Core algorithms for SAT and SAT-related problems

Vadim Ryvchin

Core algorithms for SAT and SAT-related problems

Research Thesis

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Vadim Ryvchin

Submitted to the Senate of
the Technion — Israel Institute of Technology
Shebat 5774 Haifa January 2014

This research thesis was done under the supervision of Associate Professor Ofer Strichman in the Faculty of Industrial Engineering and Management.

I wish to express my sincere gratitude to my supervisor, Associate Professor Ofer Strichman, for his guidance and kind support. In addition to thesis guidance I have learnt from Ofer analytical thinking, presentation and teaching skills. His patience, encouragement, and immense knowledge were key motivations throughout my PhD. Also I would like to thank my family for their faith and support all the way.

The generous financial help of Technion Israel Institute of Technology is gratefully acknowledged

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Boolean Satisfiability (SAT) is the canonical NP-complete problem, and has numerous practical applications. This thesis focuses on three main topics related to Conflict-Driven Clause Learning (CDCL) SAT technology: core heuristics of SAT solving, Incremental SAT Solving, and Minimal Unsatisfiable Core extraction. All the suggested algorithms were implemented and tested with hundreds of public benchmarks, which proved their effectiveness.

As an example of the techniques developed as part of the thesis, consider the problem of minimal unsatisfiable core extraction. A variety of tasks in formal verification require finding small or minimal unsatisfiable cores (unsatisfiable subsets of the original set of clauses). As a result, MUS extraction algorithms are currently a very active area of research. We provide several optimizations to well- known algorithms and new ideas for modifications. Several application (perhaps even most) require to minimize the High-Level Unsatisfiable Core (HLMUC), which means that what needs to be minimized is not the number of values that participate in the proof, rather the number of pre-defined sets of constraints that participate in the proof. In the thesis we propose seven heuristic improvements to the state-of-the-art which together result in an overall reduction of 55% in run time and 73% in the size of the resulting core, based on our experiments with hundreds of industrial test cases. Our work on optimizations for MUC and HLMUC culminated in the best known MUS and HLMUC solvers today: the solvers Haifa-MUC and Haifa-HLMUC, which were developed as part of this thesis, won the gold medals in the last annual competition for the fastest core- and high-level core extraction engine. The thesis is a collection of six published articles, with a joint introduction and summary.

Abbreviations and Notations

Notation	**Explanation**
SAT Solver	Boolean Satisfiability solver
DPLL	DavisPutnamLogemannLoveland algorithm
CDCL	Conict-Driven Clause Learning
CNF	Conjunctive Normal Form
SAT	Satisfiable
UNSAT	Unsatisfiable
BCP	Boolean Constraint Propagation
BMC	Bounded Model Checking
SMT	Satisfiability Modulo Theories
UC	Unsatisfiable Core
MUC	Minimal Unsatisfiable Core
MUS	Minimal Unsatisfiable Subformula
HLMUC	High-Level Unsatisfiable Core
GMUS	Group Minimal Unsatifiable Subset/Subformula/Set
CM	Clause-based Multiple instances
LM	Literal-based Single instance
LSS	Literal-based with Step look-ahead
CLMS	Multiple instances Clause/Literal-based with Step look-ahead
T2P	algorithm for transforming Temporary clauses To Pervasive clauses
IC	Interesting Constraints
sec.	seconds

Chapter 1

Introduction

Boolean Satisfiability (SAT) is the problem of determining the existence of variables assignment which satisfies a given Boolean formula. SAT is a classic and the first known NP-complete problem [26] which has numerous applications in many practical problems like formal verification [17,19,49,59,73,96], planning [44, 74], bioinformatics [56], and combinatorics [8, 97]. The performance of SAT Solvers has improved tremendously during the last decade and the research in this area continues to be very active. This thesis focuses on algorithms for solving three SAT-related problems: SAT, Incremental SAT, and extraction of Minimal Unsatisfiable Cores (MUC) and High-Level Minimal Unsatisfiable Cores (HLMUC).

1.1 SAT Solving

The SAT problem consists of determining a satisfying variable assignment for a Boolean formula φ or proving that no such assignment exists. In case such assignment exists we refer to a formula φ as satisfiable (SAT), and otherwise as unsatisfiable (UNSAT). Let $V = \{v_1, v_2, \ldots\}$ denote Boolean variables. A literal l_i is either a variable v_i or its negation $\neg v_i$, for $i \geq 1$. All propositional formulas in this thesis are represented in Conjunctive Normal Form (CNF). A CNF formula φ consists of a conjunction of clauses, each of which consists of a disjunction of literals. A CNF formula can also be viewed as a set of clauses, and each clause c can be viewed as a set of literals. The

representation used will be clear from the context.

Example 1.1.1 *(CNF Formula). An example of a CNF formula is:*

$$\varphi = (v_1 \vee \neg v_2) \wedge (v_2 \vee v_3) \wedge (v_2 \vee \neg v_4) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3 \vee \neg v_4)$$

The alternative set representation is:

$$\varphi = \{\{v_1, \neg v_2\}, \{v_2, v_3\}, \{v_2, \neg v_4\}, \{\neg v_1, \neg v_2, \neg v_3, \neg v_4\}\}$$

In this thesis we refer only to Conflict-Driven Clause Learning (CDCL) solvers operating on CNF formulas. All competitive SAT solvers these days belong to this category. CDCL is similar to the earlier DavisPutnamLogemannLoveland (DPLL) solvers [58], but includes in addition conflict-driven learning. In Alg. 1 we show pseudo-code of a modern CDCL solver.

The six functions mentioned in the pseudocode are:

Boolean Constraint Propagation (BCP): propagates literals from unary clauses and might find a conflict in case a literal should get an opposite value of its current assignment.

Conflict Analyzer: In case BCP discovered a conflict, this function analyzes an implication graph and learns a new clause that prevents the solver from exploring the same space again.

Decision: A function that chooses the next literal as a decision.

Restart: Performs restart if condition "Restart-Condition" is TRUE.

Clauses Deletion: Under the given condition "Clause-Deletion-Condition" some of the learnt clauses are deleted.

Inprocessing: Collection of different strategies for formula simplification.

We now mention two contributions we made in this thesis to core SAT solving.

Algorithm 1 Modern CDCL SAT Solver

Input: Boolean formula in CNF form.
Output: SAT or UNSAT (or TIMEOUT).

```
1: Init();
2: while no timeout do
3:     confl = BCP();
4:     if confl != NULL then
5:         if no decisions made then
6:             return UNSAT
7:         else
8:             ConflictAnalyzer(confl)
9:     else
10:        if Inprocessing-Condition then
11:            Inprocessing();
12:        if Restart-Condition then
13:            Restart();
14:        if Clause-Deletion-Condition then
15:            ClausesDeletion();
16:        Decision();
17:        if no new decision were made then
18:            return SAT
19: return TIMEOUT;
```

1.1.1 Local Restarts

In most or even all SAT solvers the restart strategy is based on the number of conflicts during the solution process, but this number is a constant decided by the developer and does not relate in any way to the solver's state. We addressed this issue by correlating it to the solver's search tree. The motivation is to prevent useless restarts in case the solver only entered to a new search space and on the other hand to perform restart when it spent a significant effort under a specific search space. To measure the solver's effort we use the number of conflicts that occurred since entering a specific search space. If the number of conflicts is higher than a specific threshold the solver performs restart. This way the solver does not restart on newly entered branches and restarts on the old ones. Full details of our work can be found in Chapter 2.

1.1.2 Clause Shrinking

Clauses learnt by the SAT solver can frequently be made stronger and hence improve the search. This idea is called Shrinking [64], and was implemented by A. Nadel in the solver JERUSAT, which was a winner of the SAT'04 competition. After a conflict, JERUSAT applies shrinking if its *shrinking condition* is satisfied. The shrinking condition of JERUSAT is satisfied if the conflict clause contains no more than one variable from each decision level. The solver then sorts the conflict clause literals according to its *sorting scheme*. The sorting scheme of JERUSAT sorts the clause by decision level from lowest to highest. Afterwards it backtracks to the *shrinking backtrack level*, which in the case of JERUSAT is the highest possible decision level where all the literals of the conflict clause become unassigned. It then guides the decision heuristic to select the literals of the conflict clause according to the sorted order and assigns them the value false whenever possible. As usual, BCP follows each assignment. As result of a backtracking followed by specific decisions, one can see in a shrinking strategy a combination between partial restart combined with a decision strategy. In Chapter 3 we propose two new heuristics for improving Clause Shrinking. First, we propose generalizing the shrinking condition of JERUSAT. We count the number of decision levels associated with a conflict clauses variables and perform shrinking if

this number is greater than a threshold x. Second, we propose using a new sorting scheme, called *activity ordering*. Our scheme sorts the variables of the conflict clause according to VSIDSs scores, from highest to lowest. Our proposal is intended to make the solver even more dynamic, since it reorders the relevant variables according to their contribution to the derivation of recent conflict clauses.

1.2 Incremental SAT Solving

In numerous industrial applications the SAT solver is a component in a bigger system that sends it satisfiability queries. For example, a program that plans a path for a robot may use a SAT solver to find out if there exists a path within k steps from the current state. If the answer is negative, it increases k and tries again. The important point here is that the sequence of formulas that the SAT solver is asked to solve is not arbitrary: these formulas have a lot in common. Can we use this fact to make the SAT solver run faster? we should somehow reuse information that was gathered in previous instances to expedite the solution of the current one. To make things simpler, consider two CNF formulas, C_1 and C_2, which are solved consecutively, and assume that C_2 is known at the time of solving C_1. There are two kinds of information that can be reused when solving C_2:

- *Reuse clauses.* We should answer the following question: if c is a conflict clause learned while solving C_1, under what conditions is C_2 and $C_2 \wedge c$ equisatisfiable? It is easier to answer this question if we view C_1 and C_2 as sets of clauses. Let C denote the clauses in the intersection $C_1 \cap C_2$. Any clause learnt solely from C clauses can be reused when solving C_2. In practice, as in the path planning problem mentioned above, consecutive formulas in the sequence are very similar, and hence C_1 and C_2 share the vast majority of their clauses, which means that most of what was learnt can be reused.

- *Reuse heuristic parameters.* Various weights are updated during the solving process, and used to heuristically guide the search, e.g., variable score is used in decision making, weights expressing the activity of

clauses in deriving new clauses are used for determining which learned clauses should be maintained and which should be deleted, etc. If C_1 and C_2 are sufficiently similar, starting to solve C_2 with the weights at the end of the solving process of C_1 can expedite the solving of C_2.

To understand how modern SAT solvers support incremental solving, one should first understand a mechanism called *assumptions*, which was introduced with the SAT solver MiniSAT [32]. Assumptions are literals that are known to hold when solving C_1, but may be removed or negated when solving C_2. The list of assumption literals is passed to the solver as a parameter. The solver treats assumptions as special literals that dictate the initial set of decisions. If the solver backtracks beyond the decision level of the last assumption, it declares the formula to be unsatisfiable, since there is no solution without changing the assumptions. For example, suppose $a_1, \ldots, a_n$ are the assumption literals. Then the solver begins by making the decisions $a_1 = \text{TRUE}, \ldots, a_n = \text{TRUE}$, while applying BCP as usual. If at any point the solver backtracks to level n or less, it declares the formula to be unsatisfiable.

The key point here, is that all clauses that are learnt are *independent of the assumptions* and can therefore be reused when these assumptions no longer hold. This is the nature of learning: it learns clauses that are independent of specific decisions, and assumptions are just decisions. Hence, we can start solving C_2 while maintaining all the clauses that were learnt during the solving process of C_1. Note that this way we reuse both types of information mentioned above, and save the time of re-parsing the formula.

We now describe how assumptions are used for solving the general incremental SAT problem, which requires both addition and deletion of clauses between instances. As for adding clauses, the solver receives the set of clauses that should be added ($C_2 \setminus C_1$ in our case) as part of its interface. Removing clauses is done by adding a new assumption literal (corresponding to a *new* variable) to every clause $c \in (C_1 \setminus C_2)$, negated. For example, if $c = (x_1 \vee x_2)$, then it is replaced with $c' = (\neg a \vee x_1 \vee x_2)$, where a is a new variable. Note that under the assumption $a = \text{TRUE}$, $c = c'$, and hence the added assumption literal does not change the satisfiability of the formula. When solving C_2, however, we replace that assumption with the assumption $a = \text{FALSE}$, which is equivalent to erasing the clause c. Assumption literal used in this way are

called *clause selectors*.

One of the major breakthroughs in practical SAT solving in the last few years has been the combined preprocessing techniques that were suggested by [30]: non-increasing variable elimination through resolution, coupled with subsumption and self-subsumption. Those preprocessing techniques are widely adopted by most of the SAT solvers today. A known problem with variable elimination is the fact that it is incompatible at least in its basic form as published, with incremental SAT solving [32, 81, 92]. The reason, as was pointed out already in [30], is that variables that are eliminated may reappear in future instances. Soundness is not maintained in this scenario. Several attempts were made to deal with the reappearing variables problem, but those solutions require prior knowledge about the problem. For example, if it is known in advance which variables are going to participate on next incremental SAT solver calls, we could freeze them for elimination and prevent the problem. In many real life application this prior knowledge is impossible to have, so the solution was not to solve the problem incrementally but create a new formula instance each individual call. Using that solution the SAT solver loses all its conflict clauses and heuristics adaptation parameters, which makes the solver run much slower and in many cases it makes the use of preprocessing techniques un-beneficial. In addition, such preprocessing techniques cannot handle assumptions as if they were unit clauses because this would affect soundness. In case the number of assumptions is high, the lack of such preprocessing hinders performance. In this thesis we present two possible solutions:

1. *Low number of assumptions* — This solution can be used in any incremental case, but more beneficial if the number of assumptions is low. The idea is to track the eliminated variables, and maintain information that enables us to retrieve them when needed. All clauses that contain eliminated variables are kept for future use and not just one polarity as was done in MINISAT [32]. In addition the order of elimination is kept fixed. In case a new clause is added during an incremental call and this clause contains one of the previously eliminated variables we can decide to re-eliminate it using the saved clauses or re-introduce it back to the formula. Assumptions are frozen for preprocessing and if

the eliminated variable appears as assumption in an incremental call, it is re-introduced back to the formula. Full details on this technique appear in Chapter 4.

2. *High number of assumptions* — In case the number of assumptions is high and many of the assumptions repeat in several incremental calls it is usually useful to treat those repeated assumptions as unit clauses and activate formula simplification, in contrast to the current state-of-the-art approach that models assumptions as first decision variables. We show that a notable advantage of our approach is that it can make pre-processing algorithms much more efficient. However, our initial scheme renders assumption-dependent (or temporary) conflict clauses unusable in subsequent invocations. To resolve the resulting problem of reduced learning power, we introduce an algorithm that transforms such temporary clauses into assumption-independent pervasive clauses. In addition, we show that our approach can be enhanced further when a limited form of look-ahead information is available. Full details are in Chapter 5.

1.3 MUS and HLMUC

Subset S of a given SAT problem φ is an *unsatisfiable core* (UC) of φ if S is unsatisfiable. S is a *Minimal Unsatisfiable Core* (MUC) (*Minimal Unsatisfiable Subformula* (MUS)) if removal of any clause from S makes it satisfiable. More formally:

Definition 1.3.1 *if $S \subseteq \varphi$ and S is unsatisfiable, then S is UC.*

Definition 1.3.2 *if $\forall c \in S$, $S \setminus \{c\}$ is satisfiable, then S is MUS.*

Example 1.3.1 *(UC and MUS). An example of a CNF formula is:*

$$c_1 = (v_1 \vee v_2) \quad c_2 = (\neg v_1 \vee \neg v_2) \quad c_3 = (\neg v_1 \vee v_2)$$

$$c_4 = (v_1 \vee \neg v_2) \quad c_5 = (v_3 \vee v_4) \quad c_6 = (v_4 \vee \neg v_5)$$

$$\varphi = c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6$$

Possible unsatisfiable core (UC) is :

$$S = \{c_1, c_2, c_3, c_4, c_5\}$$

Minimal unsatisfiable subset (MUS) is:

$$S' = \{c_1, c_2, c_3, c_4\}$$

In this case there is only one MUS, but in general there can be many minimal cores.

A variety of tasks in formal verification require finding small or minimal unsatisfiable cores. For example, MUSes are used in a number of verification tasks to extract a concise description of inconsistency. As a result, MUS extraction algorithms are currently a very active area of research and some recent work include [10, 29, 66, 82, 83, 89]. MUS solvers use SAT solvers as their engines. The most recent overview of MUS extraction algorithms can be found in [9]. As mentioned in [9] three main approaches have been proposed for the MUS computation: constructive, destructive and dichotomic. Our solver is based on the destructive algorithm as seen in Figure 1.1.

Most of the latest MUS solvers are based on addition of assumptions literals to clauses. By manipulating those assumptions, clauses can be added and removed. In addition, using those assumptions makes it easy to find which clauses are required for deriving the empty clause. This approach has an advantage of using any available SAT solver without any modifications, but prevents using the fact that the SAT Solver is a part of the MUS extractor and therefore additional optimizations can be performed. Use of assumptions has the disadvantage that it increases the size of conflict clauses. There are several works that are trying to solve this issue, like [3, 51]. To avoid potential problems with assumptions, *resolution-based* SAT solver can be used, as was published in [66]. Resolution-based SAT solver means that for every new learnt clause we keep its resolution DAG. When the empty clause is reached, all the input clauses in the resolution tree are marked as an unsatisfiable core. Keeping resolution instead of using assumptions literals

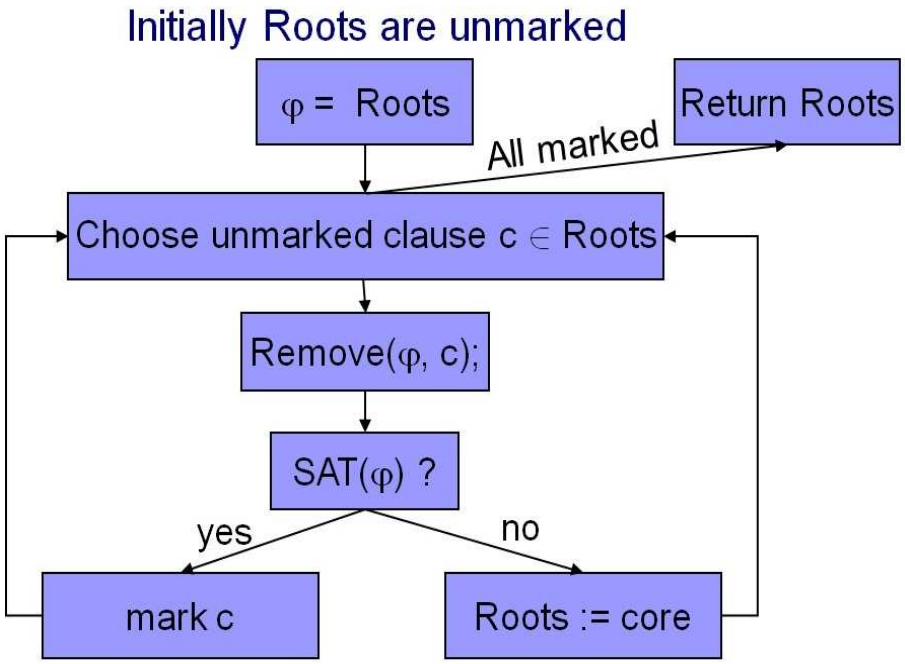

Figure 1.1: Description of deletion-based minimization MUS algorithm which works for both assumptions-based and resolution-based

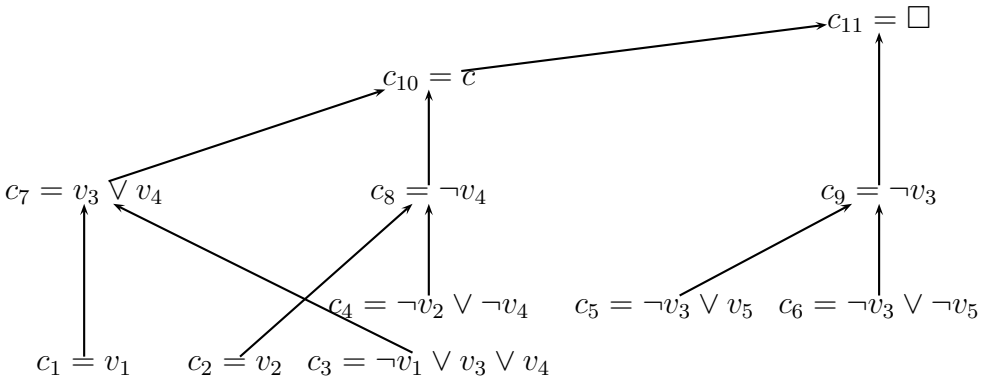

Figure 1.2: An example of a resolution tree of an empty clause

allows creating smaller conflict clauses and keep relations between clauses, in addition to other optimizations that are detailed in Chapter 7. An example of a resolution tree can be seen in Fig. 1.2.

Our first version of a MUS solver HAIFA-MUC [76] won the two first places in the SAT Competition 2011 on the MUS track [25] while the main advantage over other solvers was the use of resolution graph instead of assumptions (our assumptions-based solver got only to the 6th place). Later an additional technique of *model rotation* was presented in [10, 53]. This technique has a major positive impact on run-times. In Chapter 7 we improved model rotation to become more eager, which improved performance even further. We show in that chapter various other modifications to the SAT core engine based on the resolution graph, which makes our solver HAIFA-MUC faster than any other solver in existence.

1.3.1 High Level Minimal Unsatisfiable Core extraction

In most cases it is not the core itself that is being used, rather it is processed further in order to check which clauses from a preknown set of *Interesting Constraints* (where each constraint is modeled with a conjunction of clauses) participate in the proof. The problem of minimizing the participation of interesting constraints was recently coined *high-level* minimal unsatisfiable core (HLMUC) in [66], also known as Group Minimal Unsatifiable Sub-

set/Subformula/Set (GMUS). The HLMUC input is a Boolean formula Ψ and a set of interesting constraints IC. Each IC is a set of clauses. The problem of HLMUC is to find a minimal number of interesting constraints that are unsatisfiable in conjunction with the rest of the input formula. More formally:

Definition 1.3.3 *For formula $\Psi = \bigwedge_{R_i \in IC} R_i \wedge \Omega$, if C is UC of Ψ then, $HUC = \{R_j | \exists c : c \in R_j \wedge c \in C\}$ is a high-level unsatisfiable core. If in addition $\forall R_j, (C \setminus R_j) \wedge \Omega$ is satisfiable then HUC is a High-Level Minimal Unsatisfiable Core (HLMUC).*

Example 1.3.2 *(HLMUC). An example of a CNF formula is:*

$$c_1 = (v_1 \vee v_2) \quad c_2 = (\neg v_1 \vee \neg v_2) \quad c_3 = (\neg v_1 \vee v_2)$$

$$c_4 = (v_1 \vee \neg v_2) \quad c_5 = (v_3 \vee v_4) \quad c_6 = (v_4 \vee \neg v_5)$$

$$R_1 = \{c_1, c_4\} \quad R_2 = \{c_5, c_6\}$$

$$\Omega = c_2 \wedge c_3$$

$$\Psi = c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6$$

As in Example 1.3.1 our UC is:

$$S = \{c_1, c_2, c_3, c_4, c_5\}$$

Then high-level UC is:

$$HLS = \{R_1, R_2\}$$

High-Level Minimal Unsatisfiable Core (HLMUC) is:

$$HLS' = \{R_1\}$$

Because c_5 and c_6 are not required.

For HLMUC with $\Omega = \emptyset$, and when all R_i are a single clause, then it is a MUS problem; therefore MUS is just a special case of an HLMUC problem. Two prominent examples of verification techniques that need such small

cores are 1) abstraction-refinement model-checking techniques, which use the core in order to identify the state variables that will be used for refinement (smaller number of such variables in the core implies that more state variables can be replaced with free inputs in the abstract model), and 2) assumption minimization, where the goal is to minimize the usage of environment assumptions in the proof, because these assumptions have to be proved separately. We propose seven improvements to the recent solution given in [66], which together result in an overall reduction of 55% in run time and 73% in the size of the resulting core, based on our experiments with hundreds of industrial test cases. The optimized procedure is also better empirically than the assumptions-based minimization technique, and faster by more than an order of magnitude than the best known general MUS solver. Similar to MUS, HLMUC can be easily implemented using assumptions literals, but performance wise using resolution is more beneficial. Our resolution based solver Haifa-HLMUC [76] won the first place in the 2011 SAT Competition in the High Level MUS track [24] (no competition was held since), while the assumption-based solver Haifa-HLMUC-A took second and third places with very significant performance difference. Our solver gives higher priority to clauses from Ω, so each unsatisfiable core returned by each invocation contains a reduced number of important constraints clauses. Haifa-MUC is currently the fastest published solver for High-Level MUS extraction. In Chapter 6 the reader can find our suggested improvements.

Chapter 2

Local Restarts

Vadim Ryvchin[1,2] and Ofer Strichman[1]

[1] *Information Systems Engineering, IE, Technion, Haifa, Israel*

[2] *Design Technology Solutions Group, Intel Corporation, Haifa, Israel*

Abstract

Most or even all competitive DPLL-based SAT solvers have a *restart* policy, by which the solver is forced to backtrack to decision level 0 according to some criterion. Although not a sophisticated technique, there is mounting evidence that this technique has crucial impact on performance. The common explanation is that restarts help the solver avoid spending too much time in branches in which there is neither an easy-to-find satisfying assignment nor opportunities for fast learning of strong clauses. All existing techniques rely on a global criterion such as the number of conflicts learned as of the previous restart, and differ in the method of calculating the threshold after which the solver is forced to restart. This approach disregards, in some sense, the original motivation of focusing on 'bad' branches. It is possible that a restart is activated right after going into a good branch, or that it spends all of its time in a single bad branch. We suggest instead to localize restarts, i.e., apply restarts according to measures local to each branch. This adds a dimension to the restart policy, namely the decision level in which the solver is currently in. Our experiments with both Minisat and Eureka show that with certain parameters this improves the run time by 15% - 30% on average (when applied to the 100 test benchmarks of SAT-race'06), and reduces the number of time-outs.

2.1 Global vs. Local Restarts

Most or even all competitive DPLL SAT solvers have a "restart" policy, a strategy initially proposed by Gomes et. al [38]. The solver is restarted after a certain number of conflict clauses have been learned. The fact that new clauses have been added to the clause database deviates the search from one restart to the next. In those solvers that is relevant, the search is changed also owing to randomness.

Different restart policies are used by different solvers. A recent survey by Huang [43] includes several types of restart policies. We briefly describe various types of popular restart techniques based on that survey and on some new developments.

1. *Arithmetic (or fixed) series.* Parameters: x, y. A policy in which there is a restart every x conflicts, which is increased by y every restart. Some sample values are: in zchaff 2004 $x = 700$, in Berkmin $x = 550$, in Siege $x = 16000$ and in Eureka $x = 2000$. In all of these solvers the series is in fact fixed (i.e., $y = 0$), owing to the observation that completeness is meaningless in the realm of timeouts.

2. *Geometric series.* Parameters: x, y. A policy in which the initial interval is x, which is then multiplied by a factor of y in each restart, for some $y > 1$. This policy is used in Minisat-2 with $x = 100$ conflicts and $y = 1.5$.

3. *Inner-Outer Geometric series.* Parameters: x, y, z. An idea suggested by Biere and implemented in PicoSAT [16], by which restarts follow what can be seen as a two dimensional pattern that increases geometrically in both dimensions. The inner loop multiplies a number initialized to x, by z, at each restart. When this number is larger than a threshold y, it is reset back to x and the threshold y is also multiplied by z (this is the outer loop). Hence, both the inner and outer loops follow a geometric series, and the whole series creates an oscillating pattern.

4. *Luby et al. series* [54]. Parameter: x. A policy in which restarts are performed according to the following series of numbers:

1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,... multiplied by the constant x (called the *unit-run*). Formally, let t_i denote the i-th number in this series. Then t_i is defined recursively:

$$t_i = \begin{cases} 2^{k-1} & \text{if } \exists k \in \mathbb{N}.\ i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{if } \exists k \in \mathbb{N}.\ 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

This is a well-defined series, as the two conditions are mutually-exclusive. This policy has some nice theoretical characteristics in a class of randomized algorithms called Las Vegas algorithms[1], but the relevance of these results to DPLL has only been empirical so far – it is not clear what is the reason that it works well in practice. The experiments reported in [43] show that it outperforms the other restart strategies, and indeed this is now the restart method of choice of several state-of-the-art solvers, such as TinySAT [43] and RSAT [72].

For completeness of this list, we should also mention that there is a family of techniques in which 'restart' does not entail backtracking to level 0, but rather to some decision level which is lower than what is computed as the backtracking level by a conflict analysis procedure. Such a procedure was proposed, for example, by Lynch [55]. We did not experiment with these techniques, however.

All of the strategies listed above are based on a global counter of conflict clauses, and therefore they measure progress over many branches together. Assuming that the motivation for restarts is to prevent the solver from getting stuck in a bad branch (which can, informally, be defined as a branch which neither contains an easy-to-find satisfying assignment nor leads to efficient learning that directs the solver to a different search-space or to a proof of unsatisfiability), such a global policy may miss the point.

For example, it is possible that the solver spent a significant amount of time searching in a branch, eventually left it, and very soon after that it restarts (since the global threshold was reached), although there is no knowledge yet about the potential of the current branch. It is also possible

[1] Algorithms that use randomness, but the quality of the result is not affected by it. Typically randomness in such algorithms only affects run-times.

that the restart is too late, for example if it spends all its time between restarts in a single bad branch.

A possibly better strategy is to localize the measure of difficulty of branches, and restart when the branch is more difficult than some threshold. Each of the global strategies mentioned above can be applied locally, because we can count the number of conflicts under each branch easily, as follows. For each decision level d we maintain a counter $c(d)$, which is initially (when a decision is made at that level) set to the global number of conflicts. When backtracking back to that level, we examine the difference between the current global number of conflicts, and $c(d)$. This difference reflects the number of conflicts that were encountered above level d, since the last time a decision was made at this level. If this difference is larger than some strategy-dependent threshold, we restart.

Locality opens a new dimension, namely that of the decision level. In other words, the threshold can be a function of the level in which the solver is currently in. We call such strategies *dynamic*. It can be expected that the work done between two visits to a decision level (from decision to backtracking back to that level) will be smaller as the level increases. Also, we collected statistics regarding the size of learned clauses at each level, and it shows that conflict clauses at low decision levels are smaller on average. Hence giving less chance to deeper levels forces the solver to learn stronger facts first. Each of the strategies above can be made dynamic, although in strategies in which the series oscillates as in Luby et al. and the Inner-Outer strategy, it is not clear how to add this new dimension. We focused, then, on the following strategy:

5. Dynamic-fix. Parameters: x, y, d, min. A policy in which at decision level i there is a restart every $\max(x - i \cdot d, min)$ conflicts, which is increased by y every restart.

Making the strategy local instead of global requires re-tuning of the parameters – there is no reason to believe that parameters that optimize a global restart policy also optimize a local one. Hence a major empirical evaluation is needed in order to check the effect of locality on each of these strategies.

2.2 Experimental Results and Conclusions

The table in Figure 2.1 shows results with 40 different restart configurations, when implemented on top of MINISAT 2007 [34], and ran on the 100 industrial benchmarks that were used as preparation for SAT-race'06 (divided evenly to the two test-sets TS1 and TS2). A similar table for the latest version of Eureka [67], with 41 configurations, appears in Figure 2.2. The set of configurations is not identical, but close, because we chose them dynamically: when a good strategy was found, we tried to change it incrementally. The tables are sorted according to the type of strategy, local/global, and parameters. The third column indicates whether this strategy is implemented globally or locally. Timeout was set to 30 minutes. Instances that timed-out are included and contribute 30 minutes (we added them to the SAT or UNSAT column according to our prior knowledge of the expected result). Instances that none of our configurations nor any SAT'06-race competitor can solve are not included. The overall number of timeouts and total run time are given in the last two columns, where time is measured in hours. All together the two tables represent over 40 days of CPU time.

The first column indicates the position of each solver when measured by the total run time, and the best three configurations according to this measure are preceded by '✓'. With both solvers, the best three configurations that we tried are local (also when measured by time-outs).

To the extent that the benchmark set is representative of industrial problems, and that MiniSat 2007 and Eureka represent state-of-the-art solvers, it seems that locality can help with the four types of strategies that we tried. The following table shows, for the Luby and Inner-Outer strategies, the figures corresponding to the best local and best global configurations that we could find.

	Minisat				Eureka			
Strategy	Global		Local		Global		Local	
	TO	Time	TO	Time	TO	Time	TO	Time
Luby	11	8.98	9	7.89	9	8.90	8	8.40
IO	10	8.86	8	7.38	9	8.64	8	8.12

There seems to be such an advantage for the local geometric and local arithmetic strategies as well, but more global configurations of these strate-

gies need to be tested in order to draw concrete conclusions. If we take the default parameters of Minisat and Eureka as best of their respective global strategies, then this can be said with some confidence.

What about the dynamic strategy? It does not seem to score well in general, at least not with the 4 parameters set that we tried, but it performs well with unsatisfiable instances. In the case of the first table (Minisat), the dynamic strategies with parameters 1000,0.1,20,10 and 1000,0.1,10,10 arrive at the second and third places, respectively, if we measure only unsatisfiable instances. More parameters and variations of this strategy are necessary in order to see if it can become competitive in the general case.

We are currently trying more configurations and looking for other measures for the quality of the branch that can be checked with a marginal cost in run-time. It is possible that measures such as the size of the backtrack can be factored in the restart policy.

		G/		TS1				TS2				Overall	
Place	Strategy	L	Parameters	SAT	UNSAT	TO	Total	SAT	UNSAT	TO	Total	TO	Time
✓3	Arith	L	100,10	1.12	2.06	4	3.18	2.17	2.59	6	4.75	10	7.93
26	Arith	L	10,1	2.12	2.62	6	4.74	2.42	2.99	6	5.41	12	10.15
8	Arith	L	100,1	1.89	1.96	4	3.85	2.37	2.84	6	5.21	10	9.05
6	Arith	L	100,20	2.49	1.99	6	4.48	2.32	2.21	5	4.53	11	9.02
12	Arith	L	100,40	2.51	1.95	6	4.47	2.11	2.74	6	4.86	12	9.33
10	Arith	L	1000,0.1	2.3	2.05	4	4.35	1.89	2.85	6	4.74	10	9.09
9	Arith	L	1000,1	2.15	1.93	5	4.08	2.07	2.9	6	4.97	11	9.05
32	Arith	L	1000,10	2.76	2.13	7	4.89	2.72	2.99	8	5.71	15	10.6
34	Arith	L	1000,20	3.13	2.07	8	5.2	2.61	2.93	5	5.54	13	10.74
21	Arith	L	2500,1	2.11	2.38	6	4.49	2.37	3.03	7	5.39	13	9.89
24	Arith	L	3,1	2.47	1.87	3	4.34	2.88	2.81	9	5.69	12	10.03
29	Arith	L	3,10	2.69	1.92	6	4.61	2.95	2.92	9	5.87	15	10.48
14	Arith	L	5,0.2	2.41	1.62	6	4.04	2.59	2.85	8	5.43	14	9.47
15	Arith	L	5000,1	2.33	2.48	7	4.81	2.13	2.56	4	4.69	11	9.5
18	Arith	L	6,1	2.02	2.23	5	4.25	2.61	2.86	8	5.46	13	9.71
27	Geom.	L	10,1.1	2.53	2.03	6	4.56	2.5	3.18	8	5.68	14	10.24
37	Geom.	L	10,1.5	2.46	2.63	7	5.08	2.62	3.29	6	5.91	13	10.99
40	Geom.	L	10,2	2.89	2.77	9	5.65	3.03	3.39	9	6.42	18	12.07
16	Geom.	L	100,1.1	1.71	2.16	3	3.86	2.55	3.14	8	5.69	11	9.56
38	Geom.	L	100,1.5	3.33	2.71	9	6.03	2.94	2.77	6	5.71	15	11.75
36	Geom.	L	100,2	2.33	2.86	7	5.19	2.42	3.35	7	5.76	14	10.95
33	Geom. *	G	100,1.5	1.6	2.76	6	4.36	3.06	3.22	8	6.28	14	10.64
11	IO	G	100,1000,1.1	2.68	2.07	6	4.75	1.72	2.86	7	4.57	13	9.32
4	IO	G	100,1000,1.5	1.81	2.04	4	3.86	2.04	2.97	6	5	10	8.86
39	IO	G	100,1000,2	2.81	2.16	8	4.97	3.33	3.48	10	6.81	18	11.78
✓1	IO	L	100,1000,1.1	1.59	2	4	3.59	1.27	2.51	4	3.78	8	7.38
7	IO	L	100,1000,1.5	2.22	2.02	5	4.24	1.92	2.88	6	4.8	11	9.04
30	IO	L	100,1000,2	2.89	2.22	8	5.11	2.6	2.79	7	5.39	15	10.5
22	Luby	G	32	2.22	1.49	3	3.71	3.06	3.15	10	6.21	13	9.91
23	Luby	G	128	3.08	1.76	6	4.84	2.21	2.89	7	5.1	13	9.94
13	Luby	G	512	2.84	1.93	7	4.77	1.92	2.64	5	4.56	12	9.33
5	Luby	G	1024	2.26	1.97	5	4.22	2.02	2.74	6	4.76	11	8.98
✓2	Luby	L	32	1.6	1.15	3	2.75	2.22	2.92	6	5.14	9	7.89
25	Luby	L	128	2.75	2.01	7	4.76	2.29	3.02	7	5.32	14	10.08
17	Luby	L	512	2.18	2.08	5	4.26	2.33	3.1	6	5.43	11	9.69
19	Luby	L	1024	2.71	2.02	4	4.73	1.94	3.05	7	5	11	9.73
28	D-arith	L	1000,0.1,10,10	3.45	1.02	6	4.47	2.7	3.13	8	5.84	14	10.31
20	D-arith	L	1000,0.1,20,10	2.92	0.99	4	3.91	2.77	3.1	8	5.87	12	9.78
31	D-arith	L	1000,10,10,10	3.5	2	8	5.51	1.64	3.41	7	5.05	15	10.56
35	D-arith	L	1000,10,20,10	3.22	2.02	8	5.24	2.25	3.4	8	5.65	16	10.89

Figure 2.1: Results, in hours, based on MINISAT 2007. The original configuration of MINISAT 2007 is marked with *.

		G/		TS1				TS2				Overall	
Place	Strategy	L	Parameters	SAT	UNSAT	TO	Total	SAT	UNSAT	TO	Total	TO	Time
39	Arith	L	10,0.1	2.34	1.26	4	3.6	2.78	4.22	11	7	15	10.59
38	Arith	L	10,1	1.92	1.67	4	3.59	2.93	4.06	10	6.98	14	10.58
41	Arith	L	100,1	2.19	1.63	3	3.81	3.24	4.04	10	7.28	13	11.09
17	Arith	L	100,10	1.78	1.11	2	2.89	2.8	3.44	7	6.24	9	9.13
✓2	Arith	L	1000,1	1.6	1.04	2	2.64	2.74	2.72	6	5.46	8	8.09
5	Arith	L	1000,10	1.63	0.96	2	2.59	3.05	2.68	5	5.72	7	8.31
✓1	Arith	L	1000,20	1.83	0.92	2	2.75	2.57	2.67	5	5.24	7	7.98
40	Arith	L	20,0.1	2.47	1.35	4	3.82	2.65	4.23	11	6.87	15	10.69
31	Arith	L	20,1	2.4	1.32	3	3.72	2.63	3.69	9	6.32	12	10.04
14	Arith	L	2000,1	1.76	1.1	2	2.86	3.4	2.81	6	6.21	8	9.08
32	Arith	L	3,1	2.04	1.19	3	3.23	3.4	3.43	9	6.83	12	10.06
8	Arith	L	3,10	1.63	1	2	2.63	2.66	3.24	6	5.89	8	8.52
4	Arith	L	3,20	1.7	0.9	2	2.6	2.47	3.21	7	5.68	9	8.28
21	Arith	L	3,40	1.79	0.92	2	2.71	3.54	3.39	8	6.93	10	9.64
37	Arith	L	5,0.2	2.29	1.23	3	3.53	3.17	3.85	10	7.02	13	10.55
18	Arith	L	5000,1	1.71	1.08	2	2.79	3.01	3.44	7	6.45	9	9.24
19	Arith*	G	2000,0	2.15	1.07	3	3.22	3.17	3	6	6.17	9	9.39
29	Geom.	L	10,1.1	2.2	1.07	3	3.26	3.27	3.49	9	6.76	12	10.03
36	Geom.	L	10,1.5	1.89	1.1	2	2.99	3.17	4.23	10	7.4	12	10.39
25	Geom.	L	10,2	1.96	1.32	2	3.28	3.14	3.38	9	6.52	11	9.80
11	Geom.	L	100,1.1	1.98	0.9	2	2.88	2.8	3.1	7	5.9	9	8.78
28	Geom.	L	100,1.5	1.73	0.95	2	2.68	3.46	3.78	9	7.24	11	9.93
30	Geom.	L	100,2	2.11	1.01	2	3.12	3.16	3.75	7	6.91	9	10.04
10	IO	G	100,1000,1.1	1.54	0.93	2	2.47	3.05	3.12	7	6.17	9	8.64
15	IO	G	100,1000,1.5	1.59	0.9	1	2.49	3.01	3.57	8	6.58	9	9.08
26	IO	G	100,1000,2	2.12	0.87	3	2.99	3.34	3.48	8	6.83	11	9.82
✓3	IO	L	100,1000,1.1	1.72	0.88	2	2.6	2.82	2.7	6	5.52	8	8.12
22	IO	L	100,1000,1.5	2.19	0.86	3	3.05	3.14	3.55	8	6.68	11	9.73
34	IO	L	100,1000,2	2.34	1.1	3	3.44	3.13	3.76	8	6.88	11	10.32
16	Luby	G	32	1.83	1.03	3	2.86	2.97	3.29	7	6.26	10	9.12
12	Luby	G	128	2.17	0.87	2	3.05	2.92	2.94	7	5.86	9	8.90
13	Luby	G	512	1.59	1	2	2.59	3.18	3.27	7	6.46	9	9.05
23	Luby	G	1024	2.22	1.09	3	3.31	3.58	2.88	6	6.46	9	9.76
9	Luby	L	32	1.67	0.94	1	2.61	2.75	3.17	7	5.92	8	8.53
7	Luby	L	128	1.71	0.91	1	2.62	2.84	2.96	6	5.79	7	8.41
6	Luby	L	512	1.6	0.94	2	2.54	3.14	2.72	6	5.86	8	8.40
27	Luby	L	1024	2.33	1.1	3	3.43	3.6	2.87	7	6.47	10	9.90
24	D-arith	L	1000,0.1,10,10	1.91	1.34	3	3.25	3.26	3.27	8	6.53	11	9.77
35	D-arith	L	1000,0.1,20,10	1.86	1.71	4	3.57	3.15	3.66	9	6.81	13	10.38
20	D-arith	L	1000,10,10,10	1.88	1.2	2	3.08	3.25	3.28	5	6.53	7	9.61
33	D-arith	L	1000,10,20,10	1.82	1.31	2	3.13	3.25	3.74	8	6.98	10	10.11

Figure 2.2: Results, in hours, based on EUREKA. The original configuration of EUREKA is marked with *.

Chapter 3

Assignment Stack Shrinking

Alexander Nadel[1] and Vadim Ryvchin[1,2]
[1] *Intel Corporation, P.O. Box 1659, Haifa 31015 Israel*
[2] *Information Systems Engineering, IE, Technion, Haifa, Israel*

Abstract

Assignment stack shrinking is a technique that is intended to speed up the performance of modern complete SAT solvers. Shrinking was shown to be efficient in SAT'04 competition winners Jerusat and Chaff. However, existing studies lack the details of the shrinking algorithm. In addition, shrinking's performance was not tested in conjunction with the most modern techniques. This paper provides a detailed description of the shrinking algorithm and proposes two new heursitics for it. We show that using shrinking is critical for solving well-known industrial benchmark families with the latest versions of Minisat and Eureka.

3.1 Introduction

Modern SAT solvers are known to be extremely efficient on many industrial problems which may comprise up to millions of variables and clauses. Among the key features that enable the solvers to be so efficient, despite the apparent difficulty of solving huge instances of NP-complete problems, are *dynamic behavior* and *search locality*, that is, the ability to maintain the set of assigned variables and recorded clauses relevant to the currently explored space. This effect is achieved by applying various techniques, such as the VSIDS decision heuristic [62] (which gives preference to variables that participated in recent conflict clause derivations) and local restarts (such as [77]). Another important feature of modern SAT solvers is that they tend to pick *interrelated variables*, that is, variables whose joint assignment increases the chances of quickly reaching conflicts in unsatisfiable branches and satisfying clauses in satisfiable branches. Clause-based heuristics (such as CBH [27]), which prefer to pick variables from the same clause, increase the interrelation of the assigned variables.

Assignment stack shrinking (or, simply, shrinking) is a technique that seeks to boost the performance of modern SAT solvers by making their behavior more local and dynamic, as well as by improving the interrelation of the assigned variables.

Shrinking was introduced in [64] and implemented in the Jerusat SAT solver. After a conflict, Jerusat applies shrinking if its *shrinking condition* is satisfied. The shrinking condition of Jerusat is satisfied if the conflict clause contains no more than one variable from each decision level. The solver then sorts the conflict clause literals according to its *sorting scheme*. The sorting scheme of Jerusat sorts the clause by decision level from lowest to highest. Afterwards Jerusat backtracks to the *shrinking backtrack level*. The shrinking backtrack level for Jerusat is the highest possible decision level where all the literals of the conflict clause become unassigned. Jerusat then guides the decision heuristic to select the literals of the conflict clause according to the sorted order and assign them the value false, whenever possible. As usual, Boolean Constraint Propagation (BCP) follows each assignment.

One can pick out three important components of the shrinking algorithm

that can be tuned heuristically: the shrinking condition, the sorting scheme, and the determination of the shrinking backtrack level. Shrinking was implemented in the 2004 version of the Chaff SAT solver [57] with important modifications in each one of these components, as described below.

3.2 Algorithmic Details and New Heuristics

Chaff had two versions: *zchaff.2004.5.13* and *zchaff_rand*. We concentrate on *zchaff_rand*'s version of shrinking, since it was shown to be more useful in [57], and also performed better in the SAT'04 competition [13]. Suppose Chaff encounters a conflict. Chaff considers applying shrinking if the length of the conflict clause exceeds a certain threshold x. The clause is sorted according to decision levels. The algorithm finds the lowest decision level that is less than the next higher decision level by at least 2. (If no such decision level is found, shrinking is not performed.) The algorithm backtracks to this decision level, and the decision strategy starts reassigning the value false to the unassigned literals of the conflict clause, whenever possible. Chaff reassigns the variables in the reverse order, that is, in descending order of decision levels, since this sorting scheme was found to perform slightly better than Jerusat's in [57]. The threshold value x for applying shrinking is set dynamically using some measured statistics. More specifically, Alg. 2 is used in Chaff for adjusting x after every y conflicts. Chaff measures the mean and standard deviation of the lengths of the recently learned conflict clauses and tries to adjust x to keep it at a value greater than the mean. The threshold on the number of conflicts y is 600 for Chaff.

Chaff's shrinking algorithm was implemented in Intel's SAT solver Eureka with two minor differences: (1) The threshold on the number of conflicts y is 2000; (2) Eureka forbids performing shrinking for two conflicts in a row.

An important detail for understanding the reasons for the efficiency of shrinking is that a conflict clause is recorded even when shrinking is applied. Hence the solver always explores a different subspace after performing shrinking. Previous works [57, 64, 65] claimed that a "similar" conflict must follow an application of shrinking, on the assumption that a conflict clause is not recorded when shrinking is applied, but this claim does not fit the actual way

shrinking is implemented in Jerusat, Chaff, and Eureka.

Applying shrinking contributes to search locality and makes the solver more dynamic, since the set of assigned variables becomes more relevant to the recently explored search space as irrelevant variables become unassigned. Also, since the variables on the assignment stack are precisely those that appeared in recent conflict clauses, conflict clauses are more likely to share common interrelated variables. Shrinking often reduced the average length of learned conflict clauses and led to faster solving times, especially for the microprocessor verification benchmarks in Chaff [57].

We propose two new heuristics for shrinking. First, we propose generalizing the shrinking condition of Jerusat. We count the number of decision levels associated with a conflict clause's variables and perform shrinking if this number is greater than a threshold x. The threshold is calculated exactly like the conflict clause size threshold in Chaff in Alg. 2, using the number of decision levels in the clauses instead of their lengths. We dub our proposal the *decision-level-based shrinking condition*. Interestingly, Jerusat's shrinking condition and its proposed generalization correspond to the recent observation that a "good" clause should contain as few decision levels as possible [4]. The clause deletion scheme of SAT'09 competition winner Glucose is based on this observation. Second, we propose using a new sorting scheme, called *activity ordering*. Our scheme sorts the variables of the conflict clause according to VSIDS's scores, from highest to lowest. Our proposal is intended to make the solver even more dynamic, since it reorders the relevant variables according to their contribution to the derivation of recent conflict clauses.

3.3 Experimental Results and Discussion

We used Eureka and Minisat for our experiments. Minisat was enhanced by a restart strategy that was found to be optimal for this solver in [77]. We used eight publicly available benchmark families: sat04-ind-goldberg03-hard_eq_check [13] (henceforth, abbreviated to ug), sat04-ind-maris03-gripper [13] (mm), sat04-ind-velev-vliw_unsat_2.0 [91] (uv2), SAT-Race_TS_1 [85] (ms1), SAT-Race_TS_2 [85] (ms2), velev_fvp-sat.3.0 [90] (sv3), velev_fvp-unsat.3.0 [90] (uv3), velev_vliw_unsat_4.0 [91] (uv4).

Algorithm 2 Adjust Threshold for Shrinking (Threshold for shrinking x, Threshold for number of learned clauses y)

Require: x is initialized with the value 95 at the beginning of SAT solving.

1: $(mean, stdev) :=$ mean and standard deviation of last y learned clause lengths
2: $center := mean + 0.5 * stdev$; $ulimit := mean + stdev$
3: **if** $x \geq center$ **then**
4: $\quad x := x - 5$
5: **if** $x < center$ **then**
6: $\quad x := x + 5$
7: **if** $x > ulimit$ **then**
8: $\quad x := ulimit$
9: **if** $x < 5$ **then**
10: $\quad x := 5$
11: **return** x

			No Shr.		Base Shr.		Act. Order		Dec. Cond.	
Family	**SAT?**	**Inst.**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**	**Solved**	**Time**
ug	UNS	13	10	67005	13	12041	13	14389	12	28457
mm	MIX	10	5	66602	7	39870	7	39426	8	44404
uv2	UNS	8	1	78870	8	12129	8	10283	8	10914
ms1	MIX	50	47	51117	49	27352	48	38208	50	16279
ms2	MIX	50	42	109899	44	92813	43	96564	42	99882
sv3	SAT	20	20	767	20	1119	20	788	20	1375
uv3	UNS	6	1	62038	6	10863	6	11761	6	11251
uv4	UNS	4	0	43200	4	10874	4	9018	4	10677
Sum		161	126	479498	151	207061	149	220437	150	223239

Table 3.1: Shrinking within Eureka

For each solver, we compared the following four versions, applying: (1) no shrinking; (2) the base version of shrinking, corresponding to Eureka's version of shrinking (recall from Section 3.2 that Eureka's shrinking algorithm is largely similar to Chaff's: its shrinking condition is based on clause length and the sorting scheme picks variables in descending order of decision levels); (3) the base version, modified by applying activity ordering; (4) the base version, modified by using the decision-level-based shrinking condition.

Table 3.1 provides some statistics regarding the benchmark families as well as Eureka's results. The first column of the table contains the family name, the second column specifies whether the instances are satisfiable, unsatisfiable, or mixed, and the third column contains the number of in-

Family	SAT?	Inst.	No Shr.		Base Shr.		Act. Order		Dec. Cond.	
			Solved	Time	Solved	Time	Solved	Time	Solved	Time
ug	UNS	13	7	82310	10	43007	10	43686	11	44140
mm	MIX	10	0	108000	4	71234	0	108000	4	76680
uv2	UNS	8	1	85508	8	12235	8	10817	8	11743
ms1	MIX	50	48	36771	47	37771	49	26894	49	20557
ms2	MIX	50	44	82982	41	122233	42	107147	41	107780
sv3	SAT	20	16	53968	20	9330	20	10084	20	6954
uv3	UNS	6	0	64800	3	38056	0	64800	3	39652
uv4	UNS	4	1	33370	4	15230	4	9912	4	14798
Sum		161	117	547709	137	349096	133	381340	140	322304

Table 3.2: Shrinking within Minisat

stances in the family. Each subsequent pair of columns shows the number of instances solved by Eureka within a three hour timeout and the overall run-time for the particular version in seconds (10800 seconds, that is, three hours, is added for an unresolved benchmark). Table 3.2 provides Minisat's results in the same format. (A table with all the details of the experimental results appears in [68].)

Compare the empirically best shrinking algorithm versus the version without shrinking for each solver. For Eureka, shrinking (the base version) is helpful for solving seven out of eight families, and critical for solving ug, uv2, uv3 and uv4. For Minisat, shrinking (with the decision-level-based shrinking condition) is critical for solving seven out of eight families (ms2 is an exception). Overall, shrinking enables Eureka and Minisat to solve, respectively, 25 and 23 more benchmarks within the timeout. Hence employing shrinking is highly advantageous.

Compare now our two variations of shrinking versus the base version. The effect of applying the decision-level-based shrinking condition in Minisat is clearly positive as it leads to better overall performance in terms of both the number of solved instances and the run-time. Although applying the decision-level-based ordering condition within Eureka does not lead to better results overall, the solver does perform better for four families (the gap is especially significant for ms1) than with the base version. While the impact of activity ordering is negative for Minisat overall, it performs better than best version (the version with the decision-level-based shrinking condition) for three families. Activity ordering is not helpful overall for Eureka, but is does help solve four families more quickly than the best version (the version

with base shrinking). Hence it is recommended that shrinking be tuned for each specific solver and benchmark family.

An important question is whether the effect of shrinking can be achieved by applying other algorithms, proposed after shrinking. Consider the following three techniques: (1) Frequent restarts [16, 77]; (2) A clause-based heuristic, such as CBH [27]; and (3) RSAT's polarity selection heuristic [71], which assigns every decision variable the last value it was assigned. Observe that the combined effect of these three techniques seems to be similar to that of shrinking. First, restarting the search when a certain condition holds corresponds to backtracking when the shrinking condition is met. Second, applying a clause-based heuristic and RSAT's polarity selection heuristic results in selecting the last conflict clause and assigning its literals the value false, similar to what happens in shrinking. It was claimed in [16] that the impact of conflict clause minimization [7, 87] could be considered somewhat similar to the impact of shrinking, since minimization reduces the size of conflict clauses, as does shrinking, according to [57].

However, we have seen that shrinking is extremely useful within Eureka, which employs all the above-mentioned techniques, and Minisat with local restarts, which uses some of them. Thus empirically the effect of shrinking is not achieved by combining other techniques. Let us take a closer look at the differences between our basic version of shrinking and the combination of frequent restarts, CBH, and RSAT's polarity selection heuristic. First, the shrinking condition differs from the restart condition of any known restart strategy. Second, shrinking restarts the search only partially, in contrast to most modern restart strategies. Third, unlike clause-based heuristics, shrinking continues selecting variables from the last conflict clause, even if it is satisfied. Fourth, shrinking re-orders the variables in the last conflict clause. It is, therefore, the simultaneous effect of these features, achieved by carefully choosing the shrinking condition, the sorting scheme, and the shrinking backtrack level, that makes shrinking highly efficient.

3.4 Conclusion

Assignment stack shrinking is a technique that boosts the performance of modern complete SAT solvers by making them more dynamic and local, and by enhancing the interrelation of the assigned variables. We have described in detail different variations of the shrinking algorithm, including two new heuristics, one of which improves Minisat's overall performance. We have shown that shrinking is extremely efficient within Minisat and Eureka, and that its effects cannot be achieved by other modern algorithms. Shrinking is proving to be a useful concept (that is, a collective name for a family of algorithms) that can be enhanced independently of the other components of SAT solvers, such as restart strategies or decision heuristics.

Chapter 4

Preprocessing in Incremental SAT

Alexander Nadel[1], Vadim Ryvchin[1,2] and Ofer Strichman[2]
[1] *Intel Corporation, P.O. Box 1659, Haifa 31015 Israel*
[2] *Information Systems Engineering, IE, Technion, Haifa, Israel*

Abstract

Preprocessing of CNF formulas is an invaluable technique when attempting to solve large formulas, such as those that model industrial verification problems. Unfortunately, the best combination of preprocessing techniques, which involve variable elimination combined with subsumption, is incompatible with incremental satisfiability. The reason is that soundness is lost if a variable is eliminated and later reintroduced. *Look-ahead* is a known technique to solve this problem, which simply blocks elimination of variables that are expected to be part of future instances. The problem with this technique is that it relies on knowing the future instances, which is impossible in several prominent domains. We show a technique for this realm, which is empirically far better than the known alternatives: running without preprocessing at all or applying preprocessing separately at each step.

4.1 Introduction

Whereas CNF preprocessing techniques have been known for a long time (e.g., [5,12]), most are not cost-effective when it comes to formulas with millions of clauses – a typical size for industrial verification problems that are being routinely solved these days in the EDA industry. In that respect one of the major breakthroughs in practical SAT solving in the last few years has been the combined preprocessing techniques that were suggested by Een and Biere [30]: non-increasing variable elimination through resolution, coupled with subsumption and self-subsumption. These three techniques remove variables, clauses and literals, respectively. They are implemented in MiniSat [32] and the stand-alone preprocessor SatELite, and are in common use by many SAT solvers. Our experience with industrial verification instances shows that these techniques frequently remove more than half of the formula, and enable the solving of large instances that otherwise cannot be solved within a reasonable time limit. We will describe these techniques in more detail in Section 4.2.

A known problem with variable elimination is the fact that it is incompatible, at least in its basic form as published, with incremental SAT solving [32,81,92]. The reason, as was pointed out already in [30], is that variables that are eliminated may reappear in future instances. Soundness is not maintained in this scenario. For example, suppose that a formula contains the two clauses $(a \vee v), (b \vee \bar{v})$. Eliminating v results in removing these two clauses and adding the resolvent $(a \vee b)$. Suppose, now, that in the next instance the clauses $(\bar{a}), (\bar{v})$ are added, which clearly contradict $(a \vee v)$. Yet since we erased that clause and since there is no contradiction between the resolvent and the new clauses, the new formula is possibly satisfiable — soundness is lost.

A possible remedy to this problem which was already suggested in [30] and experimented with in [50], is *look-ahead*. This means that variables that are known to be added in future instances are not eliminated. The problem with look-ahead is that it is not always possible, because information about future instances is not always available. Examples of such problem domains are:

- Some applications require interactive communication with the user for determining the next portion of the problem. For example, a recent article from IBM [3] describes a process in which the verification engineer may re-invoke the same instance of the SAT-based model checker for verifying a new property, which is not known a-priory (it depends on the result of the previous property). In such a case only a small part of the formula is changed, and hence incremental satisfiability may be crucial for performance.

- In some applications the calculation of the next portion of the problem depends on the results of the previous invocation of the SAT solver. For example, various tasks in MicroCode validation [35] are solved by using a symbolic execution engine to explore the paths of the program. The generated proof obligations are solved by an incremental SAT-based SMT solver. In this application, the next explored path of the program is determined based on the result of the previous computation.

- In Intel, the conversion of BMC problems to CNF is done after applying a 'saturation' optimization at the circuit level. Saturation divides all the variables into equivalence classes and tries to unite them by propagating short clauses that were learned in a previous instance — hence the dependency that prevents precalculating the instances. The SAT solver is provided only with the representatives of the equivalence classes. As a result, simple unrolling cannot predict those variables that will be present or absent in future instances.

Another possible remedy is called *full preprocessing*. It was briefly mentioned in [50] as an option that is expected not to scale, although in our experiments it is occasionally competitive. The idea is to perform full preprocessing before each instance. This means that all variables that were previously eliminated are returned to the formula and resolvents are removed, other than those that subsumed other clauses and hence cannot be removed. Therefor preprocessing is performed independently of past or future instances, other than the fact that it marks subsuming resolvents. The disadvantage of this approach comparing to *incremental preprocessing* — the main contribution

of this article — is that it repeats a lot of work that has already been done in previous instances. Our experiments with large instances show that this extra overhead can add more than an hour to the preprocessing time.

In this article we suggest a method for combining the method of [30] with assumptions-based incremental SAT [32]. Our experiments show that it is much better than either running without preprocessing at all or full preprocessing. Look-ahead is still better overall, however, when possible. The solution we suggest is simple and rather easy to implement. Basically we eliminate variables regardless of future instances, and every time a variable is reintroduced into the formula we choose whether to *reeliminate*, or *reintroduce* it. An exception is made for the assumptions variables, which must be reintroduced. For both routes we need to save the clauses that were erased in the process of elimination: these need to be resolved with the new clauses for the former, and returned to the formula for the latter. As we show, the *order* in which variables are reeliminated or reintroduced matters for correctness. Specifically, the order must be *consistent* between instances. The order also changes the resulting reduced formula and hence the solving time. Our experiments show that in most cases the consistent order reduces the solving time.

We continue in the next section by describing the technical details of variable elimination, subsumption and self-subsumption. In Section 4.3 we present incremental preprocessing, which is an adaptation of these algorithms to the setting of incremental SAT. In Section 4.4 we summarize the results of our extensive experiments with industrial verification benchmarks from Intel.

4.2 Preliminaries

Let φ be a CNF formula. We denote by $vars(\varphi)$ the variables used in φ. For a clause c we write $c \in \varphi$ to denote that c is a clause in φ. For $v \in vars(\varphi)$ we define $\varphi_v = \{c \mid c \in \varphi \wedge v \in c\}$ and $\varphi_{\bar{v}} = \{c \mid c \in \varphi \wedge \bar{v} \in c\}$ (somewhat abusing notation, as we refer here to v as both a variable and a literal). Our setting includes the use of *assumptions* [32].

Variable elimination

Input: formula φ and a variable $v \in vars(\varphi)$.
Output: formula φ' such that $v \notin vars(\varphi')$ and φ' and φ are equisatisfiable.

Typically this preprocessing is applied only if the number of clauses in φ' is not larger than in φ. More generally one may define a positive limit on the growth in the number of clauses, but for simplicity we will assume here that this limit is 0. Alg. 3 presents a variable elimination algorithm, where the eliminated variable v is the parameter. The variable v must be unassigned.

Algorithm 3 A variable elimination algorithm similar to the one implemented in MiniSat and in [30].

```
1: function Resolve(clauseset pos, clauseset neg)
2:     clauseset res = ∅;
3:     for each clause p ∈ pos do
4:         for each clause n ∈ neg do
5:             if p and n have a single possible pivot then
6:                 res = res ∪ resolution(p, n);
7:     return res;

1: function EliminateVar(var v)
2:     clauseset Res = Resolve (φ_v, φ_v̄);
3:     if |Res| > |φ_v| + |φ_v̄| then return ∅;          ▷ no variable elimination
4:     φ = (φ ∪ Res) \ (φ_v ∪ φ_v̄);
5:     ClearDataStructures(v);  ▷ clearing occurrence list, watch-list, scores-list
6:     TouchedVars = TouchedVars ∪ vars(Res);          ▷ used in Alg. 4
7:     return Res;
```

The function Resolve computes the set of non-tautological resolvents of two sets of clauses given to it as input (the check in line 5 excludes tautological resolvents). Function EliminateVar uses Resolve to compute the set Res of such resolvents of φ_v and $\varphi_{\bar{v}}$. If this set is larger than $|\varphi_v| + |\varphi_{\bar{v}}|$ it simply returns, and hence v is not eliminated. Otherwise in line 4 it adds the resolvents Res and discards the resolved clauses. All the variables in the resolvents are added to a list $TouchedVars$ in line 6. This list will be used

later, in Alg. 4, for driving further subsumption and self-subsumption.

Subsumption

Input: $\varphi \wedge (l_1 \vee \cdots \vee l_i) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j)$.
Output: $\varphi \wedge (l_1 \vee \cdots \vee l_i)$.

Self-subsumption

Input: $\varphi \wedge (l_1 \vee \cdots \vee l_i \vee l) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j \vee \bar{l})$.
Output: $\varphi \wedge (l_1 \vee \cdots \vee l_i \vee l) \wedge (l_1 \vee \cdots \vee l_i \vee l_{i+1} \vee \cdots \vee l_j)$.

Preprocessing

The preprocessing algorithm described in Alg. 4 is similar to that implemented in MiniSat 2.2 [32] (based on the stand-alone preprocessor SatELite [30]). $SubsumptionQ$ is a global queue of clauses. For each $c \in SubsumptionQ$, and each $c' \in \varphi$, RemoveSubsumptions (1) checks if $c \subset c'$ and if yes performs subsumption, and otherwise (2) if c self-subsumes c' then it performs self-subsumption. Essentially it is similar to the implementation suggested in [30]. Self-subsumption is followed by adding the reduced clause back to the queue. The function runs until the queue is empty. Note that assumptions are not eliminated. Eliminating assumptions would render the algorithm unsound.

In line 5 the variables are scanned in an increasing order of occurrences count. Note that in line 7 RemoveSubsumptions is applied only to the set of newly generated resolvents.

4.3 Incremental preprocessing

We now describe an incremental version of the preprocessing algorithm. In contrast to the full-preprocessing algorithm that was briefly described in the introduction (performing preprocessing of the new formula, together with

Algorithm 4 Preprocessing, similar to the algorithm implemented in MiniSat 2.2.

```
1: function PREPROCESS
2:     SubsumptionQ = φ;
3:     while SubsumptionQ ≠ ∅ do
4:         REMOVESUBSUMPTIONS ();
5:         for each unassigned non-assumption variable v do          ▷ order
       heuristically
6:             SubsumptionQ = ELIMINATEVAR (v);
7:             if SubsumptionQ ≠ ∅ then REMOVESUBSUMPTIONS ();
8:         SubsumptionQ = {c | vars(c) ∩ TouchedVars ≠ ∅};
9:         TouchedVars = ∅;
```

learned clauses from previous instances), our suggested algorithm does not repeat preprocessing work that was done in previous instances.

In our setting of incremental SAT, each instance is given as a set of clauses that should be added to the formula accumulated thus far. Removal of clauses is done indirectly, by using assumptions that are clause selectors. For example, if v is an assumption variable, then we can add $\bar{v}$ to a set of clauses. Assigning this variable FALSE is equivalent to removing this set.

Let φ^0 denote the initial formula, and Δ^i denote the set of clauses added at step i. Step i for $i > 0$ begins with a formula denoted φ^i, initially assigned the conjunction of φ^{i-1} at the *end* of the solving process (i.e., after being preprocessed and with additional learned clauses), and Δ^i. This formula changes during the solving process.

Preprocessing in an incremental SAT setting requires various changes. In step i, the easy case is when we wish to eliminate a variable v that is *not* eliminated in step $i-1$. ELIMINATEVAR-INC, shown in Alg. 5 is a slight variation of ELIMINATEVAR that we saw in Alg. 3. The only difference is that if v is eliminated, then it saves additional data that will be used later on, as we will soon see. Specifically, it saves φ^i_v and $\varphi^i_{\bar{v}}$ in clause-sets denoted respectively by S_v and $S_{\bar{v}}$, and in the next line also the number of resolvents in a queue called $ElimVarQ$. This queue holds tuples of the form $\langle$variable v, int *resolvents*$\rangle$.

Algorithm 5 Variable elimination for φ^i, where the eliminated variable v was *not* eliminated in φ^{i-1}.

```
1: function EliminateVar-inc(var v, int i)
2:     clauseset Res = Resolve (φ^i_v, φ^i_v̄);
3:     if |Res| > |φ^i_v| + |φ^i_v̄| then return ∅;          ▷ no variable elimination
4:     S_v = φ^i_v; S_v̄ = φ^i_v̄;                          ▷ Save for possible reintroduction
5:     ElimVarQ.push(⟨v, |Res|⟩);                          ▷ Save #resolvents in queue
6:     φ^i = (φ^i ∪ Res) \ (φ^i_v ∪ φ^i_v̄);
7:     ClearDataStructures (v);
8:     TouchedVars = TouchedVars ∪ vars(Res);              ▷ used in Alg. 7
9:     return Res;
```

The more difficult case is when v is already eliminated at step $i-1$. In that case we invoke ReEliminate-Or-ReIntroduce, as shown in Alg. 6. This function decides between reintroduction and reelimination.

- *Reelimination.* In Line 6 the algorithm computes the set of resolvents Res that need to be added in case v is reeliminated. Note that φ^i may contain v because of two separate reasons. First, $vars(\Delta^i)$ may contain v; Second, variables that were reintroduced in step i *prior* to v may have led to reintroduction of clauses that contain v. The total number of resolvents resulting from eliminating v is $|Res|$ + the number of resolvents incurred by eliminating v up to step i, which, recall, is saved in $ElimVarQ$.

- *Reintroduction.* In case we decide to cancel elimination, the previously removed clauses S_v and $S_{\bar{v}}$ have to be reintroduced. The total number of clauses resulting from reintroducing v is thus $|S_v \cup S_{\bar{v}} \cup \varphi^i_v \cup \varphi^i_{\bar{v}}|$. Note that the algorithm reintroduces variables that appear in the assumption list.

The decision between the two options is made in line 7. If reintroduction results in a smaller number of clauses, we simply return the saved clauses S_v and $S_{\bar{v}}$ by calling ReIntroduceVar, which also removes its entry from $ElimVarQ$ because v is no longer eliminated. The rest of the code is self-explanatory.

Algorithm 6 Variable elimination for φ^i, where the eliminated variable (located in $ElimVarQ[loc].v$) was already eliminated in φ^{i-1}.

1: **function** ReIntroduceVar(var v, int loc, int i)
2: $\varphi^i \mathrel{+}= S_v \cup S_{\bar{v}}$;
3: erase $ElimVarQ[loc]$; ▷ v is not eliminated, hence 0 resolvents

1: **function** ReEliminateVar(clauseset Res, var v, int loc, int i)
2: $S_v = S_v \cup \varphi^i_v$; $S_{\bar{v}} = S_{\bar{v}} \cup \varphi^i_{\bar{v}}$;
3: $ElimVarQ[loc].resolvents \mathrel{+}= |Res|$;
4: $\varphi^i = (\varphi^i \cup Res) \setminus (\varphi^i_v \cup \varphi^i_{\bar{v}})$;
5: ClearDataStructures (v);
6: $TouchedVars = TouchedVars \cup vars(Res)$;

1: **function** ReEliminate-Or-ReIntroduce(int loc, int i)
2: var $v = ElimVarQ[loc].v$; ▷ The variable to eliminate
3: **if** v is an assumption **then**
4: ReIntroduceVar(v, loc, i);
5: **return** $\emptyset$;
6: clausesetRes = Resolve$(\varphi^i_v, \varphi^i_{\bar{v}})$ $\cup$ Resolve$(\varphi^i_v, S_{\bar{v}})$ $\cup$ Resolve$(S_v, \varphi^i_{\bar{v}})$;
7: **if** $(|Res| + ElimVarQ[loc].resolvents) > |S_v \cup S_{\bar{v}} \cup \varphi^i_v \cup \varphi^i_{\bar{v}}|$ **then**
8: ReIntroduceVar(v, loc, i);
9: **return** $\emptyset$;
10: ReEliminateVar (Res, v, loc, i);
11: **return** Res

Given EliminateVar-Inc and ReEliminate-Or-ReIntroduce we can now focus on Preprocess-inc in Alg. 7, which is parameterized by the instance number i. The difference from Alg. 4 is twofold: First, variables that are already eliminated in the end of step $i-1$ are processed by ReEliminate-Or-ReIntroduce; Second, other variables are processed in EliminateVar-inc. The crucial point here is the *order* in which variables are eliminated. Note that 1) elimination is consistent between instances, and 2) variables that are not currently eliminated are checked for elimination only at the end. These two conditions are necessary for correctness, because, recall, ReIntroduceVar may return clauses that were previously erased. These clauses may contain any variable that was not eliminated at the time they were erased.

Example 4.3.1 *Suppose that in step $i-1$, v_1 was eliminated, and as a result a clause $c = (v_1 \vee v_2)$ was removed. Then v_2 was eliminated as well. Suppose now that in step i we first reeliminate v_2, and then decide to reintroduce v_1. The clause c above is added back to the formula. But c contains v_2 which was already eliminated.*

Let $\psi^n = \varphi^0 \wedge \bigwedge_{i=1}^{n} \Delta^i$, i.e., ψ_n is the n-th formula without preprocessing at all. We claim that:

Proposition 4.3.1 *Algorithm* Preprocess-inc *is correct, i.e., for all n*

$$\psi^n \text{ is equisatisfiable with } \varphi^n \ .$$

Proof. The full proof is given in a technical report [63]. Here we only sketch its main steps. The proof is by induction on n. The base case corresponds to standard (i.e., non-incremental) preprocessing. Proving the step of the induction relies on another induction, which proves that the following two implications hold right after line 7 at the j-th iteration of the first loop in Preprocess-inc, for $j \in [0 \ldots |ElimVarQ| - 1]$:

$$\psi^n \implies (\varphi^n \wedge \bigwedge_{k=j+1}^{|ElimVarQ|-1} \bigwedge_{c \in S_{v_k} \cup S_{\bar{v}_k}} c) \implies \exists v_1 \ldots v_j .\ \psi^n \ ,$$

Algorithm 7 Preprocessing in an incremental SAT setting

```
1: function PREPROCESS-INC(int i)                          ▷ preprocessing of φ^i
2:     SubsumptionQ = {c | ∃v. v ∈ c ∧ v ∈ vars(Δ^i)};
3:     REMOVESUBSUMPTIONS ();
4:     for (j = 0 ... |ElimVarQ| − 1) do    ▷ scanning eliminated vars in order
5:         v = ElimVarQ[j].v;
6:         if |φ^i_v| = |φ^i_v̄| = 0 then continue;
7:         REELIMINATE-OR-REINTRODUCE (j, i);
8:     while SubsumptionQ ≠ ∅ do
9:         for each non-assumption variable v ∉ ElimVarQ do   ▷ scanning the
       rest
10:            SubsumptionQ = ELIMINATEVAR-INC (v, i);
11:            REMOVESUBSUMPTIONS ();
12:        SubsumptionQ = {c | vars(c) ∩ TouchedVars ≠ ∅};
13:        TouchedVars = ∅;
```

The implication on the right requires some attention: existential quantification is necessary because of variable elimination via resolution (in the same way that $Res(x \vee A)(\bar{x} \vee B) = (A \vee B)$ and $(A \vee B) \Longrightarrow \exists x.\ (x \vee A)(\bar{x} \vee B)$). The crucial point in the proof of this implication is to show that if a variable is eliminated at step j, it cannot reaapear in the formula in later iterations. This is indeed guaranteed by the order in which the first loop processes the variables.

Note that at the last iteration $j = |ElimVarQ| - 1$ and the big conjunctions disappear. This leaves us with

$$\psi^n \quad \Longrightarrow \quad \varphi^n \quad \Longrightarrow \quad \exists v_1 \dots v_j.\ \psi^n\ ,$$

which implies that ψ^n is equisatisfiable with the formula after the last iteration. The second loop of PREPROCESS-INC is non-incremental preprocessing, and hence clearly maintains satisfiability.

Removal of resolvents

Recall that ReIntroduceVar returns the clause sets S_v and $S_{\bar{v}}$ to the formula. So far we ignored the question of what to do with the resolvents: should we remove them given that we canceled the elimination of v? These clauses are implied by the original formula, so keeping them does not hinder correctness. *Removing* them, however, is not so simple, because they may have participated in subsumption / self-subsumption of other clauses. Removing them hinders soundness, as demonstrated by the following example.

Example 4.3.2 *Consider the following four clauses:*

$$\begin{array}{ll} c_1 = (l_1 \vee l_2 \vee l_3) & c_2 = (l_4 \vee l_5 \vee \bar{l_3}) \\ c_3 = (l_1 \vee l_2 \vee \bar{l_4}) & c_4 = (l_1 \vee l_2 \vee \bar{l_5}) \ , \end{array}$$

and the following sequence:

- *elimination of* $var(l_3)$*:*
  - $c_5 = res(c_1, c_2) = (l_1 \vee l_2 \vee l_4 \vee l_5)$ *is added;*
  - c_1 *and* c_2 *are removed and saved.*
- *self-subsumption between* c_3 *and* c_5*:* $c_5 = (l_1 \vee l_2 \vee l_5)$.
- *self-subsumption between* c_4 *and* c_5*:* $c_5 = (l_1 \vee l_2)$.
- *subsumption of* c_3 *and* c_4 *by* c_5.
- *removal of the resolvent* c_5 *and returning of* c_1 *and* c_2.

We are left with only a subset of the original clauses (c_1 *and* c_2*), which do not imply the rest. In this case the original formula is satisfiable, but it is not hard to see that the subsumed clauses (*c_3, c_4*) could have been part of an unsatisfiable set of clauses, and hence that their removal could have changed the result from unsat to sat. Soundness is therefore not secured if resolvents that participated in subsumption are removed.*

In our implementation we solve this problem as follows. When eliminating v, we associate all the resolvent clauses with v. In addition, we mark all clauses that subsumed other clauses. We then change ReIntroduceVar as can be seen in Alg. 8. Note that in line 3 we guarantee that unit resolvents remain: it does not affect correctness and is likely to improve performance.

Algorithm 8 ReIntroduceVar with removal of resolvents that did not participate in subsumption.

```
1: function ReIntroduceVar(var v, int loc, int i)
2:     φ^i += S_v ∪ S_v̄;
3:     for each non-unit clause c associated with v do
4:         if c is not marked then Remove c from φ^i;
5:     erase ElimVarQ[loc];
```

4.4 Experimental results

We implemented incremental preprocessing on top of Fiver[1], and experimented with hundreds of large processor Bounded Model-checking instances from Intel, categorized to four different families. In each case the problem is defined as performing BMC up to a given bound[2] in increments of size 1, or finding a satisfying assignment on the way to that bound. The time out was set to 4000 sec. After removing those benchmarks that cannot be solved by any of the tested methods within the time limit we were left with 206 BMC problems.[3] We turned off the 'saturation' optimization at the circuit level that was described in the introduction, in order to be able to compare our results to look-ahead. Overall in about half of the cases there is no satisfying assignment up to the given bound.

The first graph, in Figure 4.1, summarizes the overall results of the four compared methods: full-preprocessing, no-preprocessing, incremental-

[1] Fiver is a new SAT solver that was developed in Intel. It is a CDCL solver, combining techniques from Eureka, Minisat, and other modern solvers.

[2] Internal customers in Intel are typically interested in checking properties up to a given bound.

[3] The benchmarks are available upon request from the authors.

Method	Time-outs	Avg. total run-time
full-preprocessing	68	2465.5
no-preprocessing	42	1784.7
incremental-preprocessing	2	1221.3
look-ahead	0	1064.9

Table 4.1: The number of time-outs and the average total run time (incl. preprocessing) achieved by the four compared methods.

preprocessing, and look-ahead. The number of time-outs and the average total run-time with these four methods is summarized in Table 4.1.

Look-ahead wins overall, but recall that in this article we focus on scenarios in which lookahead is impossible. Also note that it only has an advantage in a setting in which there is a short time-out. Incremental-preprocessing is able to close the gap and become almost equivalent once the time-out is set to a high value. It seems that the reason for the advantage of incremental preprocessing over look-ahead in hard instances is that unlike the latter, it does not force each variable to stay in the formula until it is known that it will not be added from thereon.

We now examine the results in more detail. Figure 4.2 shows the consistent benefit of incremental preprocessing over full preprocessing. The generated formula is not necessarily the same because of the order in which the variables are examined. Recall that it is consistent between instances in Preprocess-Inc and gives priority to those variables that are currently eliminated. In full preprocessing, on the other hand, it checks each time the variable that is contained in the minimal number of clauses. The impact of the preprocessing order on the search time is inconsistent, but there is a slight advantage to that of Preprocess-Inc, as can be seen in the middle figure. The overall run time favors Preprocess-Inc, as can be seen at the bottom figure.

Figure 4.3 compares incremental preprocessing and no preprocessing at all. Again, the advantage of the former is very clear.

Finally, Figure 4.4 compares incremental preprocessing and look-ahead, which shows the benefit of knowing the future. The fact that the preprocess-

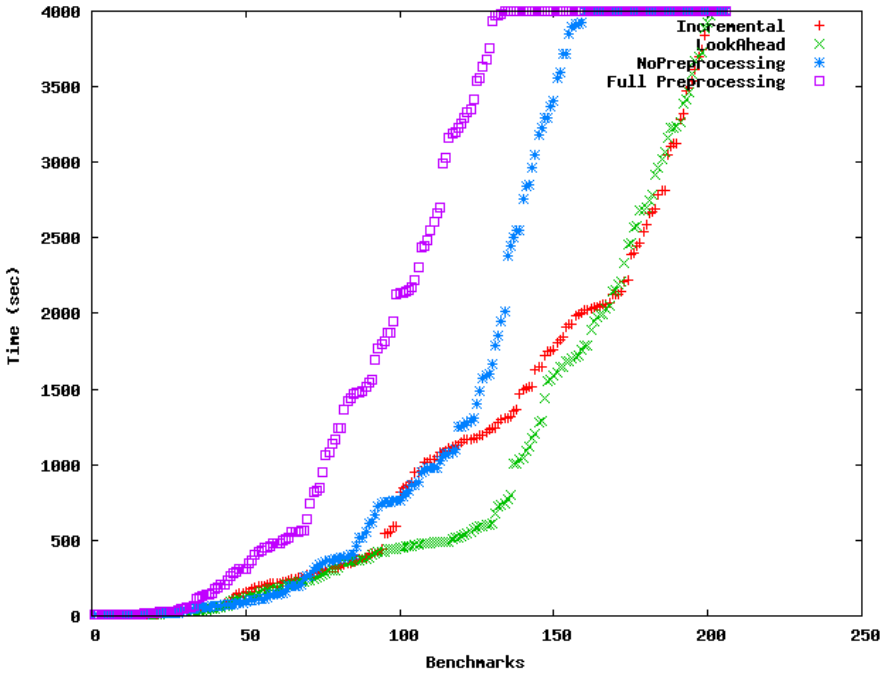

Figure 4.1: Overall run-time of the four compared methods.

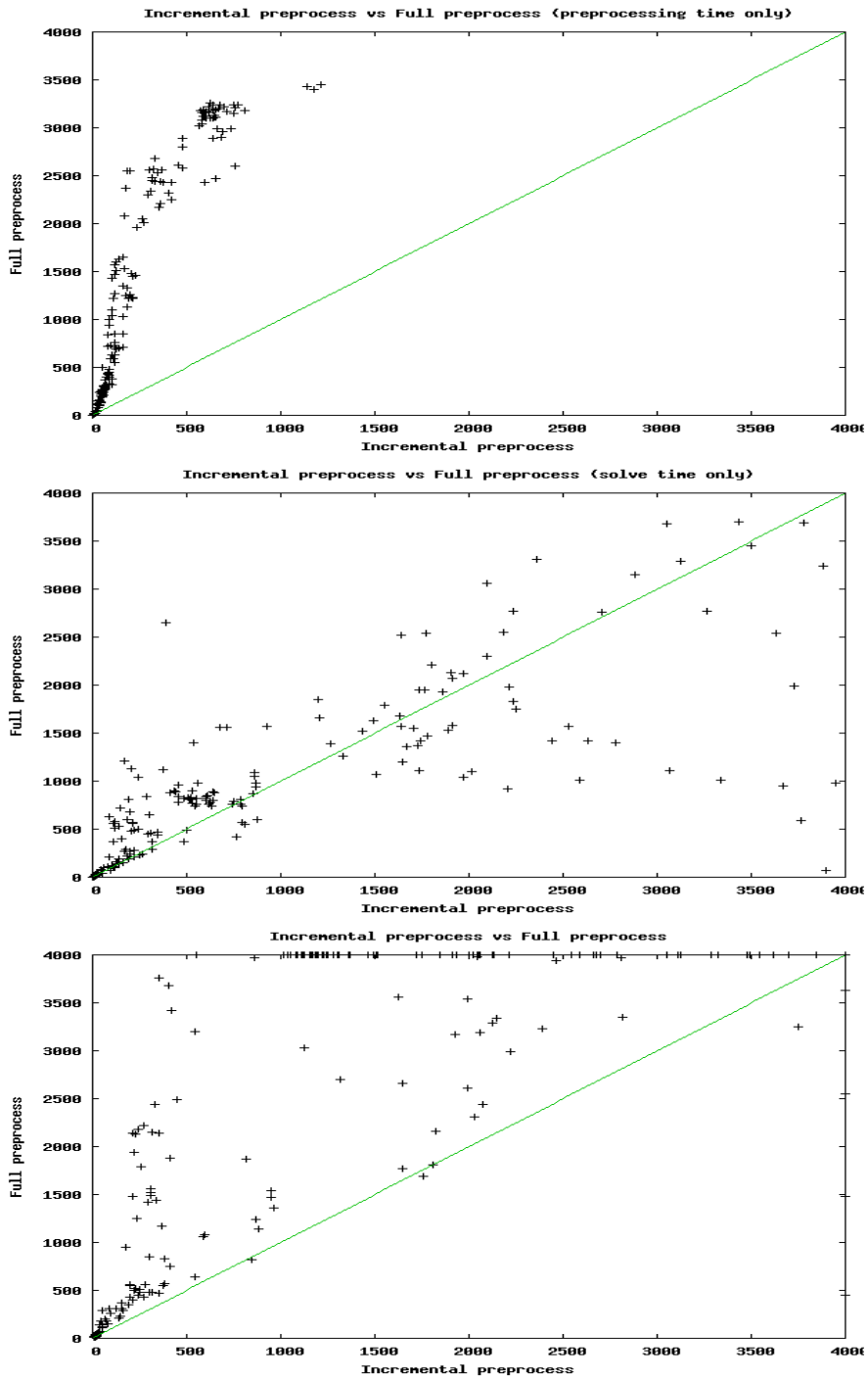

Figure 4.2: Incremental preprocessing vs. full preprocessing: (top) preprocessing time, (middle) SAT time, and (bottom) total time.

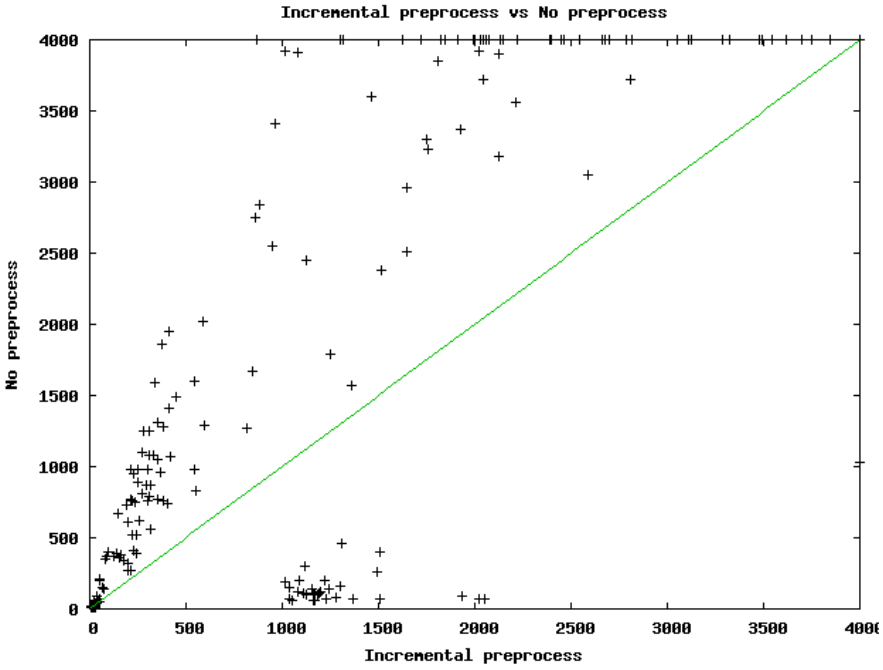

Figure 4.3: Incremental preprocessing vs. no-preprocessing.

ing time of the latter is smaller is very much expected, because it does not have the overhead incurred by the checks in Alg. 5 and the multiple times that each variable can be reeliminated and reintroduced. The last graph shows that a few more instances were solved overall faster with look-ahead, but recall that according to Figure 4.1 with a long-enough timeout the two methods have very similar results in terms of the number of solved instances.

4.5 Conclusion

In various domains there is a need for incremental SAT, but the sequence of instances cannot be computed apriori, because of dependance on the result of previous instances. In such scenarios applying preprocessing with look-ahead, namely preventing elimination of variables that are expected to be reintroduced, is impossible. Incremental preprocessing, the method we suggest here, is an effective algorithm for solving this problem. Our experiments with hundreds of industrial benchmarks show that it is much faster than the two known alternatives, namely full-preprocessing and no-preprocessing. Specifically, with a time-out of 4000 sec. it is able to reduce the number of time-outs by a factor of four and three, respectively.

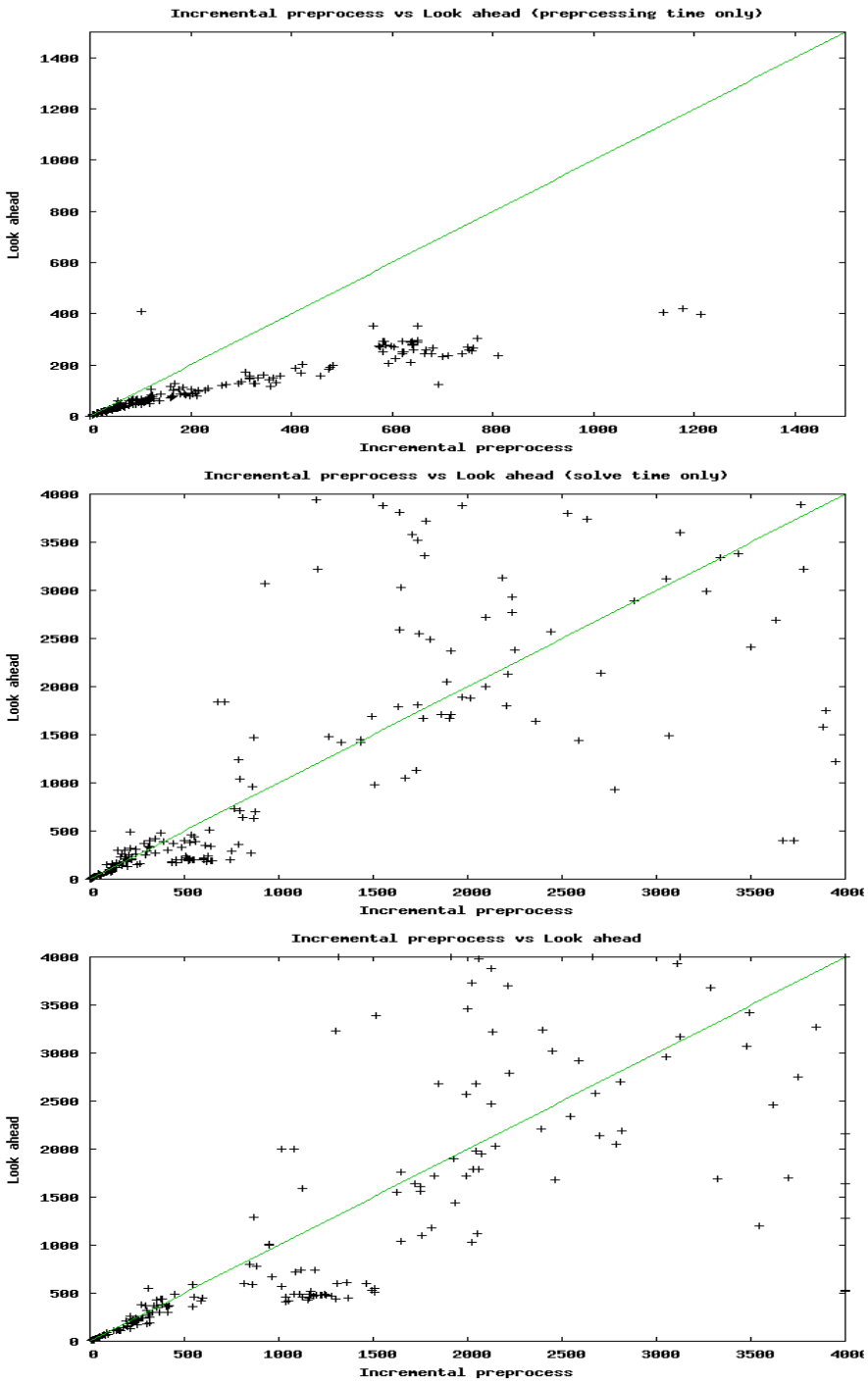

Figure 4.4: Incremental preprocessing vs. look-ahead: (top) preprocessing time, (middle) SAT time, and (bottom) total time.

Chapter 5

Efficient SAT Solving under Assumptions

Alexander Nadel[1], Vadim Ryvchin[1,2]
[1] *Intel Corporation, P.O. Box 1659, Haifa 31015 Israel*
[2] *Information Systems Engineering, IE, Technion, Haifa, Israel*

Abstract

In incremental SAT solving, assumptions are propositions that hold solely for one specific invocation of the solver. Effective propagation of assumptions is vital for ensuring SAT solving efficiency in a variety of applications. We propose algorithms to handle assumptions. In our approach, assumptions are modeled as unit clauses, in contrast to the current state-of-the-art approach that models assumptions as first decision variables. We show that a notable advantage of our approach is that it can make preprocessing algorithms much more effective. However, our initial scheme renders assumption-dependent (or temporary) conflict clauses unusable in subsequent invocations. To resolve the resulting problem of reduced learning power, we introduce an algorithm that transforms such temporary clauses into assumption-independent pervasive clauses. In addition, we show that our approach can be enhanced further when a limited form of look-ahead information is available. We demonstrate that our approach results in a considerable performance boost of the SAT solver on instances generated by a prominent industrial application in hardware validation.

5.1 Introduction

A variety of SAT applications require the ability to solve incrementally generated SAT instances online [22,31,33,35,84,88,93]. In such settings the solver is expected to be invoked multiple times. Each time it is asked to check the satisfiability status of all the available clauses under *assumptions* that hold solely for one specific invocation. The naïve algorithm which solves the instances independently is inefficient, since all learning is lost [33, 84, 88, 93].

The current state-of-the-art approach to this problem was proposed in [33] and implemented in the MiniSat SAT solver [32]. MiniSat reuses a single SAT solver instance for all the invocations. Each time after solving is completed, the user can add new clauses to the solver and reinvoke it. The user is also allowed to provide the solver a set of *assumption literals*, that is, literals that are always picked as the first decision literals by the solver. In this scheme, all the conflict clauses generated are *pervasive*, that is, assumption-independent. We call this approach to the problem of incremental SAT solving under assumptions the *Literal-based Single instance (LS)* approach, since it reuses a single SAT solver instance and models assumptions as decision literals. The approach of [84] to our problem would use a separate SAT solver instance for each invocation, where each assumption would be encoded as a unit clause. To increase the efficiency of learning, it would store and reuse the set of assumption-independent *pervasive* conflict clauses throughout all the SAT invocations. We call this approach the *Clause-based Multiple instances (CM)* approach, since it uses multiple SAT solver instances and models assumptions as unit clauses.

It was shown in [33] that LS outperforms CM in the context of model checking. As a result, LS is currently widely applied in practice (e.g. [22, 31, 35]). The goal of this paper is to demonstrate its limitations and to propose an efficient alternative.

This study springs from the authors' experiences, described herein, in tuning Intel's formal verification flow. Verification engineers reported to us that a critical property could not be solved by the SAT solver within two days. Our default flow used the LS approach, where to check a property the property's negation is provided as an assumption. The property holds iff

the instance is unsatisfiable. Surprisingly, we discovered that providing the negation of the property as a unit clause, rather than as an assumption, rendered the property solvable within 30 minutes. The reason for this was that the unit clause triggered a huge simplification chain for our SatELite [30]-like preprocessor that drastically reduced the number of clauses in the formula.

Our experience highlights a drawback of LS: preprocessing techniques cannot propagate assumptions in LS, since they are modeled as decision variables, while assumptions can be propagated in CM, where they are modeled as unit clauses. Section 5.3 of this work demonstrates how to incorporate the SatELite algorithm within CM and shows why the applicability of SatELite for LS is an open problem.

LS has important advantages over CM related to the efficiency of learning. First, in LS all the conflict clauses are pervasive and can be reused, while CM cannot reuse *temporary* conflict clauses, that is, clauses that depend on assumptions. Second, LS reuses all the information relevant to guiding the SAT solver's heuristics, while CM has to gather relevant information from scratch for each new incremental invocation of the solver. Section 5.4 of this paper proposes an algorithm that overcomes the first of the above-mentioned drawbacks of CM: our algorithm transforms temporary clauses into pervasive clauses as a post-processing step. Section 5.5 introduces an algorithm that mitigates the second of the above-mentioned advantages of plain LS over CM, given that limited look-ahead information is available to the solver. In fact, we propose an algorithm that combines LS and CM to achieve the most efficient results.

We study the performance of algorithms for incremental SAT solving under assumptions on instances generated by a prominent industrial application in hardware validation, detailed in Section 5.2. Section 5.2 also provides some definitions and background information. Experimental results demonstrating the efficiency of our algorithms are provided in Section 5.6. We would like to emphasize that all the SAT instances used in this paper are publicly available from the authors. Section 5.7 concludes our work.

5.2 Background

An incremental SAT solver is provided with the input $\{F_i, A_i\}$ at each invocation i, where for each i, F_i is a formula in Conjunctive Normal Form (CNF) and $A_i = \{l_1, l_2, \ldots, l_n\}$ is a set (conjunction) of *assumptions*, where each assumption l_j is a unit clause (it is also a literal). Invocation i of the solver decides the satisfiability of $(\bigwedge_{j=1}^{i} F_j) \wedge A_i$. Intuitively, before each invocation the solver is provided with a new block of clauses and a set of assumptions. It is asked to solve a problem comprising all the clauses it has been provided with up to that moment under the set of assumptions relevant only to a single invocation of the solver. Modern SAT solvers generate *conflict clauses* by resolution over input clauses and previously generated conflict clauses. A clause α is *pervasive* if $(\bigwedge_{j=1}^{i} F_j) \rightarrow \alpha$, otherwise it is *temporary*.

The Clause-based Multiple instances (CM) approach [84] to incremental SAT solving under assumptions operates as follows. CM creates a new instance of a SAT solver for each invocation. Each invocation decides the satisfiability of $(\bigwedge_{j=1}^{i} F_j) \wedge (\bigwedge_{l=1}^{i-1} P_l) \wedge A_i$, where P_l is the set of pervasive conflict clauses generated at invocation l of the solver. To keep track of temporary and pervasive conflict clauses, the algorithm marks all the assumptions as temporary clauses and marks a newly generated conflict clause as temporary iff one or more temporary clauses participated in its resolution derivation.

The Literal-based Single instance (LS) approach [33] to incremental SAT solving under assumptions reuses the same SAT instance for all the invocations. The instance is always updated with a new block of clauses. The key idea is in providing the assumptions as *assumption literals*, that is, literals that are always picked as the first decision literals by the solver. Conflict-clause learning algorithms ensure that any conflict clause that depends on a set of assumptions will contain the negation of these assumptions. Hence, in LS all the conflict clauses are pervasive.

While all the algorithms for incremental SAT solving under assumptions discussed in this paper are application-independent, the experimental results section studies the performance of various algorithms on instances generated by the following prominent industrial application in hardware validation.

Assume that a verification engineer needs to formally verify a set of prop-

erties in some circuit up to a certain bound. Formal verification techniques cannot scale to large modern circuits, hence the engineer needs to select a sub-circuit and mimic the environment of the larger circuit by imposing assumptions (also called constraints) [47]. The engineer then invokes SAT-based Bounded Model Checking (BMC) [18] to verify a property under the assumptions. If the result is satisfiable, then either the environment is not set correctly, that is, assumptions are incorrect or missing, or there is a real bug. In practice the first reason is much more common than the second. To discover which of the possibilities is the correct one, the engineer needs to analyze the counter-example. If the reason for satisfiability lies in incorrect modeling of the environment, the assumptions must be modified and BMC invoked again. When one property has been verified, the engineer can move on to another. Practice shows that most of the validation time is spent in this process, which is known as the *debug loop*.

In the standard industrial BMC-based formal validation flow the model checker instance is built from scratch for each iteration of the debug loop. The key idea behind our solution is to take advantage of incremental SAT solving under assumptions *across multiple invocations of the model checker*. We keep only one instance of the model checker. For each invocation of BMC, given a transition system Ψ, a safety property Δ, and a set of assumptions Λ, we check whether Ψ satisfies Δ given Λ at each bound up to a given bound k using incremental SAT solving under assumptions, as follows. At each bound i, the transition system Ψ unrolled to i is translated to CNF and comprises the formula, while the set comprising the negation of Δ unrolled to i and the assumptions Λ unrolled to i is the set of assumptions provided to the SAT solver. We call our model checking algorithm *incremental Bounded Model Checking (BMC) under assumptions*.

Some recent works dedicated to BMC propose taking advantage of look-ahead information that is available, since the instance can be unrolled beyond the current bound [48, 50]. In particular, it is proposed in [50] to apply preprocessing, including SatELite [30], for LS-based BMC, where complete look-ahead information is required to ensure soundness, as variables that are expected to appear in the future must not be eliminated. The technique of [50] cannot be applied in our application, since it is unknown a priori

how the user would update the formula before subsequent invocations of the incremental model checker. The in-depth BMC algorithm of [48], which uses a limited form of look-ahead to boost BMC, served as an inspiration for our algorithm for incremental SAT solving under assumptions with step look-ahead, presented in Section 5.5.

5.3 Preprocessing under Assumptions

Preprocessing refers to a family of algorithms whose goal is to simplify the input CNF formula prior to the CDCL-based search in SAT. Preprocessing has commonly been applied in modern SAT solvers since the introduction of the SatELite preprocessor [30]. This section first explains why even a rather straightforward form of preprocessing, known as database simplification, is expected to be much more effective when used with CM as compared to LS. We then show that, unmodified, SatELite cannot be used with either CM or LS, and demonstrate how it can be modified so as to be safely used with CM.

Consider the following algorithm, which we call *database simplification* following MiniSat [32] notation: First, propagate unit clauses with Boolean Constraint Propagation (BCP). Second, eliminate satisfied clauses and falsified literals.

Database simplification is applied as an inprocessing step (that is, as an on-the-fly simplification procedure, applied at the global decision level) in modern SAT solvers [15, 32, 86]. It can be applied during preprocessing and inprocessing with either LS or CM without further modification. A key observation is that the efficiency of the first application of database simplification after a new portion of the incremental problem becomes available can be dramatically higher when assumptions are modeled as unit clauses (as in CM) rather than as assumption literals (as in LS). Indeed, database simplification takes full advantage of unit clauses by propagating them and eliminating resulting redundancies, while it does not take any advantage of assumption literals. In addition, variables representing assumptions are eliminated by database simplification with CM, but not with LS. Our experimental data, presented in Section 5.6, demonstrates that database simplification elimi-

nates significantly more clauses for CM than for LS, and that the average conflict clause length for LS is much greater than it is for CM. These two factors favor CM as compared to LS as they have a significant impact on the efficiency of BCP and the overall efficiency of SAT solving.

Consider now the preprocessing algorithm of SatELite [30]. SatELite is a highly efficient algorithm used in leading SAT solvers [15, 32, 86]. SatELite is composed of the following three techniques:

1. *Variable elimination*: for each variable v, the algorithm performs resolution between clauses containing v (denoted by V^+) and $\neg v$ (denoted by V^-). Let U be the set of resulting clauses. If the number of clauses in U is less than or equal to the number of clauses in $V^+ \cup V^-$, then the algorithm eliminates v by replacing $V^+ \cup V^-$ by U.

2. *Subsumption*: if a clause α is subsumed by the clause β, that is, $\beta \subseteq \alpha$, α is removed.

3. *Self-subsuming resolution*: if $\alpha = \alpha_1 \vee l$ and $\beta = \beta_1 \vee \neg l$, where α_1 is subsumed by β_1, then α is replaced by α_1.

It is unclear how to apply SatELite with LS, let alone make its performance efficient. It is currently unknown how to apply SatELite for incremental SAT solving, since eliminated variables may be reintroduced (unless full look-ahead information is available [50], which is not always the case). However, even if the problem of incremental SatELite is solved, it is still unclear how to efficiently propagate assumptions when SatELite is applied with LS. One cannot apply SatELite as is, since eliminating assumption literals would render the algorithm unsound. A simple solution for ensuring soundness would be *freezing* the assumption literals [33, 50], that is, not carrying out variable elimination for them. However, this solution has the same potential severe performance drawback as database simplification applied with LS as compared to CM: freezing assumptions is expected to have a significant negative impact on the preprocessor's ability to simplify the instance.

It is also unknown how SatELite can be applied with CM. The problem is that one has to keep track of pervasive and temporary clauses. Fortunately, we can propose a simple solution for this problem, based on the observation

that SatELite uses nothing but resolution. SatELite can be updated as follows to keep track of pervasive and temporary clauses. If a variable is eliminated, each new clause $\alpha = \beta_1 \otimes \beta_2$ is marked as temporary iff one of the clauses β_1 or β_2 is temporary (where $\otimes$ corresponds to an application of the resolution rule). Whenever self-subsuming resolution is applied, the new clause α_1 is temporary iff either α or β is temporary (this operation is sound since α_1 is a resolvent of α and β). No changes are required for subsumption.

5.4 Transforming Temporary Clauses to Pervasive Clauses

We saw in Section 5.3 that CM has an important advantage over LS: preprocessing is expected to be much more efficient for CM. However, LS has its own advantages. An important advantage is efficiency of learning: all the conflict clauses learned by LS are pervasive, hence they can always be reused. In CM, all the temporary conflict clauses are lost. In this section we propose an algorithm that converts temporary clauses to pervasive clauses as a post-processing step after the SAT solver is invoked. Our algorithm overcomes the above-mentioned disadvantage of CM as compared to LS.

We start by providing some resolution-related definitions. The *resolution rule* states that given clauses $\alpha_1 = \beta_1 \vee v$ and $\alpha_2 = \beta_2 \vee \neg v$, where β_1 and β_2 are also clauses, one can derive the clause $\alpha_3 = \beta_1 \vee \beta_2$. The resolution rule application is denoted by $\alpha_3 = \alpha_1 \otimes^v \alpha_2$. A *resolution derivation* of a target clause α from a CNF formula $G = \{\alpha_1, \alpha_2, \ldots, \alpha_q\}$ is a sequence $\pi = (\alpha_1, \alpha_2, \ldots, \alpha_q, \alpha_{q+1}, \alpha_{q+2}, \ldots, \alpha_p \equiv \alpha)$, where each clause α_i for $i \leq q$ is *initial* and α_i for $i > q$ is *derived* by applying the resolution rule to α_j and α_k, where $j, k < i$.[1] A *resolution refutation* is a resolution derivation of the empty clause $\square$. Modern SAT solvers are able to generate resolution refutations given an unsatisfiable formula.

A resolution derivation π can naturally be considered as a directed acyclic graph (dag) whose vertices correspond to all the clauses of π and in which

[1] We force the resolution derivation to start with all the initial clauses, since such a convention is more convenient for the subsequent discussion.

there is an edge from a clause α_j to a clause α_i iff $\alpha_i = \alpha_j \otimes \alpha_k$ (an example of such a dag appears in Figure 5.1). A clause $\beta \in \pi$ is *backward reachable* from $\gamma \in \pi$ if there is a path (of 0 or more edges) from β to γ.

Assume now that the SAT solver is invoked over the CNF formula $A = \{\alpha_1 = l_1, \ldots, \alpha_n = l_n\} \wedge F = \{\alpha_{n+1}, \ldots, \alpha_r\}$ (where the first n clauses are temporary unit clauses corresponding to assumptions and the rest of the clauses are pervasive). Assume that the solver generated a resolution refutation π of $A \wedge F$. Let $\beta \in \pi$ be a clause. We denote by $\Gamma(\pi, \beta)$ the conjunction (set) of all the *backward reachable assumptions* from β, that is, the conjunction (set) of all the assumptions whose associated unit clauses are backward reachable from $\beta \in \pi$. Let $\Gamma(\beta)$ be short for $\Gamma(\pi, \beta)$. To transform any clause $\beta \in \pi \setminus A$ to a pervasive clause we propose applying the following operation:

$\boxed{T2P(\beta) = \beta \vee \neg\Gamma(\beta)}$

That is to say, we propose to update each temporary derived clause with the negations of the assumptions that were required for its derivation, while pervasive clauses are left untouched. Consider the example in Figure 5.1. The proposed operation would transform α_7 to $c \vee d \vee \neg a$; α_8 to $\neg d \vee \neg b$; α_{10} to $c \vee \neg a \vee \neg b$; and α_{11} to $\neg a \vee \neg b$. The pervasive clauses α_3, α_4, α_5, α_6, and α_9 are left untouched.

Alg. 9 shows how to transform a resolution refutation π of $A \wedge F$ to a resolution derivation $T2P(\pi)$ from F, such that every clause $\beta \in \pi \setminus A$ is mapped to a clause $T2P(\beta) = \beta \vee \neg\Gamma(\beta) \in T2P(\pi)$. The pre- and post-conditions that must hold for Alg. 9 appear at the beginning of its text. The second pre-condition is not necessary, but it makes the algorithm's formulation and correctness proof easier. The algorithm's correctness is proved below.

Proposition 5.4.1 *Algorithm 9 is sound, that is, its pre-conditions imply its post-conditions.*

Proof.

The proof is by induction on i, starting with $i = r + 1$. Both post-conditions hold when the "for" loop condition is reached when $i = r + 1$, since $T2P(\pi)$ comprises precisely the clauses of F at that stage. Indeed, every clause α_i visited until that point is initial and is mapped to $T2P(\alpha_i) = \alpha_i$ by construction. It is left to prove that both post-conditions hold each time after

Algorithm 9 Transform π to $T2P(\pi)$

Require: $\pi = (A = \{\alpha_1 = l_1, \ldots, \alpha_n = l_n\}, F = \{\alpha_{n+1}, \ldots, \alpha_r\}, \alpha_{r+1}, \ldots, \alpha_p)$ is a resolution refutation of $A \wedge F$
Require: All the assumptions in A are distinct and non-contradictory
Ensure: $T2P(\pi) = (T2P(\alpha_{n+1}), T2P(\alpha_{n+2}), \ldots, T2P(\alpha_r), T2P(\alpha_{r+1}), \ldots, T2P(\alpha_p))$ is a resolution derivation from F
Ensure: For each $i \in \{n+1, n+2, \ldots, r, \ldots, p\}$: $T2P(\alpha_i) = \alpha_i \vee \neg\Gamma(\alpha_i)$

```
1: for i ∈ {n+1, n+2, ..., p} do
2:     if α_i ∈ F then
3:         T2P(α_i) := α_i
4:     else
5:         Assume α_i = α_j ⊗^v α_k
6:         if α_j or α_k is an assumption then
7:             Assume without limiting the generality that α_j is the assumption
8:             T2P(α_i) := T2P(α_k)
9:         else
10:            T2P(α_i) := T2P(α_j) ⊗^v T2P(α_k)
11:    Append T2P(α_i) to T2P(π)
```

a derived clause $\alpha_i \in \pi$ is translated to $T2P(\alpha_i)$ and $T2P(\alpha_i)$ is appended to $T2P(\pi)$. We divide the proof into three cases depending on the status of α_i.

When α_i is a pervasive derived clause, its sources α_j and α_k must also be pervasive by definition. By induction, we have $T2P(\alpha_j) = \alpha_j$ and $T2P(\alpha_k) = \alpha_k$, since $\Gamma(\alpha_j)$ and $\Gamma(\alpha_k)$ are empty. Hence, $T2P(\alpha_i) = T2P(\alpha_j) \otimes^v T2P(\alpha_k) = \alpha_j \otimes^v \alpha_k$. Thus, it holds that $T2P(\alpha_i)$ is derived from F by resolution, so the first post-condition holds. We also have the second post-condition, since we have seen that $T2P(\alpha_i) = \alpha_j \otimes^v \alpha_k = \alpha_i$, while $\Gamma(\alpha_i)$ is empty.

Consider the case where α_i is temporary and α_j is an assumption. The second pre-condition of the algorithm ensures that α_k will not be an assumption. The algorithm's flow ensures that $T2P(\alpha_i) = T2P(\alpha_k)$. By induction, $T2P(\alpha_k)$ is derived from F by resolution, hence $T2P(\alpha_i)$ is also derived from F by resolution and the first post-condition holds. The induction hypothesis yields that $T2P(\alpha_i) = T2P(\alpha_k) = \alpha_k \vee \neg\Gamma(\alpha_k)$. It must hold that

$\alpha_k = \alpha_i \vee \neg l_j$, otherwise the resolution rule application $\alpha_i = (\alpha_j = l_j) \otimes^v \alpha_k$ would not be correct. Substituting the equation $\alpha_k = \alpha_i \vee \neg l_j$ into $T2P(\alpha_i) = \alpha_k \vee \neg\Gamma(\alpha_k)$ gives us $T2P(\alpha_i) = \alpha_i \vee \neg l_j \vee \neg\Gamma(\alpha_k) = \alpha_i \vee \neg(l_j \wedge \Gamma(\alpha_k))$. It must hold that $\Gamma(\alpha_i) = l_j \wedge \Gamma(\alpha_k)$ by resolution derivation construction. Substituting the latter equation into $T2P(\alpha_i) = \alpha_i \vee \neg(l_j \wedge \Gamma(\alpha_k))$ gives us precisely the second post-condition.

Finally consider the case where α_i is temporary and neither α_j nor α_k is an assumption. The first post-condition still holds after $T2P(\pi)$ is updated with $T2P(\alpha_i)$, since $T2P(\alpha_i) = T2P(\alpha_j) \otimes^v T2P(\alpha_k)$ by construction and both $T2P(\alpha_j)$ and $T2P(\alpha_k)$ are derived from F by resolution by the induction hypothesis. The induction hypothesis yields that $T2P(\alpha_i) = T2P(\alpha_j) \otimes^v T2P(\alpha_k) = (\alpha_j \vee \neg\Gamma(\alpha_j)) \otimes^v (\alpha_k \vee \neg\Gamma(\alpha_k))$. We have $\alpha_i = \alpha_j \otimes^v \alpha_k$. Hence, it holds that $T2P(\alpha_i) = (\alpha_j \otimes^v \alpha_k) \vee \neg\Gamma(\alpha_j) \vee \neg\Gamma(\alpha_k) = \alpha_i \vee \neg\Gamma(\alpha_j) \vee \neg\Gamma(\alpha_k)$. By resolution derivation construction, it holds that $\Gamma(\alpha_i) = \Gamma(\alpha_j) \wedge \Gamma(\alpha_k)$. Hence, $T2P(\alpha_i) = \alpha_i \vee \neg\Gamma(\alpha_i)$ and we have proved the second post-condition.

We implemented our method as follows. After SAT solving is completed, we go over the derived clauses in the generated resolution refutation π and associate each derived clause α with the set $\Gamma(\alpha)$. This operation can be applied independently of the SAT solving result, even if the problem is satisfiable. After that, we update each remaining temporary conflict clause α with $\neg\Gamma(\alpha)$ and mark the resulting clause as pervasive. In practice, there is no need to create a new resolution derivation $T2P(\pi)$.

Note that one needs to store and maintain the resolution derivation in order to apply our transformation. This may have a negative impact on performance. To mitigate this problem, we store only a subset of the resolution derivation, where each clause's associated set of backward reachable assumptions is non-empty. The idea of holding and maintaining only the relevant parts of the resolution derivation was proposed and proved useful in [78].

Finally, when the number of assumptions is large, our transformation might create pervasive clauses which are too large. To cope with this problem we use a user-given threshold n. Whenever the number of backward reachable assumptions for a clause is higher than n, that clause is not transformed into a pervasive clause, and thus is not reused in subsequent SAT invocations.

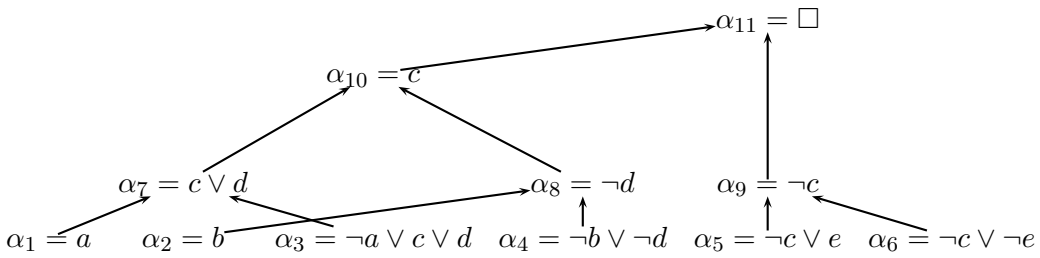

Figure 5.1: An example of a resolution refutation for illustrating the *T2P* transformation. The pervasive input clauses are $F = \alpha_3 \wedge \alpha_4 \wedge \alpha_5 \wedge \alpha_6$; the assumptions are $\alpha_1 = a$ and $\alpha_2 = b$. The only pervasive derived clause is α_9; the rest of the derived clauses are temporary.

5.5 Incremental SAT Solving under Assumptions with Step Look-Ahead

In some applications of incremental SAT solving under assumptions, look-ahead information is available. Specifically, before invocation number i, the solver may already know the clauses F_j and assumptions A_j for some or all future invocations $j > i$. In this section, we propose an algorithm for incremental SAT solving under assumptions given a limited form of look-ahead, which we call step look-ahead. The reason for choosing this form of look-ahead is inspired by step-based approaches to BMC [48].

Given an integer step $s > 1$, an invocation i is *step-relevant* iff i *modulo* $s = 0$ (invocations are numbered starting with 0). Given an invocation q, its *step interval* is a set of successive invocations $SI(q) = [n * s, \ldots, q, \ldots, ((n + 1) * s) - 1]$, where $n * s$ is the largest step-relevant invocation smaller than or equal to q. For example, for $s = 3$, invocations $0, 3, 6, 9, 12, \ldots$ are step-relevant; and $SI(3) = SI(4) = SI(5) = [3, 4, 5]$. In *step look-ahead*, at each step-relevant invocation i, the solver can access all the clauses and assumptions associated with invocations within $SI(i)$. In addition, in step look-ahead, given a step-relevant invocation i, it holds that $F_j \wedge A_j$ is satisfiable iff $F_j \wedge A_j \wedge F_k$ is satisfiable for every $j, k \in SI(i)$. That is to say, we assume that all the clauses available within the step interval hold for every invocation within that step interval.

One can adjust LS to take advantage of the fact that the solver has a

wider view of the problem as follows. At a step-relevant invocation i, LS can be provided the problem $\bigwedge_{j=i}^{i+s-1} F_j$ and solve it s times, each time under a new set of assumptions A_j for each $j \in SI(i)$ (in this scheme non-step-relevant invocations are ignored). We call this approach the *Single instance Literal-based with Step look-ahead (LSS)* approach. LSS was proved to have advantages over the plain LS algorithm (which has a narrower view of the problem) in the context of standard BMC [48]. However, it suffers from the same major drawback as plain LS: preprocessing does not take advantage of assumptions.

We need to refine the semantics of the problem before proposing our solution. Given a step-relevant invocation i, an assumption $l \in A_i$ is *invocation-generic* iff $l \in A_j$ for every $j \in SI(i)$. Any assumption that is not invocation-generic is *invocation-specific*. That is, an assumption is invocation-generic iff it can be assumed for every invocation within the given step interval. In our application of incremental BMC under assumptions, described in Section 5.2, the negation of the property for each bound is invocation-specific, while the unrolled temporal assumptions are invocation-generic.

We propose an algorithm, called *Multiple instances Clause/Literal-based with Step look-ahead (CLMS)* (shown in Alg. 10), that combines LS and CM. The algorithm is applied at each step-relevant invocation. It creates the instance $\bigwedge_{j=i}^{i+s-1} F_j$ once as in LS. The key idea is that invocation-generic assumptions can be provided as unit clauses, since assuming them does not change the satisfiability status of the problem for any invocation within the current step interval. To ensure the soundness of solving subsequent step intervals, the unit clauses corresponding to invocation-generic assumptions must be temporary as in CM. After creating the instance the solver is invoked s times for each invocation in the step interval, each time under the corresponding invocation-specific assumptions. To combine SatELite with Alg. 10 in a sound manner, all the invocation-specific assumptions must be frozen. Finally, note that our *T2P* transformation is applicable for CLMS.

Algorithm 10 CLMS Algorithm

1: **if** i is step-relevant **then**
2: Let $G = \bigcap_{j=i}^{i+s-1} A_j$ be the set of all invocation-generic assumptions
3: Create a SAT solver instance with pervasive clauses $\bigwedge_{j=i}^{i+s-1} F_j$ and temporary clauses G
4: Optionally, apply SatELite, where all the invocation-specific assumptions in $\bigcup_{j=i}^{i+s-1} A_j$ must be frozen
5: **for** $j \in \{i, i+1, \ldots, i+s-1\}$ **do**
6: Invoke the SAT solver under the assumptions $A_j \setminus G$
7: Optionally, transform temporary clauses to pervasive clauses using *T2P*
8: Store the pervasive clauses and delete the SAT instance

5.6 Experimental Results

This section analyzes the performance of various algorithms for incremental SAT solving under assumptions on instances generated by incremental BMC under assumptions. In our analysis, we consider an instance satisfiable iff a certain invocation over that instance by one of the algorithms under consideration was satisfiable within a time-out of one hour. We picked instances from three satisfiable families comprising satitisfiable instances only (128 instances) and four unsatisfiable families comprising unsatisfiable instances only (81 instances). We measured the number of completed incremental invocations for unsatisfiable families and the solving time until the first time an invocation was proved to be satisfiable for satisfiable families (the time-out was used as the solving time when an algorithm could not prove the satisfiability of a satisfiable instance). Each pair of invocations corresponds to a BMC bound (a clock transition and a real bound), where the complexity of SAT invocations in BMC grows exponentially with the bound. We implemented the algorithms in Intel's internal state-of-the-art Eureka SAT solver and used machines with Intel® Xeon® processors with 3Ghz CPU frequency and 32Gb of memory for the experiments.

We checked the performance of LS and CM as well as of LSS and CLMS with steps 10 and 50. We tested CM and CLMS with and without SatELite and with different thresholds for applying *T2P* transformation (0, 100, 100000).

Algorithms				Completed Invocations				
LS?	SatELite?	Step	T2P Thr.	Overall	Fam. 1	Fam. 2	Fam. 3	Fam. 4
-	+	50	0	2967	1443	470	562	492
-	+	10	100	2934	1413	472	563	486
-	+	10	0	2932	1408	474	568	482
-	+	50	100	2927	1427	462	552	486
-	+	50	100000	2927	1427	462	552	486
-	+	1	0	2828	1365	468	539	456
-	+	1	100	2813	1363	462	535	453
-	-	10	100000	2806	1378	442	528	458
-	-	50	0	2801	1375	444	526	456
-	-	50	100	2795	1373	442	522	458
-	-	50	100000	2795	1373	442	522	458
-	-	10	100	2779	1357	440	528	454
-	-	10	0	2775	1353	438	530	454
-	-	1	100000	2736	1335	432	537	432
-	-	1	100	2734	1339	436	526	433
-	-	1	0	2732	1339	436	524	433
+	-	10	N/A	2579	1295	380	494	410
+	-	1	N/A	2575	1295	378	494	408
+	-	50	N/A	2563	1291	376	488	408
-	+	10	100000	2525	1245	390	507	383
-	+	1	100000	2250	1133	296	493	328

Table 5.1: The number of invocations completed within an hour for the unsatisfiable instances from four families. The algorithms are sorted by the sum of completed invocations in decreasing order.

Our solver uses database minimization during inprocessing by default.

The graph on the left-hand side of Figure 5.2 provides information about the number of variables and assumptions (satisfiable and unsatisfiable instances appear separately). For each instance we measured these numbers at the last invocation completed by both CM and LS (the basic algorithms). Note that the distribution of variables and assumptions for the satisfiable instances is more diverse. This is explained by the fact that for satisfiable instances, the last invocation is sometimes very low or very high, while for unsatisfiable instances it is moderate. Overall, our satisfiable instances are easier to solve.

Consider Table 5.1, which compares the number of completed invocations for unsatisfiable instances. Compare basic CM and LS (configurations [-,-,1,0] and [+,-,1], respectively). CM significantly outperforms LS. As we discussed in Section 5.3, the reasons for this are related to the relative efficiency of database simplification and the average clause length for both algorithms. Figure 5.3 demonstrates the huge difference between the two

algorithms in these parameters in favor of CM. Note that when SatELite is not applied, the best performance is achieved by CLMS_10 (CLMS with step 10) with *T2P*_100000 (*T2P* with threshold 100000). Hence, without SatELite, both CLMS and *T2P* are helpful. SatELite increases the number of completed invocations considerably, while the absolutely best result is achieved by combining SatELite with CLMS_50 when *T2P* is turned off. Figure 5.4 demonstrates that the reason for the inefficiency of the combination of *T2P* and SatELite is related to the fact that the time spent in preprocessing increases drastically when *T2P* is applied with threshold 100000. The degradation still exists, but is not that critical when the threshold is 100.

Consider now Table 5.2, which compares the run-time for satisfiable instances. Note that, unlike in the case of unsatisfiable instances, the default LS is one of the best algorithms. The advantage of LS over CM-based algorithms is that it maintains all the information relevant to the decision heuristic. This advantage proves to be very important in the context of relatively easy falsifiable instances. Still, the absolutely best configuration is the combination of CLMS_10 with SatELite and *T2P*_100, which uses all the algorithms proposed in this paper. The graph on the right-hand side of Figure 5.2 shows that the advantage of our approach over LS becomes apparent as the run-time increases, while LS is still preferable for easier instances.

One can also see that the combination of CLMS_10 with SatELite and *T2P*_100 ([-,+,10,100]) is the most robust approach overall: it is the second best for unsatisfiable instances and the absolute best for satisfiable instances.

5.7 Conclusion

This paper introduced efficient algorithms for incremental SAT solving under assumptions. While the currently widely-used approach (which we called LS) models assumptions as first decision variables, we proposed modeling assumptions as unit clauses. The advantage of our approach is that we allow the preprocessor to use assumptions while simplifying the formula. In particular, we demonstrated that the efficient SatELite preprocessor can easily be modified for use in our scheme, while it cannot be used with LS. Furthermore, we proposed an enhancement to our algorithm that transforms

Algorithms				Time			
LS?	SatELite?	Step	T2P Thr.	Overall	Fam. 1	Fam. 2	Fam. 3
-	+	10	100	104845	10843	35083	58919
+	-	1	N/A	118954	18005	41624	59325
-	+	10	0	134917	16886	40965	77067
+	-	10	N/A	139787	21726	53304	64757
-	+	10	100000	154437	22280	53436	78721
-	+	50	0	172104	10496	56087	105521
-	+	50	100	189965	11649	69373	108943
-	+	50	100000	192790	15220	68475	109096
-	-	10	100000	196784	12521	126153	58110
+	-	50	N/A	200261	22832	93635	83794
-	-	10	100	205124	16133	125529	63462
-	-	10	0	206390	14991	125400	65999
-	+	1	100	213278	31628	83009	98641
-	-	1	100	216714	20889	118703	77122
-	-	1	100000	220054	20639	128871	70545
-	+	1	0	219346	34447	89040	95859
-	-	1	0	228404	23642	121608	83154
-	-	50	0	244202	18996	138971	86235
-	+	1	100000	244826	34735	111862	98229
-	-	50	100000	247347	18514	138552	90281
-	-	50	100	250937	18897	141524	90516

Table 5.2: Solving time in seconds for instances from three falsifiable families. The algorithms are sorted by overall solving time in increasing order.

temporary clauses into pervasive clauses as a post-processing step, thus improving learning efficiency. In addition, we developed an algorithm which improves the performance further by taking advantage of a limited form of look-ahead information, which we called step look-ahead, when available. We showed that the combination of our algorithms outperforms LS on instances generated by a prominent industrial application. The empirical gap is especially significant for difficult unsatisfiable instances.

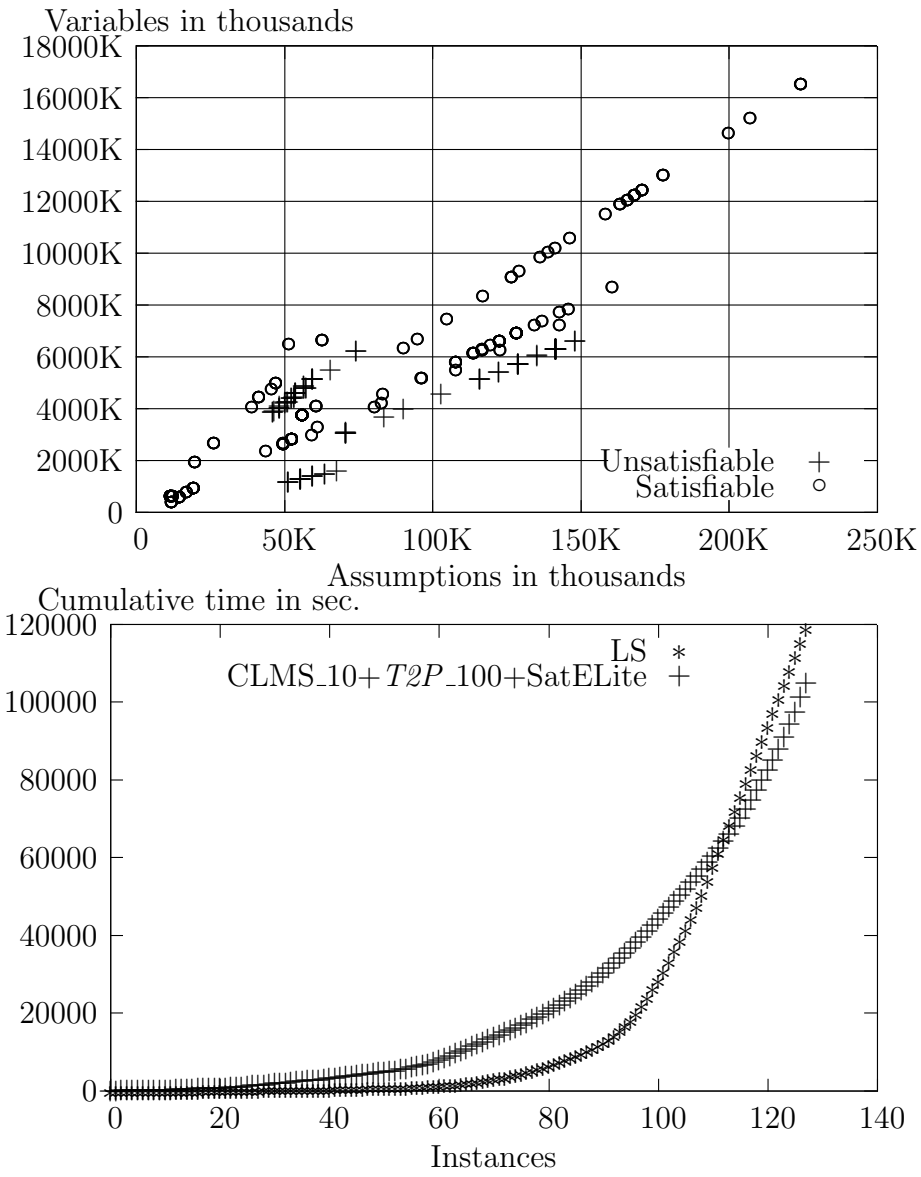

Figure 5.2: Left-hand side: variables to assumptions ratio; Right-hand side: a comparison between plain LS and CLMS_10+*T2P*_100+SatELite with respect to the number of satisfiable instances solved within a given time.

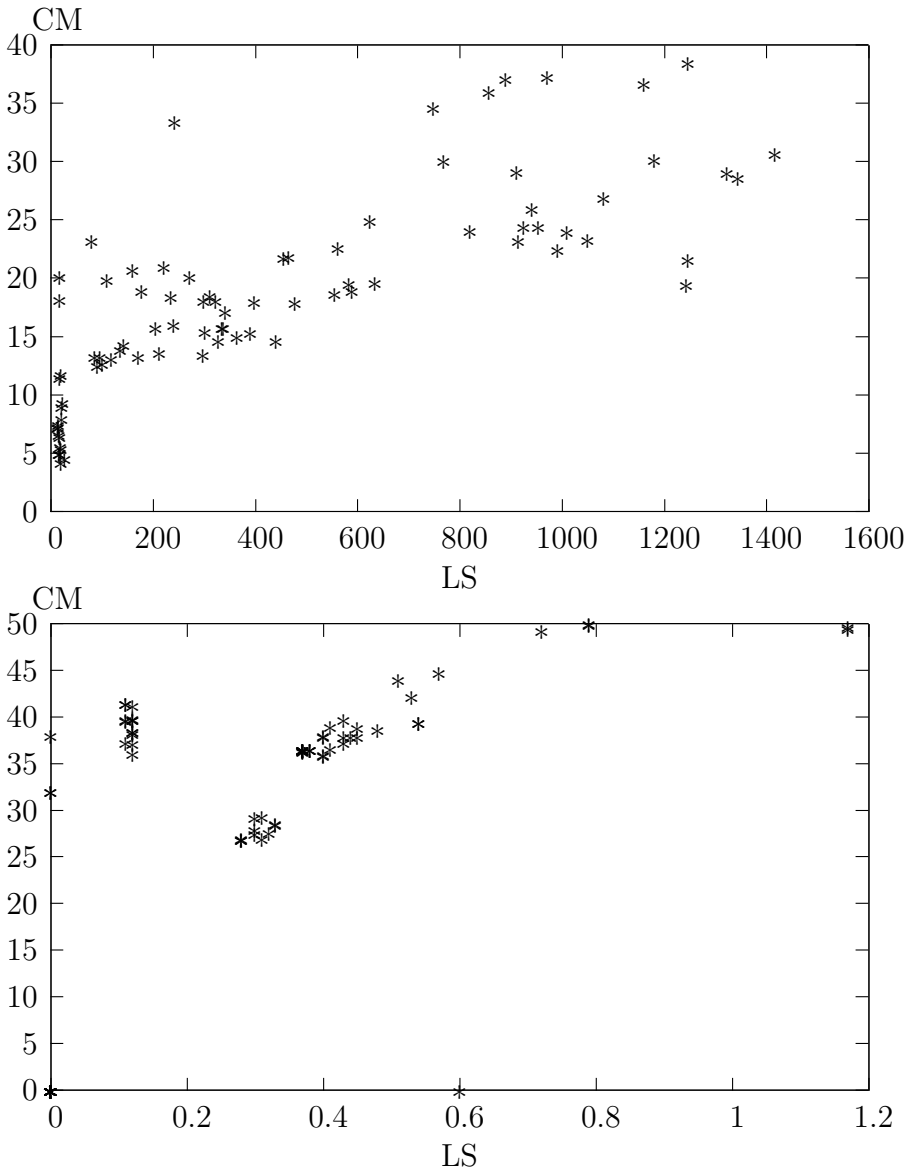

Figure 5.3: Comparison of CM and LS with respect to average conflict cause length (left-hand side) and the percent of clauses removed by database simplification (right-hand side). Note the difference in the scales of the axes.

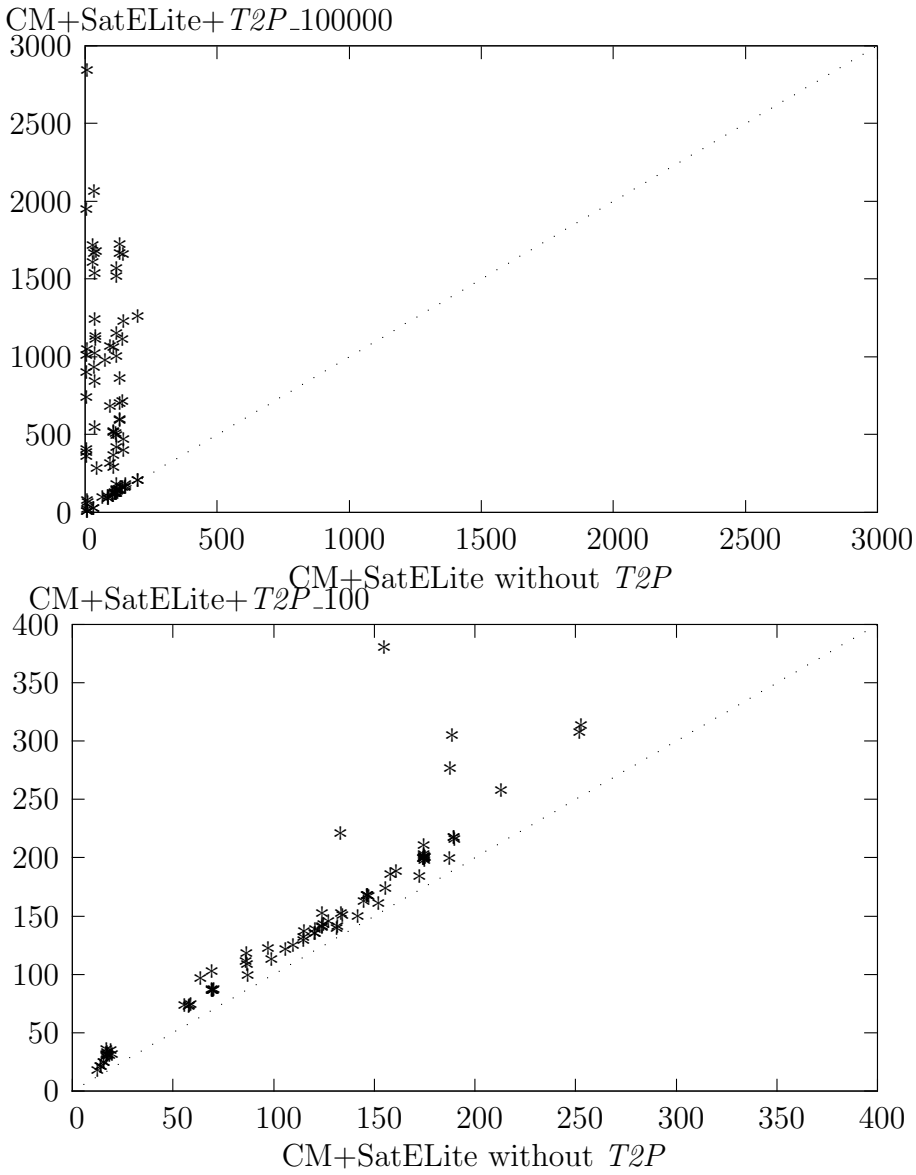

Figure 5.4: Comparison between CM and CM+*T2P*_100000 (left-hand side) and between CM and CM+*T2P*_100 (right-hand side) in terms of time in seconds spent in SatELite.

Chapter 6

Faster Extraction of High-Level Minimal Unsatisfiable Cores

Vadim Ryvchin[1,2] and Ofer Strichman[1]

[1] *Information Systems Engineering, IE, Technion, Haifa, Israel*

[2] *Design Technology Solutions Group, Intel Corporation, Haifa, Israel*

Abstract

Various verification techniques are based on SAT's capability to identify a small, or even minimal, unsatisfiable core in case the formula is unsatisfiable, i.e., a small subset of the clauses that are unsatisfiable regardless of the rest of the formula. In most cases it is not the core itself that is being used, rather it is processed further in order to check which clauses from a pre-known set of *Interesting Constraints* (where each constraint is modeled with a conjunction of clauses) participate in the proof. The problem of minimizing the participation of interesting constraints was recently coined *high-level* minimal unsatisfiable core by Nadel [66]. Two prominent examples of verification techniques that need such small cores are 1) abstraction-refinement model-checking techniques, which use the core in order to identify the state variables that will be used for refinement (smaller number of such variables in the core implies that more state variables can be replaced with free inputs in the abstract model), and 2) assumption minimization, where the goal is to minimize the usage of environment assumptions in the proof, because these assumptions have to be proved separately. We propose seven improvements to the recent solution given in [66], which together result in an overall reduction of 55% in run time and 73% in the size of the resulting core, based on our experiments with hundreds of industrial test cases. The optimized procedure is also better empirically than the assumptions-based minimization technique, and faster by more than an order of magnitude than the best known general MUS solver.

6.1 Introduction

Given an unsatisfiable CNF formula φ, an unsatisfiable core (UC) is any subset of φ that is unsatisfiable. The decision problem corresponding to finding the *minimum* UC is a Σ_2-complete problem [40]. Finding a *minimal* UC (a UC such that the removal of any one of its clauses makes the formula satisfiable) is D^P-complete [70][1]. There are many works in the literature on extracting minimum [40,52], minimal [28,39,53,69], or just small cores [36,98] — see [66] for an extensive survey.

There are many uses to the core in SAT-based verification, typically related to abstraction or decomposition. In most cases, however, it is not the core C itself that is being used, rather C is processed further in order to check which *Interesting Constraints* participate in the proof, where which constraints are interesting is given as input to the problem. Hence we can assume that in addition to the formula we are given as input a set of sets of clauses $IC = \{R_1 \dots R_m\}$, where each R_i is a set of clauses that together encode an interesting constraint. The goal is thus to minimize the number of constraints in IC that have a non-empty intersection with C. This problem was first mentioned in [53] and recently coined *the high-level minimal unsatisfiable core* problem by Nadel [66], who observed that in his experiments with industrial problems the number of clauses that belong to interesting constraints is on average about 5% of the clause database. In fact in the verification group in Intel high-level cores are the only type of cores that are being computed, and we are not aware of any use of the general core in the EDA industry.

Two prominent examples of such techniques that are used in Intel and are described in more detail in the above reference are:

- A popular abstraction-refinement model-checking is based on iterating between a complete model checker and a SAT-based bounded model checker [41, 61]. The model checker takes an abstract model, in which some of the state variables are replaced with inputs, and either proves

[1] D^P is the class containing all languages that can be considered as the difference between two languages in NP, or equivalently, the intersection of a language in NP with a language in co-NP.

the property or returns the depth in which it found a counterexample. In the latter case, this depth is used in a bounded-model checking run over the concrete model, which may either terminate with a concrete counterexample, or with an unsat answer. In the latter case SAT's capability to identify an unsatisfiable core is used for identifying those state variables that are sufficient for proving that there is no counterexample at that depth. All the clauses that contain a given state variable (in any time-frame) constitute a constraint in IC. Those state variables that participate in the proof define the next abstract model (these are the state variables that are *not* replaced by inputs), which is a refinement of the previous one. The process then reiterates until either the model checker is able to prove the property or the SAT solver finds a concrete counterexample.

- In formal equivalence verification (see, e.g., [47]), two similar circuits are verified to be functionally equivalent. This is done by decomposing the two circuits to 'slices' which are pair-wise verified for equivalence. The equivalence of each such pair is verified against various assumptions on the environment. In other words, rather than integrating a model of the environment with the equivalence verification condition, various properties of the environment are assumed, and added as constraints on the inputs of that condition. Then, if the equivalence is proven, it is still necessary to verify that the assumptions are indeed maintained by the environment. Each assumption is modeled with a set of clauses. The unsatisfiable core obtained when checking the equivalence is analyzed in order to find those assumptions that were used in the proof. Hence, here each constraint in IC is a set of clauses that encode an environment assumption. Here too the verification process attempts to minimize the high-level core in order to minimize the number of environment assumptions that should be verified.

We will address the question of how to minimize the core in the next section. A problem which is mostly orthogonal to minimization is how to make the SAT solver emit a core once it determines that a formula is unsatisfiable. There are two well-known approaches to solve this problem:

- **Resolution-based.** The first approach is based on the ability of many modern SAT solvers to produce a resolution proof in case the formula is unsatisfiable. The solver traverses the proof backwards from the empty clause, and reports the clauses at the leaves as the core [37, 98]. This core is then intersected with the sets of clauses in IC in order to find a high-level core.

- **Assumptions-based.** A second approach is based on the *assumptions* technique, which was first implemented in an early version of Minisat [32]. Assumptions are literals that are assigned TRUE (as decisions) before any other decision. If constraint propagation leads to flipping the assignment of one of the assumptions to FALSE, it means that with these assumptions the formula is unsatisfiable. Minisat is capable of identifying which assumptions led to this conflict, which is exactly what is needed for extracting a high-level core. This can be done with *clause selectors* as follows: Let R_i be constraint in IC and let $\{c_1, \ldots, c_n\}$ be the clauses that encode it. To each clause in this set we add the literal $\neg l_i$, where l_i is a new variable. Then we add l_i to the set of assumptions. Hence setting l_i to TRUE activates this constraint, and setting it to FALSE deactivates it.

  The process of extracting the set of assumptions that led to a conflict is computationally easy. Let C be the clause that forces an assumption to its opposite value. Minisat resolves C with all its predecessors in the implication graph until a clause is generated which contains only negation of assumption literals. The negation of this clause is a conjunction of the assumptions that led to the conflict, also known as the *relevant assumptions*. The relevant assumptions constitute a high-level core.

The assumptions technique generates larger conflict clauses owing to the new selector variables, which may become significant if there are many assumptions [2, 66]. The alternative of activating and deactivating constraints with unit clauses is more economic, as it simplifies and removes clauses. On the other hand, the assumptions technique does not consume memory for saving the proof, nor does it consume time to extract the core. Another difference between these two approaches, which turns out to be very important

in our context, is related to clause minimization [7, 87], which is a technique for shrinking conflict clauses. Whereas in resolution-based core extraction minimization of a clause may pull into the proof additional constraints, this does not happen in the assumptions-based approach. We will describe this issue in more detail in Section 6.4. The experiments in [66] showed that the assumptions-based method is on average faster than the resolution-based method, and produces slightly smaller cores. In the experiments we conducted (on a larger set of benchmarks) we witnessed similar results.

In this article we study seven improvements to the resolution-based high-level MUC problem. With these techniques, which we implemented on top of MiniSat-2.2 and ran over hundreds of industrial examples from Intel, we are able to show a 55% reduction in run time comparing to the techniques in [66], and a 28% improvement comparing to the assumptions-based technique. The configuration that achieves these improvements also reduces the core by 73% and 57%, respectively. More details on our experiments can be found in Section 6.4.

Since we take [66] as the starting point of our optimizations, we begin in the next section by describing it in some detail.

6.2 Resolution-based high-level core minimization

The improvements we consider are relevant to resolution-based core extraction. We implemented inside Minisat 2.2 a rather standard mechanism for maintaining the resolution DAG. The resolution information is kept in a separate database, which we will call here the *resolution table*. This table maintains the indices of the parents and children of each derived clause. On top of this we implemented the reference counter technique of Shacham et al. [80]. In this technique every conflict clause has a counter, which is increased every time it resolves a new clause, and decreased when a child clause is erased. Once the counter of a clause is 0, it does not need to be maintained any longer for the purpose of later retrieving the resolution DAG. In the experiments that were reported in [80], this optimization led to a reduction by

a factor of 3 to 6 in the size of the resolution table.

The unsatisfiable core is retrieved as usual by backward traversal from the empty clause to the roots. But since we are interested in minimizing the core, the story does not end here. We implemented the high-level core minimization algorithm of [66], which appears in Pseudo-code in Alg. 11. The input to this algorithm is a set of interesting constraints $IC = \{R_1 \dots R_m\}$, each of which is a set (or a conjunction, depending on the context) of clauses, and a formula Ω, which is called the *remainder*. The formula $\Psi = \bigwedge_{j=1}^{m} R_j \wedge \Omega$ is assumed to be unsatisfiable, and the proof is available at the beginning of the algorithm. We denote the initial core by *initial_core*. The output of the algorithm is a high-level minimal unsatisfiable core with respect to IC and Ω, i.e., a subset $IC' \subseteq IC$ such that $\Psi' = \bigwedge_{R_j \in IC'} R_j \wedge \Omega$ is unsatisfiable, and no constraint can be removed of IC' without making Ψ' satisfiable.

The algorithm is rather self-explanatory, so we will be brief in describing it. In line 1 any constraint R_i that none of its clauses participated in the proof is removed together with its cone, i.e., all the clauses that were derived (transitively) from R_i clauses. The next line defines the set of candidate indices for the core, which is initiated to the indices of the constraints in IC that were not removed in the previous step. From here on the algorithm attempts to remove elements of this set. In each iteration of the loop, it removes a constraint R_k together with its cone and checks for satisfiability. If the formula is satisfiable, then R_k with its cone is returned to the formula, and R_k is added to the solution set *muc*. Otherwise, the unsatisfiability proof is checked in order to remove any constraint R_i, together with its cone, that did not participate in the proof.

It is interesting to note that this algorithm is tailored for *high-level* core minimization, and not for general core minimization. The difference is evident by observing that the whole set of clauses associated with a constraint R_i is removed, together with their joint core. Had the object of minimization been the whole core, we would rather remove all clauses that did not participate in the proof, even if other clauses that share the same constraint *do* participate in the proof. For example, if $R_i = \{c_1, c_2\}$, and only c_1 participate in the proof, Alg. 11 retains both c_1 and c_2, because removing c_2 does not reduce the size of the high-level core, whereas it may assist in consec-

Algorithm 11 Resolution-based high-level MUC extraction (Based on Alg. 2 in [66])

Input: Unsatisfiable formula of the form $\Psi = \bigwedge_{R_j \in IC} R_j \wedge \Omega$.
Output: A high-level MUC with respect to IC and Ω.

Remove any R_i together with its cone if it is not reachable from the empty clause;
$muc\_cands := \{R_i \mid R_i \cap initial\_core \neq \emptyset\}$; ▷ MUC Candidates
$muc := \{\}$;
while muc_cands is non-empty **do**
    $R_k :=$ a member of muc_cands;
    Check satisfiability of the formula without R_k and its cone;
    if satisfiable **then**
        return R_k and its cone to the formula;
        $muc := muc \cup \{R_k\}$;
    else
        for $R_i \in muc\_cands$ **do**
            if $R_i \cap core = \emptyset$ **then** ▷ $core$ is the unsat core of the proof
                Remove R_i and its cone;
                $muc\_cands := muc\_cands \setminus \{R_i\}$;
return muc;

utive iterations. Furthermore, retaining c_2 is needed in order to guarantee minimality. Without it we may miss the fact that some other constraint can be removed.

6.3 Optimizations

In this section we describe seven low-level optimizations to the basic algorithm that was presented in the previous section. We will use the following terminology: a clause is an *IC-clause* if it either belongs to one of the initial constraints in IC or is a descendant of such a clause in the resolution DAG. Other clauses are called *remainder* clauses. We say that a literal is *IC-implied* if it is implied by an IC-clause or just *implied* otherwise.

A: Maintaining partial resolution proofs.

In this optimization we maintain only clauses in the cone of IC-clauses in the resolution table, and the links between them. That is, we save an IC-clause, and the parents and children that are also IC-clauses. Comparing to full resolution, this reduces the amount of memory required by more than an order of magnitude in most cases, reduces the amount of time that it takes to find clauses that are in the cone of an IC (recall that in line 13 of Alg. 11 IC-clauses are removed together with their cones), and, more importantly, allows to activate a certain simplification (see below) for remainder clauses, which is normally turned off when running Alg. 11.

The simplification in point is applied in decision level 0, owing to constants. If the clause database includes a unit clause, e.g., (x), then many solvers would remove those clauses that contain x, and remove $\neg x$ from all other clauses, at decision level 0 (MiniSat is a little different in this respect: it does not remove $\neg x$ from existing clauses once x is learned, but rather it does not add $\neg x$ to new learned clauses). This simple, yet powerful simplification has to be turned off when running Alg. 11. For example, if (x) is an IC-clause associated with constraint R_1, then we cannot just remove clauses with x from the formula, since we might decide at line 13 to remove R_1, which will force us to retrieve these clauses. Empirically it is better to

retain such clauses rather than keeping them in a file and then retrieving them. The same issue occurs when removing the negation of x from clauses: here too, we will need to retrieve the original clauses once R_1 is removed. One of the advantages of this optimization, therefore, is that we can turn back on this simplification for the remainder clauses.

B: Selective clause minimization.

Clause minimization [7,87] is a technique for shrinking conflict clauses. Once a clause is learnt, each of its literals is tested: if it implies other literals in the clause, it can be removed.

Example 6.3.1 *Consider the following clauses:*

$$\begin{array}{lcllcl} C_1 & = & (\neg v_1 \vee v_2) & C_2 & = & (\neg v_2 \vee v_3) \\ C_3 & = & (\neg v_4 \vee v_5) & C_4 & = & (\neg v_5 \vee v_6) \\ C_5 & = & (\neg v_1 \vee \neg v_3 \vee \neg v_4 \vee \neg v_6) & & & \end{array}$$

Suppose that the first decision is v_1. This decision implies v_2 (from C_1) and v_3 (from C_2). Suppose now that the next decision is v_4. This decision implies v_5 (from C_3) and v_6 (from C_4) and a conflict in clause C_5. Conflict analysis based on 1-UIP returns in this case a new clause $C = (\neg v_1 \vee \neg v_3 \vee \neg v_4)$. From C_1 and C_2 we can see that $v_1 \rightarrow v_3$, or equivalently $\neg v_3 \rightarrow \neg v_1$, which is an implication between literals in C. Clause minimization will find this implication by following the resolution DAG and remove $\neg v_3$.

We will not present the full algorithm for clause minimization here, but rather only mention that it is based on traversing the resolution DAG backward from each literal l in the learned clause. The hope is to hit a 'frontier' of other literals from the same clause that by themselves imply l. If in this process we hit a decision variable, it means that l cannot be removed.

Example 6.3.2 *Continuing the previous example, the algorithm scans each non-decision literal in C. Consider v_3: this literal was implied in C_2, and hence we progress to look at the other literal in that clause, namely v_2. This literal was implied by C_1 and hence we look at v_1. But since $v_1 \in C$, it*

means that we found an implication within C, and hence $\neg v_3$ can be removed. Note that the minimized clause can be resolved from the original one and the clauses that are traversed in the process. In this case $Res(C, Res(C_1, C_2)) = (\neg v_1 \vee \neg v_4)$.

The problem with clause minimization in our context is that it may turn a non-IC-clause C into a shorter IC-clause C'. This can happen if the minimization process uses an IC-clause: in that case C' has to be marked as an IC-clause as well. Furthermore, it can turn an IC-clause C that depends on a certain set of interesting constraints, into a shorter IC-clause that depends on *more* such constraints. This means that if that clause will participate in the proof, it will 'pull-in' more constraints into the core.

Our suggested optimization is to cancel clause minimization in any case that an IC-clause is involved. In other words, we prefer a large clause that depends on a few constraints, over a smaller one with more such dependencies. The latter may pull more constraints into the proof, and lead to other such clauses. We aspire, instead, to keep the resolution table as small as possible and with the fewest connections to IC-constraints. Ideally we should check whether using a certain IC-clause in the minimization process indeed adds dependencies, but this is simply too expensive: for this we would need to traverse the DAG backwards all the way to the roots in order to check which constraints are involved.

It is interesting to analyze the behavior of the assumptions-based method with respect to clause minimization. It turns out that it solves this problem for free, and hence in this respect it is a superior method. In fact from analyzing various cases in which it performs much better than the clause-based method (before the optimizations suggested here were added), we realized that this is the main cause for the difference in run-time, rather than the facts mentioned in the introduction (the fact that it does not need to save the resolution table, nor to extract the core in the end of each iteration). How does it solve this problem for free? Observe that with this technique all IC-clauses have as literals all the selector variables that correspond to constraints that were used in deriving that clause. For example, let R_1, R_2 be two constraints with associated selector variables l_1, l_2 respectively. If R_1 and R_2 participate in inferring C, then C must contain $\neg l_1$ and $\neg l_2$. This is

implied by the fact that selector variables appear only in one phase in the formula, and hence cannot be resolved away. Hence the presence of these literals in IC-clauses is an invariant. If we falsely assume that a minimized clause C can increase its dependency on constraints, we immediately reach a contradiction: the supposedly added constraint implies that a new selector variable was added to C, which contradicts the fact that literals are only removed from C in the minimization process.

C: Postponed propagation over IC-clauses.

In this optimization we control the BCP order. We first run BCP over non-IC-clauses until completion. If there is no conflict, we propagate a single implication due to an IC-clause, and run regular BCP again. We repeat this process until no more propagations are possible or reaching a conflict. The idea behind this optimization is to increase the chances of learning a remainder clause rather than an IC-clause.

D: Reclassifying IC-clauses.

When we discover that some IC-constraint R must be in the MUC (line 8 in Alg. 11), we add its clauses back as remainder clauses, together with all the clauses in its cone that do not depend on other constraints. To identify this set of constraints, we employ an algorithm in the style of a least-fix-point computation. We insert all the R clauses into a set S. Then we add all the children of those clauses that all their parents are in S. We repeat this process until reaching a fix-point.

Without this optimization R's clauses are added back as is, with their marking as IC-clauses. By adding them back as remainder clauses, we enable more simplifications, as described in the case of optimization **A**.

E: Selective learning of IC-clauses.

When detecting a conflict, the learned clause may be an IC-clause. If all else is equal, such a clause is less preferable than a remainder clause, as it may increase the high-level core, in addition to the fact that it leads to a larger

resolution table and hence longer run times. We found that learning a non-asserting remainder clause instead, combined with partial restart, improves the overall performance. The learning of the remainder clause is essential for termination, and also turns out to decrease run time. The alternative remainder clause that we learn is even closer to the conflict than the first UIP. We can learn it only if the conflicting clause is not an IC-clause; in other cases we simply revert to learning the IC-clause. Learning the remainder clause is done by reanalyzing the conflict graph *as if the IC-implications were decisions*. This optimization is only ran in conjunction with optimizations **B** and **C** above, for reasons that we will soon clarify. Alg. 12 describes the procedure for learning this clause.

Algorithm 12 An algorithm that attempts to find a remainder conflict clause by reanalyzing the conflict graph as if the IC-implications were decisions. Returns a remainder clause if one can be found, and NULL otherwise.

---

function Get_Remainder_Clause

1. If the conflicting clause is an IC-clause then return NULL.
2. Search an IC-implied literal l in the trail, starting from the latest implied literal and ending just before the 1-UIP literal.
3. Convert the implication of l into a decision, and update accordingly the decision level of all implied literals in the trail that come after it.
4. Call Analyze_Conflict() with the same conflicting clause, but while referring to the new decision levels. Let C be the resulting conflict clause.
5. Return C.

Note that the fact that we use this algorithm only when optimization **C** is active, guarantees that the literals searched and updated in steps 2 and 3 are implied by l, i.e., the fact that BCP was ran to completion on non-IC-clauses before asserting l, guarantees that the rest of the implications at that decision level depend on asserting l. Also note that the clause learnt in step 4 is necessarily a remainder clause because Analyze_Conflict() cannot cross

an IC-implied literal (such implications were made into decisions), and that it corresponds to a cut in the implication graph to the right of the first UIP. The reason we activate this optimization in conjunction with optimization **B**, is that we want to refrain from a case in which we learn a remainder clause, but it then turns into an IC-clause owing to clause minimization. This is not essential for correctness, however: we could also have just compared this smaller IC-clause to the original one and choose between the two, but our experience is that it is better to give priority to minimizing the number of IC-clauses. Finally, note that there is no reason to revert the changes made to the trail, because backtracking removes this part of the trail anyway.

Example 6.3.3 *Figure 6.1 presents an implication graph, where IC-implications are marked with dashed edges. The marked 1-UIP cut in the top drawing is calculated while considering such implications as any other implication. The suggested heuristic is to learn instead a normal clause, by considering such implications as new decisions, as depicted in the bottom drawing.*

As mentioned earlier, learning the alternative clause is combined with a partial restart. Let dl be the level to which we would have jumped had we learned the IC-clause. We backtrack to dl, but at this point nothing is asserted because we did not learn an asserting clause. We then move to the next decision level, $dl + 1$, and decide the negation of the original 1-UIP literal. Hence instead of learning an asserting clause and implying the negation of the 1-UIP literal, we refrain from learning that clause and decide on the same value. This assignment in neither necessary or sufficient for preventing the same conflict to occur. What prevents us from entering an infinite loop in the absence of standard learning is the fact that we learn at least one clause between such partial restarts. Since the solver cannot enter a conflict state that leads to learning an existing clause, we are guaranteed not to enter an infinite loop.

Example 6.3.4 *Referring again to the conflict graphs in Example 6.3.3, our solver backtracks to the end of level 3 — the same level we would have jumped with the original IC-clause — progress to level 4 and decides $\neg l_1$.*

In our experiments we also tried other decisions (such as $\neg l_2$ in the example above), but $\neg l_1$ seems to work better in practice. We also tried different strategies of updating the scores. The best strategy we found in our experiments is to update the score according to both the original and the alternative clause.

F: Selective Chronological backtracking.

Recall that optimization **E** involves a partial restart when learning an IC-clause. Different heuristics can be applied in order to choose the backtracking level. Our experiments show that if we only backtrack one level, rather than to the original backtrack level as explained above, the results improve significantly. The complete set of data, available from [75], shows that this heuristic improves the run time in most instances, and that it improves the search itself and not only reduces constants, as is evident by the fact that it reduces the number of conflicts. It seems that the reason for the success of this heuristic is related to the fact that with normal backtracking and score scheme we may lose the connection to the clause that we actually learn, i.e., the scores might divert the search from a space which is more relevant to the alternative clause that we learn.

G: A removal strategy.

Recall that in line 5 of Alg. 11 constraints are removed in an arbitrary order. We suggest a simple greedy heuristic instead: remove the constraint that contributed the largest number of clauses to the proof. This heuristic, as will be evident in the next section, reduces the size of the resulting core but slightly increases run time.

We also experimented with a heuristic by which we remove the constraint with the *least* number of clauses in the proof, speculating that this leaves more clauses in the formula and hence increases the chance that there will be a proof without this constraint. This option also improves performance comparing to the arbitrary order with which we started, but is not as good as the one suggested above. There is an indirect cause behind this difference: the large constraints (i.e., those that have many clauses) are typically neces-

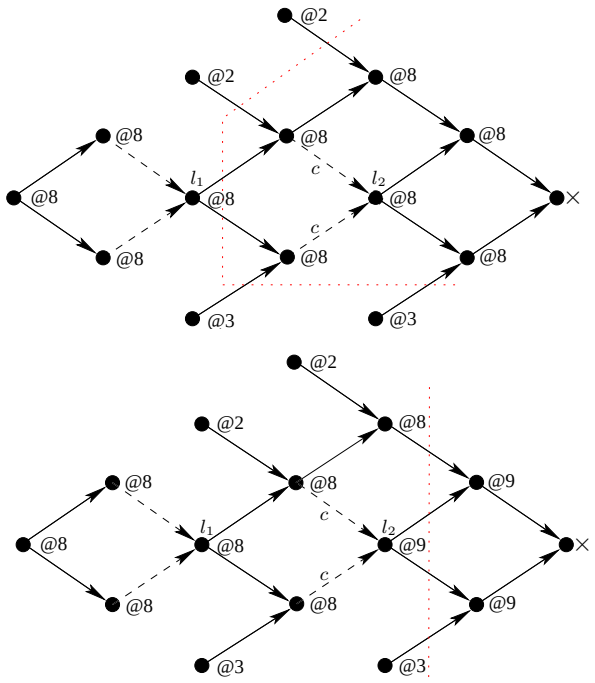

Figure 6.1: In these conflict graphs, dashed arrows denote IC-implications, and the dotted lines denote 1-UIP cuts. In the top drawing, where such implications are referred to as any other implications, the learned 1-UIP clause must be marked as an IC-clause, since it is resolved from the IC-clause c. We can learn instead a normal clause by taking, for example, the 1-UIP clause in the bottom conflict graph. In that graph, c's implication are considered as decisions, which changes the decision levels labeling the nodes.

sary for the proof regardless of the other constraints, and hence the faster we make them remainder constraints – with optimization **D** – the faster the rest of the solution process is. This, in turn, affects the size of the core because it leads to less time-outs. As we will explain in the next section, the result of the algorithm when interrupted by a time-out is the last computed core, or, in case that even the first iteration does not terminate, the entire set of IC-clauses.

6.4 Experimental results

As was mentioned earlier, as a starting point we implemented the algorithm from Section 6.2 in Minisat 2.2, and reduced the amount of required data in the resolution table by using the reference-counter technique of [80]. On top of this we implemented the optimizations that were described in the previous section, and ran all possible combinations (excluding the restrictions mentioned in optimization **E**), on the set used in [66] (family 'lat-fmcad10' in the tables below), and additional nine families of harder abstraction-refinement benchmarks from Intel. We removed from the benchmark set instances that could not be solved by any of the configurations in the given time-out of one hour. This left us with 144 benchmarks, all of which are from the two application domains that were described in the introduction. This set constitute Intel's contribution to the benchmarks repository that will be used in the upcoming SAT competition dedicated to this problem. The average number of clauses per instance is 2,572,270; the average number of constraints per instance is 3804; and, finally, the average number of interesting clauses per instance is 96568 (25.3 clauses per constraint), which is approximately 6% of the clauses. All experiments were ran on Intel® Xeon® machines with 4Ghz CPU frequency and 32Gb of memory.

Table 6.1 shows run time results for selected configurations.[2] The second column ("Full") refers to our starting point as explained above. One may observe that the best result is achieved when combining the first six optimizations, whereas the seventh slightly increases the overall run-time.

[2]The full set of results can be downloaded from [75].

We also compared our results to assumptions-based minimization. We tried both a simple scheme, and the improvement suggested in [66]. In the simple scheme, a constraint is added to the MUC (line 8 in Alg.11) by setting its associated selector variable to true; In the improved method the same effect is achieved by adding a unit clause asserting this literal to TRUE. Similarly, in the simple scheme an environment assumption is removed from the formula (line 13 in Alg.11) by setting its associated selector to FALSE; In the improved method the same effect is achieved by adding a unit clause asserting this literal to FALSE. The improved method is better empirically apparently because the unit clause invokes a simplification step in decision level 0, which removes the selector variable and erases some clauses. The results we witnessed with the two methods appear in the last two columns of the table. Overall the combination of optimizations achieve a reduction of 55% in run time comparing to our starting point, and a reduction of 28% comparing to the assumptions-based method.

All the presented methods can be affected by the order in which constraints are removed in line 5. We therefore tried three different arbitrary removal orders in each case. Empirically this hardly had an effect on the average run-time when using the resolution-based methods, whereas it had some effect when using the assumption-based methods. The table below represents the best overall run times among the different orders we tried (i.e., we present the results that together have the minimum run-time). Regarding the size of the resulting core, the different arbitrary orders had inconsistent effect, as expected, but the order referred to in optimization **G** had a non-negligible positive effect on the size of the core, as will be shown momentarily.

Next, we consider the size of the resulting high-level MUC. The configuration that achieves the best run-time (A–F) achieves the second smallest high-level core, whereas the second best configuration in terms of run time (A–G) achieves the smallest core. If a solver timed-out in our experiments, we considered its latest computed core, i.e., the set $muc \cup muc\_cands$. If a solver did not finish even the first iteration, then we considered the entire set of clauses in IC as its achieved core. This policy, which reflects the way such cores are used, explains the different results of strategies that are supposed to be equivalent with respect to the size of the core. For example, the

Benchmark	Resolution-based								Assumption-based	
family	Full	A	AB	ABC	ABCE	A–E	A–F	A–G		units
latch1	2001	1604	660	465	570	575	425	423	819	798
gate1	3747	1403	705	636	620	579	490	477	856	855
latch2	9113	5915	6636	6116	5685	5656	2424	2370	8153	8043
latch3	348	293	274	274	283	275	262	200	236	236
latch4	769	529	506	457	467	455	443	379	504	521
latch5	1103	820	735	657	678	630	632	625	747	689
lat-fmcad10	785	457	445	451	435	435	400	394	417	425
latch6	8868	5456	5329	5188	5007	5006	4948	4943	5322	5279
latch7	9956	7050	5719	5244	5094	5096	5302	5286	5688	5652
latch8	8223	7946	5673	6133	5459	5420	5127	5587	8004	5534
Total	44913	31473	26682	25621	24298	24127	**20453**	20684	30746	28032

Table 6.1: Summary of run-time results by family (144 instances all together).

Benchmark	Resolution-based								Assumption-based	
family	Full	A	AB	ABC	ABCE	A–E	A-F	A-G		units
latch1	41	41	41	41	42	42	41	42	52	45
gate1	1143	1210	1089	568	1029	1029	870	901	618	1192
latch2	5887	2851	127	3040	2851	2851	131	129	3782	4165
latch3	168	202	202	199	211	211	208	123	140	132
latch4	236	237	248	236	238	238	237	162	177	217
latch5	224	266	266	206	206	206	220	222	222	223
lat-fmcad10	577	456	456	489	540	540	453	454	457	450
latch6	2550	2502	2502	2490	2490	2490	2480	2480	2463	2502
latch7	2578	322	585	253	154	154	211	204	304	287
latch8	5591	615	2867	393	344	344	371	373	2887	2877
TO	8	5	3	3	2	2	2	2	6	5
Total	18995	8702	8383	7915	8105	8105	5222	**5090**	11102	12090

Table 6.2: Summary of the size of the high-level core by family. The 'TO' row indicates the number of time-outs.

partial-resolution proof optimization (**A**) does not remove more clauses than 'Full', but since the latter is generally slower, it times-out more times and hence its core count is larger. The 'TO' row contains the number of such time-outs with each configuration.

A comparison to MUC solvers We also experimented with Minimal Unsatisfiable Core (MUC) solvers, by counting those interesting constraints that are used in the core that they find. Our experiments with five recent solvers (AOMUS, sat4j, zchaff, mucsat, and PicoMUS-936) show that PicoMUS is the best among them. Yet it times-out 70 times and has a total run-time of 296151 seconds with our benchmarks – more than an order of magnitude slower than our solver. However, note that this is a comparison of tools, not of techniques.

6.5 Summary and future work

The recently introduced problem of finding a *high-level* minimal unsatisfiable core has various applications in the industry. Until [66] the standard practice was to minimize the core itself, and only then to find the interesting part of it. Our experiments show that this approach cannot compete with a solver that focuses on the high-level core. In this article we introduced seven techniques that reduce both the run time and the resulting high-level core.

A straight-forward direction for future research is to migrate some of the suggested optimizations to the assumptions-based approach. Related SAT problems may also benefit from these methods. First - it is possible that general SAT solving can be improved with some combination of optimizations **E** and **F**. Second, the same techniques can potentially expedite other methods in which the SAT component needs to extract only partial information from the resolution proof, like interpolation-based model checking [60]. In interpolation only a small part of the proof is necessary in order to generate the interpolant, and we want to explore possibilities to minimize that part and decrease the overall run time with variants of the methods suggested here.

Chapter 7

Efficient MUS Extraction with Resolution

Alexander Nadel[1], Vadim Ryvchin[1,2] and Ofer Strichman[2]
[1] *Intel Corporation, P.O. Box 1659, Haifa 31015 Israel*
[2] *Information Systems Engineering, IE, Technion, Haifa, Israel*

Abstract

We report advances in state-of-the-art algorithms for the problem of Minimal Unsatisfiable Subformula (MUS) extraction. First, we demonstrate how to apply techniques used in the past to speed up resolution-based Group MUS extraction to plain MUS extraction. Second, we show that *model rotation*, presented in the context of *assumption-based* MUS extraction, can also be used with *resolution-based* MUS extraction. Third, we introduce an improvement to rotation, called *eager rotation*. Finally, we propose a new technique for speeding-up resolution-based MUS extraction, called *path strengthening*. We integrated the above techniques into the publicly available resolution-based MUS extractor Haifa-MUC, which, as a result, now outperforms leading MUS extractors.

7.1 Introduction

Given an unsatisfiable formula in Conjunctive Normal Form (CNF), an *Unsatisfiable Subformula* (or *Unsatisfiable Core*; hereafter, *US*) is an unsatisfiable subset of its clauses. A *Minimal Unsatisfiable Subformula (MUS)* is a US such that removal of any of its clauses renders it satisfiable. The problem of finding a MUS is an active area of research [53, 66, 78, 82, 83, 94].

The basic algorithm used in modern MUS extractors such as MUSer2 [11] and Haifa-MUC [78] is as follows. In the initial *approximation stage* the algorithm finds a not-necessarily-minimal US S with one or more invocations of a SAT solver [37, 98]. It then applies the following *deletion-based* iterative process over S's clauses until S becomes a MUS. Each iteration removes a *candidate* clause c from S and invokes a SAT solver. If the resulting formula is satisfiable, c must belong to the MUS, so c is returned to S and marked as *necessary*. Otherwise c is removed from S. In addition, the following two optimizations are commonly applied. First, incremental SAT solving [33, 88] is used across all SAT invocations. Second, when a clause c is found to be not necessary, one can remove from S not only c, but all the clauses (if any) omitted from the new core found by the SAT solver. This latter technique is called *clause set refinement* in [53]. The algorithm we have described up to here was introduced in [28] and improved in [66], while the idea of removing constraints one by one in order to get a minimally infeasible set can be traced back to [6, 23]. See [66] for a more detailed presentation of the algorithm and [82] for an overview of various approaches to MUS extraction.

It was demonstrated in [66] that the approach we have described can be implemented using either a resolution-based or an assumption-based algorithm. The former relies on the resolution proof maintained by the SAT solver for detecting the core at each step, while the latter adds a new *assumption literal* to each clause and detects the core using these assumptions. It was shown in [66] that the resolution-based approach to MUS extraction is faster than the assumption-based approach mainly because of the overhead of maintaining assumption literals.

Various applications require finding a MUS with respect to user-given groups of clauses [53, 66], called *interesting constraints*, while clauses that do

not belong to any interesting constraint are called the *remainder*. The resulting problem is called *Group MUS (GMUS)* extraction (or *high-level MUS* extraction). It was shown in [66] that the approach we described for plain MUS extraction can be applied to GMUS extraction as well. Furthermore, it was shown in [78] that the resolution-based approach to GMUS extraction can be improved considerably by directing the search to ignore the interesting constraints and to use the remainder and the necessary clauses instead whenever possible. We call the techniques of [78] *MUS-biased search*.

The first contribution of this paper is in showing that MUS-biased search can be applied to plain MUS extraction. The key observation is that while there are no *remainder* clauses in plain MUS extraction, *necessary* clauses can still be used for MUS-biased search after the approximation stage.

A recent essential enhancement to the plain MUS extraction algorithm we have described is *model rotation* (or, simply, *rotation*) [10, 53, 83]. Rotation was proposed in the context of assumption-based MUS extraction. After implementing rotation, the resulting assumption-based MUS extractor MUSer2 outperformed the state-of-the-art resolution-based MUS extractor Haifa-MUC. It is sometimes postulated that rotation gives the assumption-based approach an edge over the resolution-based approach (cf. [94]).

The second contribution of this paper is thus in showing that model rotation can be integrated into the resolution-based approach. The paper's third contribution is an improvement to model rotation, called *eager rotation*, detailed in Section 7.2.2.

The fourth contribution of our paper is called *path strengthening*. It is a generalization of a technique proposed in [89] and later called *redundancy removal* in [53] and implemented in MUSer2 [11]. Redundancy removal adds the literals of $\neg c$ (where c is the candidate clause) as assumptions when checking the satisfiability of $S \setminus c$, because since S is known to be unsatisfiable, then $S \setminus c$ and $(S \setminus c) \wedge \neg c$ are equisatisfiable. Path strengthening, on the other hand, adds as assumptions the literals of $\neg c, \neg c_1, \ldots, \neg c_m$ for some $m \geq 0$, where the sequence of clauses $c, c_1, \ldots, c_m$ constitutes the longest common prefix of all paths in the resolution proof from c to the empty clause. Further details about path strengthening are provided in Section 7.2.3.

We integrated our algorithms into the resolution-based MUS extractor

HAIFA-MUC. We show in Section 7.3 that, as a result, HAIFA-MUC now outperforms the leading MUS extractors MUSER2 and MINISATABB [51]. MINISATABB improves MUSER2 considerably based on the idea of replacing blocks of assumptions with new variables [51].

7.2 The Algorithms

7.2.1 MUS-Biased Search

We will now describe how we adapted optimizations **A**-**D** of the GMUS-oriented techniques proposed in [78] to plain MUS extraction (we also tried adapting optimizations **E**-**G** [78], but their impact on plain MUS extraction was negligible). We denote the set of necessary input clauses by M. We call an input clause c *interesting* if it belongs to $S \setminus M$ (i.e., c can still serve as a candidate). A learned clause is marked as *interesting* if it is derived using at least one interesting clause; otherwise it is marked as *necessary*. If an interesting learned clause participates in the proof, then the core includes its interesting roots; this is undesirable since we are trying to minimize the core. Most of our techniques are therefore targeted at biasing the solver towards learning *necessary* rather than *interesting* clauses. This is the reason that we call them, jointly, *MUS-biased search*. An exception is the first optimization below, which is focused on reducing the amount of memory used to store the proof.

A. *Maintain partial resolution proofs.* There is no need to store in the proof any clauses identified as necessary, since the algorithm does not need to work with these clauses explicitly anymore. Hence, we discard from the proof all the clauses that emanate exclusively from M.

B. *Perform selective clause minimization.* Clause minimization [87] is a technique for shrinking conflict clauses. Specifically, if a conflict clause c contains two literals l_1, l_2 such that $l_1 \Longrightarrow l_2$ because of the rest of the formula, then l_2 can be removed from c. The disadvantage of this technique in our context is that it may reclassify c from 'necessary'

to 'interesting', if the implication $l_1 \implies l_2$ depends on an interesting clause. This in turn may increase the size of the core later on as explained above. Hence our optimization does not apply clause minimization if it leads to such a reclassification. In other words we prefer a longer conflict clause if this enables us to maintain its classification as a necessary clause.

C. *Postpone propagation over interesting clauses.* Perform Boolean Constraint Propagation (BCP) on necessary clauses first, with the aim of learning a necessary clause when possible.

D. *Reclassify interesting clauses.* When an interesting clause c becomes necessary, look for any clauses in the resolution derivation that were derived from c that also become necessary (that is, were derived solely from necessary clauses) and reclassify them.

Note that while these optimizations improve GMUS extraction even during the approximation stage owing to the availability of remainder clauses, their impact on plain MUS extraction begins only during the minimization stage, when there are enough necessary clauses (which, like remainder clauses, must be in the proof). Indeed we demonstrate in Section 7.3 that optimization **B** is not cost-effective before there is a significant number of necessary clauses, which is the reason that we invoke it starting from the 2nd satisfiable iteration.

7.2.2 Eager Model Rotation

Model rotation can improve deletion-based MUS extraction by searching for additional clauses that should be marked as necessary *without* an additional SAT call. Suppose, for example, that for an unsatisfiable set S, $S \setminus c$ is satisfiable. Consequently c is marked as necessary. Let h be the satisfying assignment. Note that $h(c) =$FALSE, because otherwise $h(S)$ would be TRUE, which contradicts S's unsatisfiability. Now, suppose that an assignment h' that is different than h in only one literal $l \in c$ satisfies all the clauses in S other than exactly one clause $c' \in S$. Hence $h'(S \setminus c') =$ TRUE, which means that like c, c' must also be in any unsatisfiable subset of S, and can therefore

be marked as necessary as well. Rotation flips the values of each of c's literals one at a time in search of such clauses. When one is found, rotation is called recursively with c'. This algorithm is summarized in Alg. 13. We observe that rotation, proposed in the context of assumption-based MUS extraction, can be integrated into our resolution-based algorithm without any changes.

Alg. 14 shows ERMR (Eager Recursive Model Rotation) – an improvement to rotation that weakens rotation's terminating condition. The reader may benefit from first reading the main algorithm in Alg. 15, which calls ERMR. The only difference between ERMR and RMR is that ERMR may call rotation with a clause that is already in M, the reason being that it can lead to additional marked clauses owing to the fact that the call is with a different assignment. Clearly there is a tradeoff between the time saved by detecting more clauses for M and the time dedicated to the search. For example, one may run RMR with more than one satisfying assignment as a starting point, but this will require additional SAT calls to find extra satisfying assignments. ERMR refrains from additional SAT calls. Rather it changes the stopping criterion: instead of stopping when $c \in M$ (line 4 in Alg. 13), it stops when $c \in K$, where K holds the clauses that were discovered in the *current* call from MUS. There are other variations on weakening the terminating condition of rotation in the literature [53, 94]. We leave to future study a detailed comparison of our algorithm to these works.

7.2.3 Path Strengthening

Path strengthening relies on the following property, which we call *cut falsifiability* (observed already in [28, 65]). Let S be an unsatisfiable formula, π its resolution proof, and c a candidate clause. Let ρ_c be the subgraph of π containing all the clauses that appear on at least one path from c to the empty clause $\square$ (including c and $\square$). Then, any model h to $S \setminus \{c\}$ must falsify at least one clause in any vertex cut of ρ_c (since otherwise a satisfiable vertex cut in π would exist). An immediate corollary is that *all* the clauses in *some* path from c to $\square$ must be falsified by any model h to $S \setminus \{c\}$.

We use this property as follows. Let $P = [c_0 = c, c_1, \ldots, c_m]$ be a path in the resolution proof starting from a candidate clause c. P is the *longest*

unique prefix if it is the longest path starting at c, such that each $c_i \in P$ has only one child (that is, c participates in the derivation of one clause only). *Path strengthening* is based on the following property, induced by cut falsifiability: all the clauses of P must be falsified in any model h to $S \setminus \{c\}$. Alg. 16 shows a variant of the main algorithm in which path strengthening has been applied: each invocation of the SAT solver is carried out under the assumptions $\neg P = \{\neg c_0, \ldots, \neg c_m\}$. Before each iteration our algorithm attempts to increase P length by removing from the resolution proof clauses that are not backward reachable from the empty clause. Note that whenever P contains clauses which do not subsume c, path strengthening will provide more assumptions to the solver than redundancy removal; hence path strengthening is expected to be more efficient than redundancy removal.

Cut falsifiability-based techniques are not immediately compliant with clause set refinement, since clause set refinement requires solving *without assumptions*. MUSer2 solves this problem for redundancy removal by applying clause set refinement only when the assumptions are not used in the proof; otherwise it skips clause set refinement. Our path strengthening algorithm applies clause set refinement when either the assumptions are not used in the proof or whenever the N latest iterations applied path strengthening and the result was unsatisfiable, N being a user-given threshold.

7.3 Experimental Results

We checked the impact of our algorithms when applied to the 295 instances used for the MUS track of the SAT 2011 competition. For the experiments we used machines with 32Gb of memory running Intel® Xeon® processors with 3Ghz CPU frequency. The time-out was set to 1800 sec. The implementation was done in Haifa-MUC. We refer to a configuration of Haifa-MUC that implements the deletion-based algorithm with incremental SAT and clause set refinement as Base. We compare our tool to the latest version of MUSer2 [11] and Minisatabb [51]. Extended experimental data is available from the second author's home page.

Figure 7.1 summarizes the main results. Several observations are in order: 1) rotation is very useful; 2) eager rotation is effective; 3) optimizations

A and **D** are useful, while optimization **B** is beneficial only if delayed until the second satisfiable iteration (2 being the optimal value, based on experiments); 4) path strengthening (with N=20, 20 being the optimal value experimentally) is more beneficial than redundancy removal, and finally 5) Haifa-MUC, enhanced by all our algorithms, is 2.18x faster than MUSer2 and solves 13 more instances, and is 48% faster than Minisatabb and solves 4 more instances. Haifa-MUC is faster than Minisatabb on 196 instances, while Minisatabb is faster than Haifa-MUC on 15 instances. Figure 7.3 shows a cactus plot comparing Base, MUSer2, Minisatabb and the new best configuration of Haifa-MUC, while Figure 7.2 compares Haifa-MUC to Minisatabb.

7.4 Conclusion

We proposed a number of algorithms for speeding up MUS extraction. First, we adapted GMUS-oriented MUS-biased search algorithms to plain MUS extraction. Second, we integrated model rotation into resolution-based MUS extraction. Third, we introduced an enhancement to rotation, called eager rotation. Finally, we introduced a new enhancement, path strengthening, to resolution-based MUS extraction. We implemented the algorithms in the resolution-based MUS extractor Haifa-MUC, which, as a result, outperformed the leading MUS extractors MUSer2 and Minisatabb.

Algorithm 13 The recursive model rotation of [10], where $UnsatSet(S, h')$ is the subset of S's clauses that are unsatisfied by the assignment h'

```
1: function RMR(S, M, c, h)                          ▷ recursive model rotation
2:     for all x ∈ Var(S) do
3:         h' = h[x ← ¬x];                            ▷ swap assignment of x
4:         if UnsatSet(S, h') = {c'} and c' ∉ M then
5:             M = M ∪ {c'};
6:             RMR (S, M, c', h');
```

Algorithm 14 ERMR our modified version of RMP. K is a set of clauses that is initialized to c before calling ERMR. $K \subseteq M$ is an invariant, and hence ERMR is called at least as many times as RMR.

```
1: function ERMR(S, M, K, c, h)                          ▷ Initially K = {c}
2:     for all x ∈ Var(S) do
3:         h' = h[x ← ¬x];
4:         if UnsatSet(S, h') = {c'} and c' ∉ K then
5:             K = K ∪ {c'};
6:             if c' ∉ M then M = M ∪ {c'};
7:             ERMR (S, M, K, c', h');
```

Algorithm 15 Deletion-based MUS extraction enhanced by eager rotation and clause set refinement, where h is the satisfying assignment, and *core* is the unsatisfiable core

```
1: function MUS(unsatisfiable formula S)
2:     M = ∅;
3:     while true do
4:         choose c ∈ S \ M. If there is none, break;
5:         if SAT(S \ {c}) then
6:             K = {c};
7:             M = ERMR (S, c, M, K, h)
8:         else
9:             S = core;
```

Algorithm 16 An improvement based on path strengthening. In line 7 the literals defined by $\{\neg c_i \mid c_i \in P\}$ are assumptions.

```
1:  function MUS(unsatisfiable formula S)
2:      M = ∅;
3:      while true do
4:          choose c ∈ S \ M. If there is none, break;
5:          let P be the longest unique prefix
6:          discard clauses not backward reachable from □
7:          if SAT(S \ {c}, {¬c_i | c_i ∈ P}) then
8:              K = {c}; M = ERMR (S, c, M, K, h)
9:          else
10:             if ¬P not used in proof then S = core;
11:             else
12:                 S = S \ {c}
13:                 if condition then                    ▷ Heuristic. See text
14:                     SAT(S);                          ▷ guaranteed unsat
15:                     S = core;
```

	Base	rot	erot	erot_AD	erot_ABD	erot_AB2D	erot_AB2CD
Time	93931	48018	44335	36295	37798	32968	32918
Unsolved	30	12	10	8	13	8	8

	erot_AB2CD_rr	**erot_AB2CD_ps20**	MUSer2	Minisatabb
Time	30800	**27263**	59502	40485
Unsolved	6	**4**	17	8

Figure 7.1: Total run-time in sec. and number of unsolved instances for various solvers, when applied to the 295 instances from the 2011 MUS competition, excluding 12 instances which were not solved by any of the solvers (the time-out value of 1800 sec. was added to the run-time when a memory-out occured). Base is defined in Section 7.3, rot = Base+rotation, erot = Base+eager rotation. **A**, **B**, **C**, and **D** correspond to the optimizations defined in Section 7.2.1. '2' in AB2CD means that the optimization was invoked after the 2nd satisfiable result. 'rr' refers to redundancy removal combined with clause set refinement using MUSer2's scheme, described in Section 7.2.3. 'ps20' means that path strengthening with $N = 20$ was applied as described in Section 7.2.3.

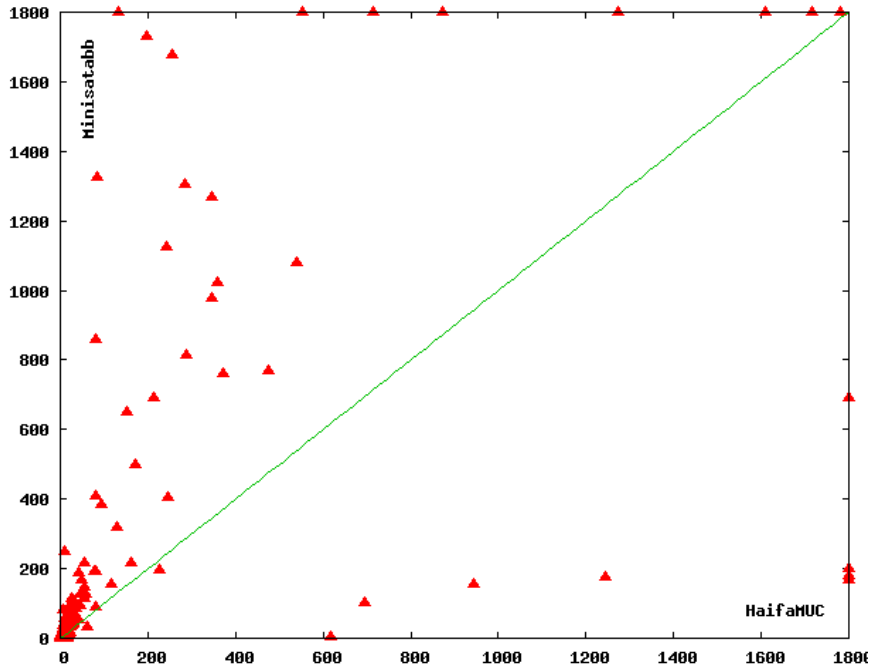

Figure 7.2: Direct comparison of the new best configuration of HAIFA-MUC erot_AB2CD_ps20 (X-Axis) and MINISATABB (Y-Axis).

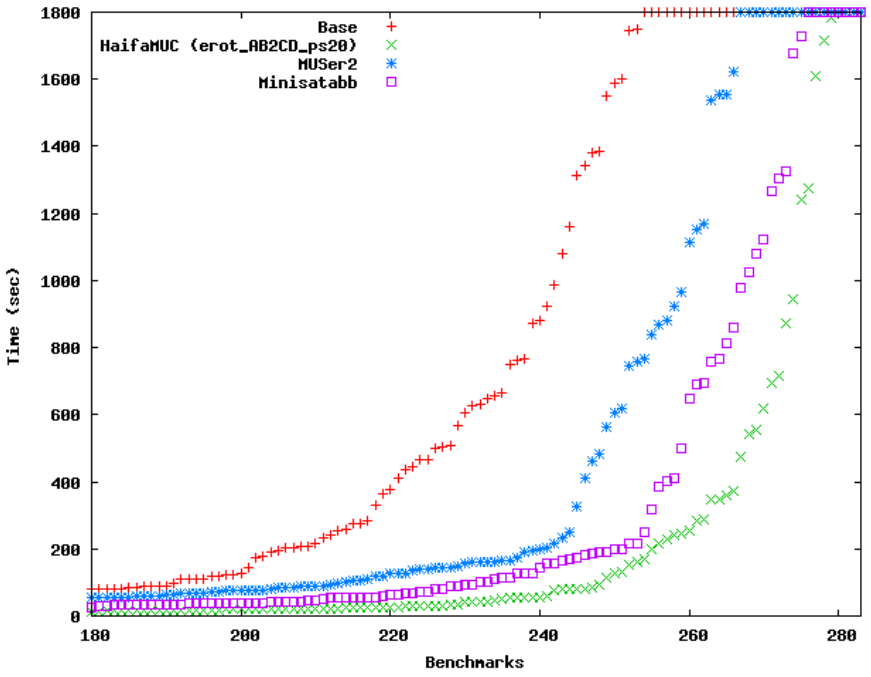

Figure 7.3: Comparison of Base, MUSer2, Minisatabb, and the new best configuration of Haifa-MUC erot_AB2CD_ps20. The graph shows the number of solved instances (X-Axis) per time-out in seconds (Y-Axis) for each solver.

Chapter 8

Summary and Future Research

Most or even all competitive CDCL-based SAT solvers have a restart policy, by which the solver is forced to backtrack to decision level 0 according to some criterion. Although not a sophisticated technique, there is mounting evidence that this technique has crucial impact on performance. The common explanation is that restarts help the solver avoid spending too much time in branches in which there is neither an easy-to-find satisfying assignment nor opportunities for fast learning of strong clauses. All existing techniques rely on a global criterion such as the number of conflicts learned as of the previous restart, and differ in the method of calculating the threshold after which the solver is forced to restart. This approach disregards, in some sense, the original motivation of focusing on bad branches. It is possible that a restart is activated right after going into a good branch, or that it spends all of its time in a single bad branch. Our contribution in Chapter 2 is a novel restart strategy which localize restarts, i.e., apply restarts according to measures local to each branch. This adds a dimension to the restart policy, namely the decision level in which the solver is currently in. Our experiments with both Minisat and Eureka show that with certain parameters this improves the run time by 15% - 30% on average (when applied to the 100 test benchmarks of SAT-race06), and reduces the number of time-outs as can be seen on Figure 2.1 and Figure 2.2. We also proposed a dynamic restart strategy that is in general less successful than the local ones, but performed successfully on unsatisfiable instances. It is interesting to check an additional SAT solver's

parameters like conflicts per decisions or implications per decisions and their effect on a solver's performance. A combined restart strategy between [14] and our local restart strategy can be a great direction for future research.

In Chapter 3 we presented an assignment stack shrinking technique that is intended to speed up the performance of modern complete SAT solvers, by making them more dynamic and local, and by enhancing the interrelation of the assigned variables. Shrinking was shown to be efficient in the SAT04 competition. However, existing studies lack the details of the shrinking algorithm. In addition, shrinkings performance was not tested in conjunction with the most modern techniques. In Section 3.2 we have described in detail different variations of the shrinking algorithm, including two new heuristics: one based on variable activity order, and the second on decision levels in clauses. We show that using shrinking is critical for solving well-known industrial benchmark families with the latest versions of Minisat as shown in Table 3.2 and Eureka as shown in Table 3.1. We also demonstrated that shrinking effects cannot be achieved by other modern algorithms. Shrinking is proving to be a useful concept that can be enhanced independently of the other components of SAT solvers, such as restart strategies or decision heuristics. The use of shrinking technique in parallel SAT solving showed great results in [95].

In the classical SAT interface, the solver receives one formula in CNF and is required to decide whether it is satisfiable or unsatisfiable. However, many practical applications [22, 33, 84, 88, 93] require solving a sequence of related SAT formulas. To answer the needs of such applications, the interface of modern SAT solvers since Minisat [32] enables incremental SAT solving under assumptions. Such interface allows the user to invoke the solving procedure multiple times, where each invocation checks the satisfiability status of the currently available set of clauses under an invocation-specific set of assumptions (that is, literals that hold solely for that specific invocations). The set of clauses can be extended, but not reduced, before each new invocation. Such interface allows one to handle a situation where a set of arbitrary clauses must hold for only one specific invocation by updating each clause in that set with a new *selector variable* and using the negation of that selector variable as an assumption [33, 69].

A naïve implementation of the incremental interface would invoke the following simple algorithm for each invocation. It would create from scratch and solve a formula containing all the available clauses and assumptions modeled as unit clauses. The modern incremental SAT solving algorithm, introduced in Minisat [32], uses a single SAT solver instance (invocation) to solve the entire sequence of formulas, and models assumptions as first decision variables. The main advantage of the described approach over the naïve one is that it reuses all the relevant learnt information, including conflict clauses and measures that guide decision, restart, and clause deletion heuristics. In addition all the learned clauses are implied by the formula regardless of the assumptions.

Independently of advances in incremental SAT solving, a breakthrough in the *non-incremental* SAT solving's efficiency was achieved with the SatELite [30] preprocessor. Preprocessing of CNF formulas is an invaluable technique when attempting to solve large formulas, such as those that model industrial verification problems. Unfortunately, the best combination of preprocessing techniques, which involve variable elimination combined with subsumption, is incompatible with incremental satisfiability. The reason is that soundness is lost if a variable is eliminated and later reintroduced. Look-ahead is a known technique to solve this problem, which simply blocks elimination of variables that are expected to be part of future instances. The problem with this technique is that it relies on knowing the future instances, which is impossible in several prominent domains.

In Chapter 5 we introduced efficient algorithms for incremental SAT solving under assumptions assuming the number of assumptions is significant. We found that effective propagation of assumptions is vital for ensuring SAT solving efficiency in a variety of applications. While the currently widely-used approach models assumptions as first decision variables, we proposed modeling assumptions as unit clauses. The advantage of our approach is that we allow the preprocessor to use assumptions while simplifying the formula. In particular, we demonstrated that the efficient SatELite preprocessor can easily be modified for use in our scheme, while it cannot be used with incremental single SAT solver instance. A notable advantage of our approach is that it can make preprocessing algorithms much more effective. However,

our initial scheme renders assumption-dependent conflict clauses unusable in subsequent invocations. To resolve the resulting problem of reduced learning power, in Section 5.4 we introduce an algorithm that transforms such temporary clauses into assumption-independent pervasive clauses as a post-processing step, thus improving learning efficiency. In addition, we developed an algorithm which improves the performance further by taking advantage of a limited form of look-ahead information, which we called step look-ahead, when available as presented in Section 5.5. In Tables 5.1 and 5.2 we showed that the combination of our algorithms outperforms LS on instances generated by a prominent industrial application. The empirical gap is especially significant for difficult unsatisfiable instances generated by a prominent industrial application in hardware validation.

The method as purposed in Chapter 5 is less effective when the number of assumptions is small or zero. In this case using one incremental instance of SAT solver without SatELite preprocessor can bring better performance than recreating an instance each call. The problem of using the SatELite preprocessor with one incremental instances of the SAT solver is in reintroduced variables which were eliminated in previous SAT solver calls as was previously described. In Chapter 4 we present a method called incremental preprocessing which is an effective algorithm for solving this problem by keeping track of eliminated variables and removed clauses. Our experiments with hundreds of industrial benchmarks show that it is much faster than the two known alternatives, namely full-preprocessing and no-preprocessing. Specifically, with a time-out of 4000 sec. it is able to reduce the number of time-outs by a factor of four and three, respectively as can be seen in Table 4.1 and in Figure 4.1. As follow up research combining between two approaches of Chapter 5 and Chapter 5 could clearly boost incremental SAT solving under assumptions and offer the first solution to the general problem: fully incremental SAT solving with SatELite preprocessing and assumption propagations. The idea to create a solver that uses a single SAT solver instance integrated with incremental preprocessing as in Chapter 5, and allows SatELite to fully propagate assumptions as in Chapter 5. To that end, we need to solve the problem of how to correctly recreate clauses, which were previously subsumed by other clauses that were only correct under certain

assumptions that are now released.

Various verification techniques are based on SATs capability to identify a small, or even minimal, unsatisfiable core in case the formula is unsatisfiable, i.e., a small subset of the clauses that are unsatisfiable regardless of the rest of the formula. In most cases it is not the core itself that is being used, rather it is processed further in order to check which clauses from a preknown set of Interesting Constraints (where each constraint is modeled with a conjunction of clauses) participate in the proof. Until [66] the standard practice was to minimize the core itself, and only then to find the interesting part of it. Our experiments show that this approach cannot compete with a solver that focuses on the high-level core. In Chapter 6 we introduced seven techniques which together result in an overall reduction of 55% in run time and 73% in the size of the resulting core, based on our experiments with hundreds of industrial test cases as can be found in Table 6.1 and Table 6.2. The optimized procedure is also better empirically than the assumptions-based minimization technique, and faster by more than an order of magnitude than the best known general MUS solver. A straight-forward direction for future research is to migrate some of the suggested optimizations to the assumptions-based approach. Related SAT problems may also benefit from these methods. First - it is possible that general SAT solving can be improved with some combination of optimizations **E** and **F**. Second, the same techniques can potentially expedite other methods in which the SAT component needs to extract only partial information from the resolution proof, like interpolation-based model checking [60]. In interpolation only a small part of the proof is necessary in order to generate the interpolant, and we want to explore possibilities to minimize that part and decrease the overall run time with variants of the methods suggested here.

In Chapter 7 we proposed a number of algorithms for speeding up MUS extraction. First in Section 7.2.1, we demonstrated how to apply techniques used in the past in a Group MUS extraction algorithm described in Chapter 6 to speed up a resolution-based MUS extraction. Second in Section 7.2.2, we show that model rotation, presented in the context of assumption-based MUS extraction, can also be used with resolution-based MUS extraction. Third, we introduce an improvement to rotation, called eager rotation. Finally

in Section 7.2.3, we proposed a new technique for speeding-up resolution-based MUS extraction, called path strengthening. We integrated the above techniques into the publicly available resolution-based MUS extractor HAIFA-MUC, which, as a result, now outperforms leading MUS extractors that were presented in Figure 7.1. As future research it is interesting to perform a deeper analysis of a resolution graph and finding additional literals that could be used in a SAT solver call preceded by clause removal as part of the MUS algorithm. For example, literals that can be found on every path from the removed clause to the empty clause.

Bibliography

[1] *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001.

[2] Roberto Asn, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodrguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in sat. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.

[3] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental sat solving with assumptions: Application to mus extraction. In Jrvisalo and Gelder [46], pages 309–317.

[4] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI*, pages 399–404, 2009.

[5] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT 2003*, volume 2919 of *LNCS*, pages 341–355, 2003.

[6] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In Ruzena Bajcsy, editor, *IJCAI'93*, pages 276–281. Morgan Kaufmann, 1993.

[7] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.

[8] Ramón Béjar, Felip Manyà, Alba Cabiscol, Cèsar Fernández, and Carla P. Gomes. Regular-sat: A many-valued approach to solving combinatorial problems. *Discrete Applied Mathematics*, 155(12):1613–1626, 2007.

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

[10] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD'11*, pages 37–40, 2011.

[11] Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *JSAT*, 8(1/2):123–128, 2012.

[12] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

[13] Daniel Le Berre and Laurent Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In Hoos and Mitchell [42], pages 321–344.

[14] Biere. Adaptive restart control for conflict driven sat solvers. In *Proc. 11th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'08), Lecture Notes in Computer Science (LNCS)*, volume 4996. Springer, 2008.

[15] Armin Biere. Lingeling and Plingeling. `http://fmv.jku.at/lingeling/`.

[16] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[17] Armin Biere. *Bounded Model Checking*, chapter 14, pages 455–481. Volume 185 of Biere et al. [20], February 2009.

[18] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC*, pages 317–320, 1999.

[19] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

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

[21] Hans Kleine Büning and Xishun Zhao, editors. *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*. Springer, 2008.

[22] Gianpiero Cabodi, Luciano Lavagno, Marco Murciano, Alex Kondratyev, and Yosinori Watanabe. Speeding-up heuristic allocation, scheduling and binding with SAT-based abstraction/refinement techniques. *ACM Trans. Design Autom. Electr. Syst.*, 15(2), 2010.

[23] John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.

[24] SAT 2011 Competition. Group-oriented mus track: ranking of solvers. `http://www.cril.univ-artois.fr/SAT11/results/ranking.php?idev=49`.

[25] SAT 2011 Competition. Plain mus track: ranking of solvers. `http://www.cril.univ-artois.fr/SAT11/results/ranking.php?idev=48`.

[26] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978. Corrigendum: SIAM J. Comput. 10(3): 612 (1981).

[27] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A clause-based heuristic for SAT solvers. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2005.

[28] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT'06*, pages 36–41, 2006.

[29] Christian Desrosiers, Philippe Galinier, Alain Hertz, and Sandrine Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *J. Comb. Optim.*, 18(2):124–150, 2009.

[30] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[31] Niklas Eén, Alan Mishchenko, and Nina Amla. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In *FMCAD*, pages 181–188, 2010.

[32] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[33] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.

[34] Niklas Een and Niklas Sorensson. Minisat v2.0 (beta). In *Solvers description, SAT-race*. 2006. `http://fmv.jku.at/sat-race-2006/descriptions/27-minisat2.pdf`.

[35] Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying smt in symbolic execution of microcode. In *FMCAD*, pages 121–128, 2010.

[36] Roman Gershman, Maya Koifman, and Ofer Strichman. An approach for extracting a small unsatisfiable core. *Formal Methods in System Design*, 33(1-3):1–27, 2008.

[37] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *DATE*, pages 10886–10891. IEEE Computer Society, 2003.

[38] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[39] ric Grgoire, Bertrand Mazure, and Cdric Piette. Extracting muses. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 387–391. IOS Press, 2006.

[40] A. Gupta. Learning abstractions for model checking. Master's thesis, CMU, 2006.

[41] Aarti Gupta, Malay K. Ganai, Zijiang Yang, and Pranav Ashar. Iterative abstraction using sat-based bmc with proof analysis. In *ICCAD*, pages 416–423. IEEE Computer Society / ACM, 2003.

[42] Holger H. Hoos and David G. Mitchell, editors. *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2005.

[43] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In Manuela M. Veloso, editor, *IJCAI*, pages 2318–2323, 2007.

[44] Mark Iwen and Amol Dattatraya Mali. Dsatz: A directional sat solver for planning. In *ICTAI*, pages 199–208. IEEE Computer Society, 2002.

[45] Warren A. Hunt Jr. and Fabio Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

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

[47] Zurab Khasidashvili, Daher Kaiss, and Doron Bustan. A compositional theory for post-reboot observational equivalence checking of hardware. In *FMCAD*, pages 136–143. IEEE, 2009.

[48] Zurab Khasidashvili and Alexander Nadel. Implicative simultaneous satisfiability and applications. In *HVC'11 (to appear)*, 2011.

[49] Daniel Kroening. *Software Verification*, chapter 16, pages 505–532. Volume 185 of Biere et al. [20], February 2009.

[50] Stefan Kupferschmid, Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Incremental preprocessing methods for use in BMC. *Formal Methods in System Design*, 39(2):185–204, 2011.

[51] Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up MUS extraction. In Jrvisalo and Gelder [46], pages 276–292.

[52] Mark H. Liffiton, Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João Marques-Silva, and Karem A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints*, 14(4):415–442, 2009.

[53] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40:1–33, January 2008.

[54] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. In *ISTCS*, pages 128–133, 1993.

[55] Lynce, Luis Baptista, and Joao P. Marques Silva. Stochastic systematic search algorithms for satisfiability. In *LICS Workshop on Theory and Applications of Satisfiability Testing*, pages 190–204, 2001.

[56] Inês Lynce and João Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 136–141. Springer, 2006.

[57] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In Hoos and Mitchell [42], pages 360–375.

[58] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, chapter 4, pages 131–153. Volume 185 of Biere et al. [20], February 2009.

[59] Kenneth L. McMillan. Interpolation and sat-based model checking. In Jr. and Somenzi [45], pages 1–13.

[60] Kenneth L. McMillan. Interpolation and sat-based model checking. In Jr. and Somenzi [45], pages 1–13.

[61] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[62] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC* [1], pages 530–535.

[63] Alex Nadel, Vadim Ryvchin, and Ofer Strichman. Preprocessing in incremental SAT. Technical Report IE/IS-2012-02, Industrial Engineering, Technion, 2012. Available also from http://ie.technion.ac.il/∼ofers/publications/sat12t.pdf.

[64] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew Univeristy of Jerusalem, Jerusalem, Israel, November 2002.

[65] Alexander Nadel. *Understanding and improving a modern SAT solver*. PhD thesis, Tel Aviv University, Tel Aviv, Israel, August 2009.

[66] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD*, pages 221–229. IEEE, 2010.

[67] Alexander Nadel, Moran Gordon, Amit Palti, and Ziyad Hanna. Eureka-2006 SAT solver. http://fmv.jku.at/sat-race-2006/descriptions/4-Eureka.pdf.

[68] Alexander Nadel and Vadim Ryvchin. Experimental results for the SAT'10 paper "Assignment stack shrinking". http://www.cs.tau.ac.il/research/alexander.nadel/sat10_ass_res.xlsx.

[69] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *DAC*, pages 518–523. ACM, 2004.

[70] Christos H. Papadimitriou and David Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, 37(1):2–13, 1988.

[71] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[72] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. SAT competition'07, 2007.

[73] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *STTT*, 7(2):156–173, 2005.

[74] Jussi Rintanen. *Planning and SAT*, chapter 15, pages 483–504. Volume 185 of Biere et al. [20], February 2009.

[75] Vadim Ryvchin. Benchmarks + results. http://ie.technion.ac.il/~$ofers/sat11.html.

[76] Vadim Ryvchin. Haifa-muc link. https://www.dropbox.com/s/uhxeps7atrac82d/Haifa-MUC.7z.

[77] Vadim Ryvchin and Ofer Strichman. Local restarts. In Büning and Zhao [21], pages 271–276.

[78] Vadim Ryvchin and Ofer Strichman. Faster extraction of high-level minimal unsatisfiable cores. In Sakallah and Simon [79], pages 174–187.

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

[80] Ohad Shacham and Karen Yorav. On-the-fly resolve trace minimization. In *DAC*, pages 594–599. IEEE, 2007.

[81] Ofer Shtrichman. Prunning techniques for the SAT-based bounded model checking problem. In *proc. of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Edinburgh, September 2001.

[82] João P. Marques Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *ISMVL'10*, pages 9–14. IEEE Computer Society, 2010.

[83] João P. Marques Silva and Inês Lynce. On improving MUS extraction algorithms. In Sakallah and Simon [79], pages 159–173.

[84] João P. Marques Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *FTCS*, pages 152–161, 1997.

[85] Carsten Sinz. SAT-Race 2006. `http://fmv.jku.at/sat-race-2006/`.

[86] Mate Soos. Cryptominisat2. `http://www.msoos.org/cryptominisat2`.

[87] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[88] Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In Tiziana Margaria and Thomas F. Melham, editors, *CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2001.

[89] Hans van Maaren and Siert Wieringa. Finding guaranteed MUSes fast. In Büning and Zhao [21], pages 291–304.

[90] Miroslav N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *Proc. Design, Automation and Test in Europe Conference and Exhibition*, pages 28–35, 2002.

[91] M.N. Velev and R.E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 226–231, 2001.

[92] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: a new incremental satisfiability engine. In *IEEE/ACM Design Automation Conference (DAC)*, 2001.

[93] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC* [1], pages 542–545.

[94] Siert Wieringa. Understanding, improving and parallelizing MUS finding using model rotation. In Michela Milano, editor, *CP'12*, volume 7514 of *Lecture Notes in Computer Science*, pages 672–687. Springer, 2012.

[95] Siert Wieringa and Keijo Heljanko. Concurrent clause strengthening. In Jrvisalo and Gelder [46], pages 116–132.

[96] Poul Frederick Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. Combining decision diagrams and sat procedures for efficient symbolic model checking. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2000.

[97] Hantao Zhang. *Combinatorial Designs by SAT Solvers*, chapter 17, pages 533–568. Volume 185 of Biere et al. [20], February 2009.

[98] Lintao Zhang and Sharad Malik. Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula. In *6th International Conference on Theory and Applications of Satisfiability Testing: SAT 2003*, May 2003.