completely independent concurrent execution, and parallelized solving of single formulas.

An important observation is that it is the parallelization of the *search* performed by a SAT solver that is proving to be difficult to achieve. Modern SAT solvers interleave search with several additional reasoning procedures. A recent example is *inprocessing* [JHB12], but the more established *conflict clause strengthening* procedures [ES05, SB09, VG11] also belong to this category. Performing such additional reasoning in parallel with search provides an alternative way of using concurrency in a SAT solver. In Publication VI we proposed the *solver-reducer architecture*, which implements concurrent conflict clause strengthening. We provided an empirical evaluation of its performance used in combination with two

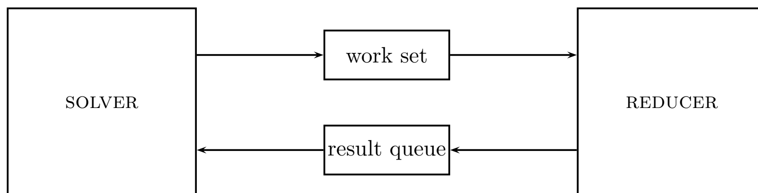


Figure 8.1. The solver-reducer architecture.

different solvers, MINISAT and GLUCOSE [AS09].

8.1 The solver-reducer architecture

The solver-reducer architecture uses two concurrently executing threads, which are called the SOLVER and the REDUCER. The SOLVER acts like any conventional SAT solver, except for its interaction with the REDUCER. The interaction between the SOLVER and the REDUCER is limited to passing clauses through two shared-memory data structures called the *work set* and the *result queue*. The work set is used to pass clauses from the SOLVER to the REDUCER, the result queue is used for passing clauses in the opposite direction, as illustrated in Fig. 8.1.

Whenever the SOLVER learns a clause it writes a copy of that clause to the work set. The REDUCER reads clauses from the work set and tries to strengthen them, in other words given a clause c such that $\mathcal{F} \models c$ it tries to find a clause $c' \subset c$ for which $\mathcal{F} \models c'$ still holds. When the REDUCER successfully reduces the length of a clause, it places the new shorter clause in the result queue. The SOLVER checks frequently whether there are any clauses in the result queue. If this is the case the SOLVER enters the clauses from the result queue in its learnt clause database.

This technique was shown to yield a consistent reduction of wall clock time for unsatisfiable formulas in Publication VI. Wall clock time is defined as the amount of time that passes from the start to the finish of the solving process, and this measure is independent of the amount of resources that are used during that time. CPU time on the other hand is the sum of the time spend by each of the cores used, i.e. if a single program uses all the computation power of two CPU cores concurrently then the CPU time grows twice as fast as the wall clock time. We showed that our technique can also yield a reduction of the average amount of CPU time required for solving unsatisfiable formulas.

8.2 Employing concurrency versus parallelization

One may consider a parallelization of an algorithm as a strategy for assigning any number of simultaneously available computation resources to performing a single task. By that definition Publication VI does not present a parallelization of a SAT solver, as only the use of exactly two concurrent computation threads is considered. However, existing techniques for parallelizing SAT algorithms can be used in combination with our two threaded solver, in order to obtain a generic parallelization.

During the development of the architecture we did consider employing more than two computation threads directly, for example by using multiple REDUCER threads for one SOLVER. A problem with running multiple concurrent REDUCER threads is that each of them will be individually weaker than one single REDUCER, unless they all operate on the same set of learnt clauses, which would be difficult to implement efficiently. As typically the REDUCER can not handle all the conflict clauses derived by one SOLVER, running multiple SOLVER threads with one REDUCER does not seem sensible. Eventually, we decided to focus on the basic technique.

The algorithm used for conflict clause strengthening in our implementation of the REDUCER is very similar to the *vivification algorithm* [PHS08] for strengthening problem clauses. Observe that the REDUCER's algorithm can be made as computationally cheap or expensive as desired. Within the implementation discussed in Publication VI the balance between the computational load of the SOLVER and the REDUCER can be easily shifted. For example, the REDUCER can be made weaker, but faster, by reducing the maximum size of its learnt clause database. On the other hand, its algorithm can be made stronger, but slower, by either increasing the size of this database, or even by allowing the REDUCER to assign some variables by branching decisions if simply assigning all literals in the input clause to **false** does not lead to a conflict. As an extreme example one may even consider running multiple REDUCER threads recursively, i.e. run a REDUCER on the learnt clauses of another REDUCER.

8.3 Applications and competitions

One of the strengths of the solver-reducer architecture is that it maintains the original interface of the SAT solver, and thus this technique can be employed in any solver application without further modifications.

Niklas Eén has integrated our solver-reducer solvers into his ZZ model checking environment[1]. For the PDR implementation TREBUCHET in ZZ these solvers provide no performance gains. This is not surprising, given the solver usage of PDR, which is characterized by an extremely small number of conflicts per job, as we discussed in Section 4.7. However, preliminary experiments suggest that the performance of the BMC implementation inside ZZ can benefit from the application of concurrent clause strengthening.

The solvers MINIRED and GLUCORED, based on MINISAT and GLUCOSE, respectively, served to illustrate the performance improvements over their respective base solvers in Publication VI. The number of modifications with respect to those base solvers was kept as small as possible, and the solvers were not extensively tuned. The relatively poor performance of GLUCORED at the SAT competition 2013[2] can be partially explained by this lack of tuning. Another problem is that it is missing a mechanism for deleting a clause from the SOLVER's learnt clause database whenever a reduced version of that clause is provided by the REDUCER. This is particularly problematic given the long execution times allowed in the competition (up to 5000 seconds). Although this feature would not be difficult to implement, it was originally left out for simplicity, and because it is not necessary for MINIRED, as we explained in Publication VI.

Answer Set Programming (ASP) is a form of declarative programming. There exist dedicated solvers for ASP (e.g. [GKNS07]), but an ASP program can also be translated to a single instance of SAT (e.g. [JN11]). Tomi Janhunen et al. submitted several tool-chains to the ASP competition of 2013[3]. One of these, named LP2SOLRED-MT, employed our two threaded solver GLUCORED to solve the SAT encoding of an ASP problem. The other tool-chain, named LP2SAT-MT, was identical except that it employed the parallel solver PLINGELING[4] version 'al6' with 6 threads. The version using GLUCORED solved 574 benchmarks, whereas the version using PLINGELING solved only 495.

This natural approach to applying concurrency in SAT solvers is the first of a kind. The illustrated efficiency shows that it holds a promise for the future. To deliver on this promise, more research into which features of this technique contribute most to its efficiency is required.

[1] http://bitbucket.org/niklaseen
[2] http://www.satcompetition.org/2013
[3] http://www.mat.unical.it/aspcomp2013
[4] http://fmv.jku.at/lingeling

9. Conclusions

This dissertation studies modern SAT solvers in real-life applications, with a focus on incremental solver usage and parallelism. The dissertation consists of six publications and this unifying introduction.

The scientific content of this dissertation begins in Chapter 3, where a visualization of incremental SAT solver behavior from Publication II is discussed. This discussion is followed by the presentation of a new visualization of such behavior, called the clause involvement visualization. Chapter 4 discusses model checking, a prominent formal verification technique. We explain basic model checking concepts in order to facilitate a study of the behavior of incremental solvers in these applications. The proposed visualizations support this study, in which behavior observable from such figures is related to known properties of the problems they represent. Furthermore, observations on the behavior of incremental solvers used inside the recent IC3 and PDR algorithms provide insights that underline the need for future research into efficient solving strategies for this application.

Chapter 5 discusses an application of incremental SAT solvers called MUS finding, which is also the subject of Publication I and Publication IV. The discussion provided in Chapter 5 focuses on providing insight in the inner working of existing algorithms. To this aim we provide a discussion on the design of such algorithms, the solver behavior they induce, and extensions of these algorithms using extra redundancy removal techniques. Furthermore, we extend a theory from Publication IV, and prove a conjecture from [BLM12].

The use of parallelism plays a major role in this dissertation, throughout the publications, and starting from Chapter 6 of this unifying introduction. In that chapter we discuss asynchronous incremental SAT solving, which we proposed in Publication V. This technique is a simple and nat-

ural extension of the most commonly used incremental solver interface, the assumptions interface. It provides a means of combining incremental solving with parallel solving, and eases implementation of application specific parallelizations. Asynchronous incremental solving provides an alternative between parallelizing the solving of a single formula, and parallel execution of independent subtasks.

The Cube and Conquer technique discussed in Chapter 7, and presented in Publication III, can be seen in several ways: It is a technique for faster solving, an application of incremental SAT solvers, and a method for parallel SAT solving. It is based around splitting a formula into pieces and solving the pieces individually, but it is significantly different from previously proposed search space splitting techniques. Such techniques typically aim to give multiple concurrent solver threads an equal share of work, whereas this technique instead aims at using the strength of look-ahead solvers to break the hard core of a problem into tens-of-thousands of highly localized pieces. The pieces are subsequently solved using one or multiple conventional CDCL solvers.

In Chapter 8 we discuss the solver-reducer architecture that was presented in Publication VI. It shows that using concurrency in a SAT solver is not limited to parallelization of the solver's search. Instead, one may use concurrency to aid a conventional CDCL search procedure. A solver using the proposed solver-reducer architecture performs conflict clause strengthening in parallel with a conventional CDCL search procedure. In Publication VI this was shown to yield a consistent performance improvement for solving of unsatisfiable formulas. The solvers build using this architecture can replace any conventional solver in any application. Moreover, concurrent clause strengthening is just one instantiation of the general idea of applying concurrency in SAT solvers without parallelizing their search.

As a whole, this dissertation aims to provide insight into key elements of SAT solvers in practical applications. This work is motivated by the believe that future improvements in our ability to solve computationally hard problems depend crucially on our understanding of the current technology.

Bibliography

[AHJ+12] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting Clause Exchange in Parallel SAT Solving. In Cimatti and Sebastiani [CS12], pages 200–213.

[ARMS03] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[AS09] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI), Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In Rance Cleaveland, editor, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), held as part of the European Joint Conferences on the Theory and Practice of Software (ETAPS), Amsterdam, The Netherlands, March 22-28, 1999*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BHJ+06] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science*, 2(5:5):1–64, 2006.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bie08] Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):75–97, 2008.

[Bie09] Armin Biere. Bounded Model Checking. In Biere et al. [BHvMW09], pages 457–481.

[Bie13] Armin Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In *Proceedings of SAT Competition 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 51–52. University of Helsinki, 2013.

[BJM13] Anton Belov, Matti Järvisalo, and João Marques Silva. Formula Preprocessing in MUS Extraction. In Nir Piterman and Scott A. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS), Rome, Italy, March 16-24, 2013*, volume 7795 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2013.

[BK09] Hans Kleine Büning and Oliver Kullmann. Minimal Unsatisfiability and Autarkies. In Biere et al. [BHvMW09], pages 339–401.

[BLM12] Anton Belov, Inês Lynce, and João Marques Silva. Towards efficient MUS extraction. *AI Communications*, 25(2):97–116, 2012.

[BM10] Robert K. Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Proceedings of the 22th International Conference on Computer Aided Verification (CAV), Edinburgh, Scotland, UK, July 15-19, 2010*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.

[BM11] Anton Belov and João Marques Silva. Accelerating MUS extraction with recursive model rotation. In Bjesse and Slobodová [BS11], pages 37–40.

[Bra11] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Austin, Texas, USA, January 23-25, 2011*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[Bra12] Aaron R. Bradley. Understanding IC3. In Cimatti and Sebastiani [CS12], pages 1–14.

[BS96] Max Böhm and Ewald Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.

[BS11] Per Bjesse and Anna Slobodová, editors. *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, Texas, USA, October 30 - November 2, 2011*. FMCAD Inc., 2011.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard version 2.0, 2010. Available from: http://www.smtlib.org.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing STOC, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[Cra57] William Craig. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[CS12] Alessandro Cimatti and Roberto Sebastiani, editors. *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT), Trento, Italy, June 17-20, 2012*, volume 7317 of *Lecture Notes in Computer Science*. Springer, 2012.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[dSNP88] J. L. de Siqueira N. and Jean-Francois Puget. Explanation-Based Generalisation of Failures. In Yves Kodratoff, editor, *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI), Munich, Germany, August 1-5, 1988*, pages 339–344. Pitmann Publishing, 1988.

[DZK13] Johannes Dellert, Christian Zielke, and Michael Kaufmann. MUStICCa: MUS Extraction with Interactive Choice of Candidates. In Järvisalo and Van Gelder [JV13], pages 408–414.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Bjesse and Slobodová [BS11], pages 125–134.

[ES03a] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Selected Revised Papers of 6th International Conference on Theory and Applications of Satisfiability Testing (SAT), Santa Margherita Ligure, Italy, May 5-8, 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[ES03b] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Proceedings of First International Workshop on Bounded Model Checking (BMC)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560, 2003.

[ES05] Niklas Eén and Niklas Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization. Poster presented at the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT), St. Andrews, UK, June 19-23, 2005. Available from: http://www.minisat.se.

[GKNS07] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-Driven Answer Set Solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 386–392. AAAI Press/The MIT Press, 2007.

[GMP08] Éric Grégoire, Bertrand Mazure, and Cédric Piette. On Approaches to Explaining Infeasibility of Sets of Boolean Clauses. In *Proceedings of the 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Dayton, Ohio, USA, November 3-5, 2008*, volume 1, pages 74–83. IEEE Computer Society, 2008.

[HHW13] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying Refutations with Extended Resolution. In Maria Paola Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction (CADE), Lake Placid, NY, USA, June 9-14, 2013*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013.

[HJL05] Keijo Heljanko, Tommi A. Junttila, and Timo Latvala. Incremental and Complete Bounded Model Checking for Full PLTL. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV), Edinburgh, Scotland, UK, July 6-10, 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2005.

[HJN09] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning Search Spaces of a Randomized Search. In Roberto Serra and Rita Cucchiara, editors, *Proceedings of Emergent Perspectives in Artificial Intelligence, XIth International Conference of the Italian Association for Artificial Intelligence (AI*IA), Reggio Emilia, Italy, December 9-12, 2009*, volume 5883 of *Lecture Notes in Computer Science*, pages 243–252. Springer, 2009.

[HJN10] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT Instances for Distributed Solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Yogyakarta, Indonesia, October 10-15, 2010*, volume 6397 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2010.

[HJN11] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Grid-Based SAT Solving with Iterative Partitioning and Clause Learning. In Jimmy Ho-Man Lee, editor, *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP), Perugia, Italy, September 12-16, 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 385–399. Springer, 2011.

[HJS09] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: A Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6(4):245–262, 2009.

[HLSB06] Fred Hemery, Christophe Lecoutre, Lakhdar Sais, and Frédéric Boussemart. Extracting MUCs from Constraint Networks. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI), Riva del Garda, Italy, August 29 - September 1, 2006*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 113–117. IOS Press, 2006.

[HM12] Antti Eero Johannes Hyvärinen and Norbert Manthey. Designing Scalable Parallel SAT Solvers. In Cimatti and Sebastiani [CS12], pages 214–227.

[Hoo93] John N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(1&2):177–186, 1993.

[HvM09] Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In Biere et al. [BHvMW09], pages 155–184.

[JHB12] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR), Manchester, UK, June 26-29, 2012*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012.

[JN11] Tomi Janhunen and Ilkka Niemelä. Compact Translations of Non-disjunctive Answer Set Programs to Propositional Clauses. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2011.

[Jun04] Ulrich Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the 19th National Conference on Artificial Intelligence, 16th Conference on Innovative Applications of Artificial Intelligence, San Jose, California, USA, July 25-29, 2004*, pages 167–172. AAAI Press / The MIT Press, 2004.

[JV13] Matti Järvisalo and Allen Van Gelder, editors. *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT), Helsinki, Finland, July 8-12, 2013*, volume 7962 of *Lecture Notes in Computer Science*. Springer, 2013.

[KJN12] Roland Kindermann, Tommi A. Junttila, and Ilkka Niemelä. SMT-Based Induction Methods for Timed Systems. In Marcin Jurdzinski and Dejan Nickovic, editors, *Proceedings of the 10th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS), London, UK, September 18-20, 2012*, volume 7595 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2012.

[KLM06] Oliver Kullmann, Inês Lynce, and João Marques Silva. Categorisation of Clauses in Conjunctive Normal Forms: Minimally Unsatisfiable Sub-clause-sets and the Lean Kernel. In Armin Biere and Carla P. Gomes, editors, *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT), Seattle, WA, USA, August 12-15, 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 22–35. Springer, 2006.

[KS03] Daniel Kroening and Ofer Strichman. Efficient Computation of Recurrence Diameters. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), New York, NY, USA, January 9-11, 2003*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003.

[KSSS13] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and Parallelizability: Barriers to the Efficient Parallelization of SAT Solvers. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the 27th AAAI Conference on Artificial Intelligence, Bellevue, Washington, USA, July 14-18, 2013*. AAAI Press, 2013. Available from: http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6421.

[Kul99] Oliver Kullmann. New Methods for 3-SAT Decision and Worst-case Analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

[LB13] Jean-Marie Lagniez and Armin Biere. Factoring Out Assumptions to Speed Up MUS Extraction. In Järvisalo and Van Gelder [JV13], pages 276–292.

[Lib05] Paolo Liberatore. Redundancy in logic I: CNF propositional formulae. *Artificial Intelligence*, 163(2):203–232, 2005.

[LM13] Mark H. Liffiton and Ammar Malik. Enumerating Infeasibility: Finding Multiple MUSes Quickly. In Carla P. Gomes and Meinolf Sellmann, editors, *Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR), Yorktown Heights, NY, USA, May 18-22, 2013*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2013.

[LS05] Mark H. Liffiton and Karem A. Sakallah. On Finding All Minimally Unsatisfiable Subformulas. In Fahiem Bacchus and Toby Walsh, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT), St. Andrews, UK, June 19-23, 2005*, volume 3569 of *Lecture Notes in Computer Science*, pages 173–186. Springer, 2005.

[LS08] Mark H. Liffiton and Karem A. Sakallah. Searching for Autarkies to Trim Unsatisfiable Clause Sets. In Hans Kleine Büning and Xishun Zhao, editors, *Proceedings of the 11th International Conference of Theory and Applications of Satisfiability Testing (SAT), Guangzhou, China, May 12-15, 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 182–195. Springer, 2008.

[Mar10] João Marques Silva. Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper). In *IEEE International Symposium on Multiple-Valued Logic (ISMVL)*, pages 9–14. IEEE Computer Society, 2010.

[McM93] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[McM03] Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV), Boulder, CO, USA, July 8-12, 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[MFM04] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. ZChaff2004: An Efficient SAT Solver. In Holger H. Hoos and David G. Mitchell,

editors, *Revised Selected Papers of 7th International Conference on Theory and Applications of Satisfiability Testing (SAT), Vancouver, BC, Canada, May 10-13, 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer, 2004.

[MJB13] João Marques Silva, Mikolás Janota, and Anton Belov. Minimal Sets over Monotone Predicates in Boolean Formulae. In Natasha Sharygina and Helmut Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia, July 13-19, 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 592–607. Springer, 2013.

[ML11] João Marques Silva and Inês Lynce. On Improving MUS Extraction Algorithms. In Sakallah and Simon [SS11], pages 159–173.

[MMZ$^+$01] 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, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287 – 295, 1985.

[MS96] João Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, November 10-14, 1996*, pages 220–227. ACM and IEEE Computer Society, 1996.

[MSSS12] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Parallel SAT Solver Selection and Scheduling. In Michela Milano, editor, *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP), Québec City, QC, Canada, October 8-12, 2012*, volume 7514 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2012.

[NR12] Alexander Nadel and Vadim Ryvchin. Efficient SAT Solving under Assumptions. In Cimatti and Sebastiani [CS12], pages 242–255.

[NRS13] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS Extraction with Resolution. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Portland, OR, USA, October 20-23, 2013*, pages 197–200, 2013.

[PHS08] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. Vivifying Propositional Clausal Formulae. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI), Patras, Greece, July 21-25, 2008*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529. IOS Press, 2008.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*

(FOCS), Providence, Rhode Island, USA, 31 October - 1 November 1977, pages 46–57. IEEE Computer Society, 1977.

[PW88] Christos H. Papadimitriou and David Wolfe. The Complexity of Facets Resolved. *Journal of Computer and System Sciences*, 37(1):2–13, 1988.

[RHN06] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artifical Intelligence*, 170(12-13):1031–1080, 2006.

[SB09] Niklas Sörensson and Armin Biere. Minimizing Learned Clauses. In Oliver Kullmann, editor, *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), Swansea, UK, June 30 - July 3, 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[SB11] Fabio Somenzi and Aaron R. Bradley. IC3: Where monolithic and incremental meet. In Bjesse and Slobodová [BS11], pages 3–8.

[SEMB11] Baruch Sterin, Niklas Eén, Alan Mishchenko, and Robert Brayton. The Benefit of Concurrency in Model Checking. In *Proceedings of the 20th International Workshop on Logic and Synthesis (IWLS), San Diego, CA, June 3 - 5, 2011*, pages 176–182, 2011.

[Sin07] Carsten Sinz. Visualizing SAT Instances and Runs of the DPLL Algorithm. *Journal of Automated Reasoning*, 39(2):219–243, 2007.

[SLB09] Tobias Schubert, Matthew D. T. Lewis, and Bernd Becker. PaMiraXT: Parallel SAT Solving with Threads and Message Passing. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6(4):203–222, 2009.

[SS11] Karem A. Sakallah and Laurent Simon, editors. *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT), Ann Arbor, MI, USA, June 19-22, 2011*, volume 6695 of *Lecture Notes in Computer Science*. Springer, 2011.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Proceedings of the 3th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, Texas, USA, November 1-3, 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[VG11] Allen Van Gelder. Generalized Conflict-Clause Strengthening for Satisfiability Solvers. In Sakallah and Simon [SS11], pages 329–342.

[vdTHB12] Peter van der Tak, Marijn Heule, and Armin Biere. Concurrent Cube-and-Conquer, poster presentation. In Cimatti and Sebastiani [CS12], pages 475–476.

[Wie13] Siert Wieringa. Some notes on model rotation. *arXiv.org online e-print service*, 2013. Available from: http://arxiv.org/abs/1308.2142.

[WKS01] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Proceedings of the 38th Design Automation Conference (DAC), Las Vegas, NV, USA, June 18-22, 2001*, pages 542–545. ACM, 2001.

[WNH09] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. Tarmo: A Framework for Parallelized Bounded Model Checking. In Lubos Brim and Jaco van de Pol, editors, *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC), Eindhoven, The Netherlands, November 4, 2009*, volume 14 of *Electronic Proceedings in Theoretical Computer Science*, pages 62–76, 2009.

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

A. SMV model for Example 4.10

```
MODULE main
VAR
  x0 : boolean;
  x1 : boolean;
  x2 : boolean;
  y0 : boolean;
  y1 : boolean;
  y2 : boolean;
  y3 : boolean;

ASSIGN
  init(y0):=FALSE;
  next(y0):=!max & (y0 xor yc0);
  next(x0):=x0 xor xc0;
  init(y1):=FALSE;
  next(y1):=!max & (y1 xor yc1);
  next(x1):=x1 xor xc1;
  init(y2):=FALSE;
  next(y2):=!max & (y2 xor yc2);
  next(x2):=x2 xor xc2;
  init(y3):=FALSE;
  next(y3):=!max & (y3 xor yc3);
```

```
DEFINE
  yc0:=TRUE;
  xc0:=TRUE;
  yc1:=y0 & yc0;
  xc1:=x0 & xc0;
  yc2:=y1 & yc1;
  xc2:=x1 & xc1;
  yc3:=y2 & yc2;
  max:=x0 & x1 & x2 & TRUE;

CTLSPEC AG(!y3);
```

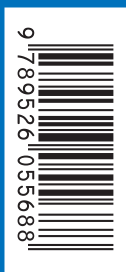
B. Errata for the publications

- Publication I: A reference to [HLSB06] should have appeared after the words 'recent work' in the first sentence of the related work section. Note that the 'latter work' referred to in the sentence following the missing reference is also supposed to refer to [HLSB06].

- Publication IV: The description of the algorithmic improvement of model rotation does not describe that the initialization of the array 'seen' is performed before *every* non-recursive call to the model rotation subroutine. See Section 5.4 of this document for an improved algorithmic description.

Aalto-DD 20/2014

This dissertation studies modern SAT solvers in real-life applications, with a focus on incremental solver usage and parallelism. It aims to provide insight into key elements of SAT solvers in practical applications. The work is motivated by the believe that future improvements in our ability to solve computationally hard problems depend crucially on our understanding of the current technology.



ISBN 978-952-60-5568-8
ISBN 978-952-60-5569-5 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS +
ECONOMY

ART +
DESIGN +
ARCHITECTURE

SCIENCE +
TECHNOLOGY

CROSSOVER

DOCTORAL
DISSERTATIONS