The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Premise Set Caching for Enumerating Minimal Correction Subsets

Alessandro Previti
HIIT, Department of Computer Science
University of Helsinki, Finland
alessandro.previti@helsinki.fi

Carlos Mencía
Department of Computer Science
University of Oviedo, Spain

Matti Järvisalo
HIIT, Department of Computer Science
University of Helsinki, Finland

Joao Marques-Silva
LASIGE, Faculty of Science
University of Lisbon, Portugal

Abstract

Methods for explaining the sources of inconsistency of overconstrained systems find an ever-increasing number of applications, ranging from diagnosis and configuration to ontology debugging and axiom pinpointing in description logics. Efficient enumeration of minimal correction subsets (MCSes), defined as sets of constraints whose removal from the system restores feasibility, is a central task in such domains. In this work, we propose a novel approach to speeding up MCS enumeration over conjunctive normal form propositional formulas by caching of so-called premise sets (PSes) seen during the enumeration process. Contrasting to earlier work, we move from caching unsatisfiable cores to caching PSes and propose a more effective way of implementing the cache. The proposed techniques noticeably improves on the performance of state-of-the-art MCS enumeration algorithms in practice.

Introduction

The analysis of overconstrained systems finds a wide range of important AI applications, including numerous problems of diagnosis, such as type debugging (Stuckey, Sulzmann, and Wazny 2003; de la Banda, Stuckey, and Wazny 2003; Bailey and Stuckey 2005), software fault localization (Jose and Majumdar 2011; Roychoudhury and Chandra 2016), spreadsheet debugging (Jannach et al. 2014), axiom pinpointing in description logics (Baader and Suntisrivaraporn 2008; Sebastiani and Vescovi 2009), and model based diagnosis in general (Reiter 1987), among many others. Central to the analysis of overconstrained systems are the tasks of finding minimal explanations of inconsistency and minimal relaxations to recover consistency. In specific settings, enumeration of explanations or relaxations, i.e., the task of finding many instead of only a single explanation or relaxation, is crucial for understanding the underlying sources of inconsistency. For example, enumeration of minimal relaxations is essential in software fault localization (Jose and Majumdar 2011). Furthermore, in settings where the automatic analysis of minimal relaxations or minimal explanations is viable, there is a natural demand for high-performance algorithms, capable of computing high numbers of minimal relaxations or explanations.

The focus of this work is on the central task of enumerating minimal relaxations in the concrete setting of analyzing overconstrained (or inconsistent) propositional formulas in conjunctive normal form (CNF). In the context of CNFs formulas, minimal relaxations to restore consistency are called minimal correction subsets (MCSes), whereas minimal explanations of inconsistency are referred to as minimal unsatisfiable subsets (MUSes). The development of practical algorithms for MCS extraction has received notable attention recently (Bailey and Stuckey 2005; Liffiton and Sakallah 2008; Felfernig, Schubert, and Zehentner 2012; Nöhrer, Biere, and Egyed 2012; Marques-Silva et al. 2013; Bacchus et al. 2014; Grégoire, Lagniez, and Mazure 2014; Mencía, Previti, and Marques-Silva 2015; Mencía et al. 2016). Furthermore, recent algorithmic advances for the computationally hard task of extracting a single MCS has enabled more efficient practical approaches to MCS enumeration (Marques-Silva et al. 2013; Mencía, Previti, and Marques-Silva 2015).

Here we take on caching (or memoization) as a complementary approach to improving MCS enumeration. While subformula caching has been earlier used to speeding up propositional model counting (Sang et al. 2004; Thurley 2006; Kitching and Bacchus 2007; Bacchus, Dalmao, and Pitassi 2009; Beame et al. 2010; Kopp, Singla, and Kautz 2016), caching in the context of MCS enumeration was only very recently proposed as a means of further scaling up MCS enumeration (Previti et al. 2017). Specifically, Previti et al. (2017) proposed caching unsatisfiable subsets of constraints seen during the MCS enumeration process within a SAT-based MCS enumeration algorithm, thereby avoiding some of the potentially time-consuming SAT solvers calls by querying the cache—implemented itself using another SAT solver—for an unsatisfiable core that would render the SAT solver call unnecessary. The empirical evidence provided in (Previti et al. 2017) suggests that the proposed approach of caching unsatisfiable subsets of constraints seen during MCS enumeration can speed up a specific MCS enumeration approach (namely, ELS, or extended linear search).

In this work, we take the idea of employing caching as a means of speeding up state-of-the-art MCS enumeration algorithms further. Specifically, instead of caching unsatisfiable subsets, we propose to cache so-called *premise sets* (Kullmann 2011). While unsatisfiable subsets and

premise sets are closely related concepts—as we will explain—a single cached premise set can account for multiple unsatisfiable subsets. Thus, by caching premise sets instead of unsatisfiable subsets, caching can—and, as we will empirically show, will—become noticeably more effective in speeding up MCS enumeration. Furthermore, instead of the rudimentary approach to implementing the cache proposed in (Previti et al. 2017), we develop an alternative, dedicated way of implementing caching of premise sets, which brings further practical gains in terms of the efficiency of MCS enumeration in practice.

Satisfiability and MCSes

Propositional formulas are represented in conjunctive normal form (CNF) and defined over a set of propositional variables $X = \{x_1, \ldots, x_n\}$. A literal is a variable x or its negation $\neg x$. A clause is a disjunction of literals, and a formula $\mathcal{F}$ is a conjunction of clauses. CNF formulas can also be viewed as sets of clauses, and clauses as sets of literals. Both representations are used interchangeably. $L(\mathcal{F})$ will refer to the set of literals appearing in $\mathcal{F}$.

A truth assignment is a (partial) mapping from X to $\{0, 1\}$, $\mu : X \to \{0, 1\}$. The truth assignment is referred to as total if the mapping is total. Truth assignments induce valuations of literals, clauses and formulas, as follows. A literal x takes the value assigned to variable x, and $\neg x$ takes the complemented value. A clause c takes value 0 (i.e., is falsified) if all of its literals take value 0; otherwise it takes value 1 (i.e., it is satisfied). A formula $\mathcal{F}$ takes value 1 if all of its clauses take value 1; otherwise it takes value 0. With a slight abuse of notation, we write $\mu(c)$ and $\mu(\mathcal{F})$ for the valuation of a clause c and of a formula $\mathcal{F}$ given the truth assignment μ. A formula $\mathcal{F}$ is satisfiable if there exists an assignment μ such that $\mu(\mathcal{F}) = 1$ (also represented as $\mu \models \mathcal{F}$); otherwise $\mathcal{F}$ is unsatisfiable. For each μ such that $\mu(\mathcal{F}) = 1$, we say that μ is a model of $\mathcal{F}$. Boolean satisfiability (SAT) is the decision problem for propositional formulas, well-known to be NP-complete (Cook 1971).

Logical entailment is characterized in terms of models. Given $\mathcal{F}$ and $\mathcal{G}$ two propositional formulas, $\mathcal{F}$ entails $\mathcal{G}$ (written $\mathcal{F} \models \mathcal{G}$) iff all the models of $\mathcal{F}$ are also models of $\mathcal{G}$. Throughout the paper, we will refer to backbone (or implied) literals of a formula $\mathcal{F}$ (Kilby et al. 2005), defined as literals l such that $\mathcal{F} \models l$.

Central to this work is the concept of minimal correction subsets (MCSes), which are defined in the standard way for a given unsatisfiable CNF formula $\mathcal{F}$ as follows (Liffiton and Sakallah 2008).

Definition 1 *$\mathcal{C} \subseteq \mathcal{F}$ is a* minimal correction subset *(MCS) if and only if $\mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall c \in \mathcal{C}, \mathcal{F} \setminus (\mathcal{C} \setminus \{c\})$ is unsatisfiable.*

Tightly connected to MCSes are the concepts of maximal satisfiable subsets (MSSes) and minimal unsatisfiable subsets (MUSes), which will also by useful throughout.

Definition 2 *$\mathcal{S} \subseteq \mathcal{F}$ is a* maximal satisfiable subset *(MSS) if and only if $\mathcal{S}$ is satisfiable and $\forall c \in \mathcal{F} \setminus \mathcal{S}, \mathcal{S} \cup \{c\}$ is unsatisfiable.*

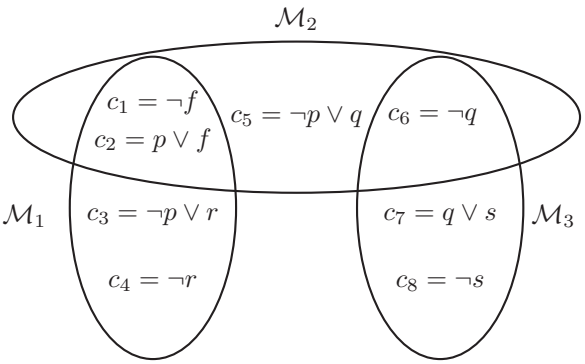

Figure 1: Unsatisfiable CNF formula and its core structure.

Definition 3 *$\mathcal{M} \subseteq \mathcal{F}$ is a* minimal unsatisfiable subset *(MUS) of $\mathcal{F}$ if and only if $\mathcal{M}$ is unsatisfiable and $\forall c \in \mathcal{M}, \mathcal{M} \setminus \{c\}$ is satisfiable.*

We will refer to (not necessarily minimal) unsatisfiable subsets as *unsatisfiable cores*. Each MCS $\mathcal{C}$ is the complement of some MSS $\mathcal{S}$ with respect to $\mathcal{F}$. Moreover, a well-known hitting set duality tightly relates MUSes and MCSes (Reiter 1987), and has been investigated in different settings (Birnbaum and Lozinskii 2003; Bailey and Stuckey 2005; Liffiton and Sakallah 2008; Slaney 2014).

Example 1 *Consider the unsatisfiable CNF formula $c_1 \wedge \cdots \wedge c_8$ as shown in Figure 1. This formula has three MUSes: $\mathcal{M}_1 = \{c_1, c_2, c_3, c_4\}$, $\mathcal{M}_2 = \{c_1, c_2, c_5, c_6\}$, $\mathcal{M}_3 = \{c_6, c_7, c_8\}$. Two of a total of 12 MCSes of this formula are $\mathcal{C}_1 = \{c_3, c_6\}$ and $\mathcal{C}_2 = \{c_4, c_5, c_7\}$, representing minimal hitting sets of the set of MUSes.*

In the following, we will consider the general setting, where a formula is composed of two disjoint sets of clauses, i.e., $\mathcal{F} = \mathcal{F}_H \cup \mathcal{F}_S$ (Biere et al. 2009, Chap. 19), where $\mathcal{F}_H$ denotes the hard clauses (which must be satisfied) and $\mathcal{F}_S$ denotes the soft clauses (which may be relaxed, i.e., left not satisfied). In this standard setting in terms of MCS enumeration, MCSes, MSSes, and MUSes are defined as minimal/maximal subsets *of the soft clauses* which satisfy the respective conditions when considered together with the hard clauses.

Algorithms for MCS extraction fall into two broad categories. One approach involves a modified search algorithm (Bacchus et al. 2014), where the condition of minimality is enforced. Another approach involves sequences of calls to a SAT solver (i.e., an NP oracle (Bailey and Stuckey 2005; Liffiton and Sakallah 2008; Felfernig, Schubert, and Zehentner 2012; Nöhrer, Biere, and Egyed 2012; Marques-Silva et al. 2013; Grégoire, Lagniez, and Mazure 2014; Mencía, Previti, and Marques-Silva 2015; Mencía et al. 2016). For this second approach, important differences include how the calls to the SAT solver are organized, and optimizations aiming at reducing the number of SAT solver calls.

Algorithm 1: MCS enumeration (general structure)

```
1  Procedure MCS-Enum(F = {F_H, F_S})
2      (st, μ) = SAT(F_H)
3      while st do
4          C ← ComputeMCS(F, μ)
5          ReportMCS(C)
6          BlockMCS(F, C)
7          (st, μ) = SAT(F_H)
       // All MCSes of F reported
```

MCS Enumeration

We continue by overviewing MCS extraction and enumeration approaches, focusing on state-of-the-art algorithms which are based on querying a SAT solver on subsets of an unsatisfiable formula $\mathcal{F} = \{\mathcal{F}_H, \mathcal{F}_S\}$ partitioned into hard and soft clauses. We will represent a call to the SAT solver by $(st, \mu, C) \leftarrow \mathsf{SAT}(\mathcal{G})$, where st is a Boolean value denoting whether the formula $\mathcal{G}$ is satisfiable or not, and μ is a model of $\mathcal{G}$ whenever st is true. If unsatisfiable, C will store an unsatisfiable core[1], that is an unsatisfiable subset of $\mathcal{G}$.

Most of the best-performing MCS enumeration algorithms share the general structure depicted in Algorithm 1. Iteratively, a model $\mu \models \mathcal{F}_H$ is computed. This model is then used as a *seed* for computing an MCS $\mathcal{C}$ of $\mathcal{F}$, which will be a subset of the clauses in $\mathcal{F}_S$ not satisfied by μ. After computing an MCS, it is reported and blocked in order to avoid repetitions along the enumeration process. The blocking of an MCS $\mathcal{C}$ is done by adding the clause $\bigvee_{l \in L(\mathcal{C})}(l)$ to $\mathcal{F}_H$. This enforces subsequent models of $\mathcal{F}_H$ to satisfy at least one of the clauses in $\mathcal{C}$, so that no superset of $\mathcal{C}$ will be ever considered afterwards in the lookout for new MCSes. Algorithm 1 terminates when $\mathcal{F}_H$ becomes unsatisfiable, performing as many iterations as the number of MCSes of $\mathcal{F}$, which can be exponential in the worst case (Liffiton and Sakallah 2008; O'Sullivan et al. 2007).

The efficiency of Algorithm 1 is highly dependent on the underlying method for computing one MCS at each iteration. To this end, a large body of MCS extraction algorithms have been proposed in the literature, with different query complexities that provide theoretical guarantees in efficiency, as well as optimization techniques often useful in practice (Marques-Silva et al. 2013).

In general, these algorithms maintain a partition of $\mathcal{F} = (\mathcal{S}, \mathcal{U})$, where $\mathcal{S}$ is a satisfiable subformula and $\mathcal{U}$ represents its complement, i.e., clauses either known to be inconsistent with $\mathcal{S}$ and thus belonging to an MCS or clauses that still need to be tested. $\mathcal{S}$ is initialized with all the clauses in $\mathcal{F}$ satisfied by the seed $\mu \models \mathcal{F}_H$, and $\mathcal{U}$ is initially composed of the clauses falsified by μ. MCS extraction algorithms aim at extending $\mathcal{S}$ as much as possible, keeping the invariant that $\mathcal{S}$ is satisfiable. At termination, $\mathcal{S}$ will be an MSS of $\mathcal{F}$, and $\mathcal{U}$ will correspond to an MCS.

[1]On the implementation level, using the so-called assumptions interface of a SAT solver, cores are represented via selector (or blocking/relaxation/assumption) variables with which clauses are instrumented.

Algorithm 2: LBX

```
1  Function LBX(F, μ)
2      (S, U) ← InitPartition(F, μ)
3      (L, B) ← (L(U), ∅)
4      while L ≠ ∅ do
5          l ← RemoveLiteral(L)
6          (st, μ, C) = SAT(S ∪ B ∪ {l})
7          if st then
8              (S, U) ← UpdateSatClauses(μ, S, U)
9              L ← L ∩ L(U)
10         else
11             B ← B ∪ {¬l}
12     return F \ S                    // MCS of F
```

Conceptually, one of the simplest MCS extraction algorithms is *linear search* (LS), sometimes referred to as the *grow* procedure (Bailey and Stuckey 2005). LS iterates over the clauses $c \in \mathcal{U}$, calling the SAT solver on $\mathcal{S} \cup \{c\}$. If satisfiable, c is moved to $\mathcal{S}$; otherwise, it remains in $\mathcal{U}$. This algorithm performs as many queries to the SAT solver as the number of clauses in $\mathcal{F}_S$ in the worst case, i.e., it has a query complexity of $\mathcal{O}(|\mathcal{F}_S|)$. Linear search, as well as other algorithms, has been improved by means of optimization techniques (Marques-Silva et al. 2013), including the employment of models found in satisfiable calls, backbone literals or disjoint unsatisfiable cores. The resulting algorithm is referred to as *enhanced linear search* (ELS).

Several other algorithms whose query complexity is characterized in terms of the number of clauses in the formula have been proposed. Some of them present superlinear query complexity, such as *dichotomic search* (O'Sullivan et al. 2007), *FastDiag* (Felfernig, Schubert, and Zehentner 2012), or CMP (Grégoire, Lagniez, and Mazure 2014) in its basic form (without optimizations). Other recent algorithms require a number of calls sublinear on the number of soft clauses, such as LOPZ or UBS (Mencía et al. 2016). Besides, based on the use of SAT with preferences, *relaxation search* (RS) (Bacchus et al. 2014) requires a single call to a modified SAT solver for the task of computing an MCS.

Alternatively, algorithms that exhibit a query complexity bounded on the number of variables of $\mathcal{F}$ exist as well. This holds, e.g., for LBX (for Literal-Based eXtractor), proposed in (Mencía, Previti, and Marques-Silva 2015) and shown in Algorithm 2. LBX stems from the observation that the backbone literals entailed from an MSS falsify all the clauses in the corresponding MCS. Thus, the computation of one MCS is reduced to the task of identifying the necessary backbone literals of $\mathcal{S}$. In addition to the sets $\mathcal{S}$ and $\mathcal{U}$, it maintains two additional sets: $\mathcal{L}$ containing literals to be tested and $\mathcal{B}$, consisting of the backbone literals identified so far. LBX starts by initializing the partition $(\mathcal{S}, \mathcal{U})$ from the model $\mu \models \mathcal{F}_H$ that is given as an argument (line 2). Then, the set $\mathcal{L}$ is initialized with all the literals appearing in $\mathcal{U}$, i.e., $L(\mathcal{U})$, while $\mathcal{B}$ is initially empty. At each step, while there are literals in

$\mathcal{L}$, one literal l is selected and removed from $\mathcal{L}$, and the SAT solver is queried on $\mathcal{S} \cup \mathcal{B} \cup \{l\}$. The set $\mathcal{B}$ is added to the formula to speed up the solving process, with no effect on its satisfiability. If the outcome is unsatisfiable, $\neg l$ is recorded as a backbone literal and added to $\mathcal{B}$ (line 11); otherwise all the clauses in $\mathcal{U}$ satisfied by the computed model μ are moved to $\mathcal{S}$ (line 8) and the set $\mathcal{L}$ is updated so that it does not contain any literal not in $L(\mathcal{U})$ (line 9).

An algorithm that has been shown very effective at MCS enumeration is CLD (Marques-Silva et al. 2013). This algorithm makes use of so-called D clauses, defined as $\bigvee_{l \in L(\mathcal{U})}(l)$. In contrast to other methods, CLD tries to extend $\mathcal{S}$ by calling the SAT solver on the formula $\mathcal{S} \cup \{D\}$, i.e., it checks whether there exists *at least* one clause in $\mathcal{U}$ that is consistent with $\mathcal{S}$. Satisfiable outcomes result in the update of the sets $\mathcal{S}$ and $\mathcal{U}$ according to the obtained models. CLD terminates upon an unsatisfiable call, meaning that the set $\mathcal{U}$ represents an MCS.

Many of the algorithms presented above have specifically targeted the task of extracting a single MCS from hard unsatisfiable CNF formulas. This holds, e.g., for CMP, LOPZ, and UBS which, consequently, have not been implemented nor evaluated for enumerating MCSes. To our best knowledge, the best-performing MCS enumeration algorithms among the ones mentioned above are ELS and CLD (Marques-Silva et al. 2013).

Recently, considerable improvements in MCS enumeration have been made by instrumenting a caching (or memoization) mechanism within the algorithm ELS (Previti et al. 2017). This approach make use of the observation that queries to the SAT solver are similar to each other in the computation of subsequent MCSes. Thus, whenever a call to the SAT solver on a subformula is found unsatisfiable, the resulting unsatisfiable core is stored in a global cache. Afterwards, for testing the satisfiability of $\mathcal{S} \cup \{c\}$ at each iteration of ELS, the cache is queried first in order to determine whether there exists a previously found unsatisfiable core contained in $\mathcal{S} \cup \{c\}$. If it is the case, the formula is declared unsatisfiable, hence saving a potentially expensive call to the SAT solver. Otherwise, the SAT solver is called normally. Implementation-wise, the algorithm makes use of selector variables s_i that are associated with each clause c_i in the formula, and are used to activate (insert) or deactivate (delete) the clauses. The cache is implemented by means of a Horn formula $\mathcal{D}$, which can be queried in polynomial time, containing for each unsatisfiable core C found in the enumeration process, a clause $\bigvee_{c_i \in C}(\neg s_i)$. For querying the cache on a formula $\mathcal{S} \cup \{c\}$, the selector variables associated to it, i.e., $\mathcal{A} = \{s_j \mid c_j \in \mathcal{S} \cup \{c\}\}$, are set as assumptions, calling a SAT solver on $\mathcal{D} \cup \mathcal{A}$.

Premise Sets

With background on MCS enumeration and the recently proposed core caching presented in the specific context of the ELS algorithm, we will now turn to the main contributions of this work. We start by defining premise sets (Kullmann 2011), which will be objects we will cache during MCS enumeration.

Definition 4 *A set of clauses P is a* premise set *(PS) for a clause c if $P \models c$. A premise set P is* minimal *(an MPS) if $\forall P' \subset P$ we have that $P' \not\models c$.*

Unless explicitly specified, we will identify a PS as a set of clauses that entails a unit clause (i.e, an implied or backbone literal).

Example 2 *Consider the CNF formula*

$$\mathcal{F} = \overset{c_1}{x_1} \wedge \overset{c_2}{(\neg x_1 \vee x_2)} \wedge \overset{c_3}{(\neg x_3 \vee x_4)} \quad (1)$$

Here x_2 is an implied literal and the minimal premise set that implies x_2 is the set of clauses $\{c_1, c_2\}$. Also x_1 is an implied literal and the minimal premise set for it is the unit clause c_1 itself.

Most modern SAT solvers come with the ability to compute cores of unsatisfiable formulas. Recall that a core is a subset of the original formula that is still unsatisfiable. Suppose now to have a satisfiable formula $\mathcal{S}$ such that $\mathcal{S} \models l$. By definition, the formula $\mathcal{S} \cup \{\neg l\}$ is unsatisfiable, and a SAT solver will provide a set of clauses $C \subseteq \mathcal{S} \cup \{\neg l\}$ representing a core. As $\mathcal{S}$ is satisfiable by construction, C is guaranteed to contain $\neg l$, and $C \setminus \{\neg l\}$ is the PS we are looking for.

The following proposition relates unsatisfiable cores and PSes; essentially, cores can be obtained from known PSes.

Proposition 1 *Suppose $\mathcal{P}_1$ and $\mathcal{P}_2$ are two PSes of a formula $\mathcal{F}$ such that $\mathcal{P}_1 \models l$ and $\mathcal{P}_2 \models \neg l$. Then $\mathcal{P}_1 \cup \mathcal{P}_2$ is a core of $\mathcal{F}$.*

The following two examples give intuition on why—instead of caching unsatisfiable cores—it can be beneficial to consider caching premise sets during the MCS enumeration process.

Example 3 *Recall the unsatisfiable CNF formula $c_1 \wedge \cdots \wedge c_8$ as shown in Figure 1 in Example 1. This formula has three MUSes: $\mathcal{M}_1 = \{c_1, c_2, c_3, c_4\}$, $\mathcal{M}_2 = \{c_1, c_2, c_5, c_6\}$, $\mathcal{M}_3 = \{c_6, c_7, c_8\}$. Notice that $\mathcal{M}_1$ and $\mathcal{M}_2$ share the two clauses c_1 and c_2. The set $\{c_1, c_2\}$ is a PS which implies the literal p. Two possible MCSes for this formula are $\mathcal{C}_1 = \{c_3, c_6\}$ and $\mathcal{C}_2 = \{c_4, c_5, c_7\}$. The two clauses c_3 and c_5 share the literal $\neg p$. So, suppose that $\mathcal{C}_1 = \{c_3, c_6\}$ was the first MCS computed using Algorithm 2. When we compute the next MCS $\mathcal{C}_2 = \{c_4, c_5, c_7\}$, LBX as presented in Algorithm 2 tests again the literal l calling the SAT solver on the formula $\{c_1, c_2, c_3, c_6, c_8\} \cup \{\neg p\}$. However, because of the previously computed MCS $\mathcal{C}_1$ we know already that $\{c_1, c_2\} \models p$. Thus, we are calling the SAT solver when it would be unnecessary.*

Example 4 *Consider a set of MUSes $\{\mathcal{M}_1, ..., \mathcal{M}_n\}$ such that $\mathcal{M}_i = \mathcal{G} \wedge \mathcal{Q}_i$, with $\mathcal{G} \models l$ being a PS and $\mathcal{Q}_m \neq \mathcal{Q}_n$ for $m \neq n$. LBX (Algorithm 2) will test multiple times the literal $\neg l$.*

Thus, compared to caching cores, as presented in (Previti et al. 2017), caching PSes has the advantage of avoiding testing literals that belong to MUSes never seen before. Moreover, caching PSes is a natural choice in conjunction with LBX, which tests one literal at a time; in case a tested literal is implied, it is straightforward to obtain a PS from the SAT solver.

Premise Set Caching

As described in the previous section, PSes are shared among multiple MUSes, and hence, intuitively, PSes are recomputed by a SAT solver multiple times during the MCS enumeration process. As a consequence, storing already discovered PSes has the potential to save a possibly considerable number of SAT calls. Incremental SAT solving can in part record previously discovered PSes, but this is limited to those that were discovered recently during the last MCS extraction and subject to the clause deletion policy of a SAT solver. In this section, we present a new caching mechanism that stores PSes and cooperates with a SAT solver by sharing information. In one direction, the SAT solver informs the cache when a newly discovered PS is found. In the other direction, the cache informs the solver of all the backbone literals that it knows to be implied by the current formula.

PS Caching for MCS Enumeration

The following proposition is at the basis of PS caching.

Proposition 2 *Given a set of clauses $\mathcal{S}$ and a literal l, if $\mathcal{S} \models l$, then $\mathcal{S}' \models l$ for each $\mathcal{S}' \supseteq \mathcal{S}$.*

Example 5 *Consider as an example that during the MCS enumeration process, we have already discovered the PS $\{c_1, c_2, c_3\}$ implying the literal x_1. If during the extraction of another MCS, we have that the set of satisfied clauses $\mathcal{S}$ is such that $\{c_1, c_2, c_3\} \subseteq \mathcal{S}$, then we know immediately that x_1 is implied by $\mathcal{S}$ without the need to query the SAT solver.*

In order for this idea to be effective in practice, we need a fast mechanism that checks whether the current formula $\mathcal{S}$ contains a previously found PS. This mechanism is implemented by means of an external cache whose only requirement is to provide two operations:

1. Cache.add($\mathcal{P}$, l) and
2. Cache.entails($\mathcal{S}$, l),

where $\mathcal{P}$ is a PS, l is a literal and $\mathcal{S}$ is a set of (satisfiable) clauses.

The MCS algorithm and the PS cache interact as follows. Whenever a new PS $\mathcal{P}$ is found by the SAT solver, the PS is added to the cache by calling Cache.add($\mathcal{P}$, l). On the other hand, when a literal l has to be tested, before calling the SAT solver the cache is queried through Cache.entails($\mathcal{S}$, l). An intuitive way to check whether $\mathcal{S}' \subset \mathcal{S}$ would be to test if for all $c \in \mathcal{S}'$ we have that $c \in S$. However, consider the following. Let $\mathcal{F}$ be an unsatisfiable CNF formula, $\mathcal{S} \subset \mathcal{F}$ and $\mathcal{U} = \mathcal{F} \setminus \mathcal{S}$. We have that $\mathcal{S}' \subset \mathcal{F}$ is a subset of $\mathcal{S}$ iff $\mathcal{S}' \cap \mathcal{U} = \emptyset$. This leads to the following, practically motivated considerations.

1. The set $\mathcal{U}$ of falsified clauses can be (much) smaller than the set of satisfied clauses $\mathcal{S}$.
2. In order to prove $\mathcal{P} \subset \mathcal{S}$, for a given $\mathcal{P} \models l$ in the cache, we need to prove that there is no $c \in \mathcal{U}$ with $c \in \mathcal{P}$. Thus, if we find a clause $c \in \mathcal{U}$ that is also in $\mathcal{P}$, we cannot conclude that $\mathcal{S} \models l$.

We will present the idea of caching PSes in the context of LBX (Algorithm 3) as a natural candidate for integrating PS caching into. Whenever the output of the SAT solver on $\mathcal{S} \cup \{l\}$ is unsatisfiable, it returns a core C explaining the reason for unsatisfiability. Since the set of clauses $\mathcal{S}$ is satisfiable, the core C is guaranteed to contain the literal l. The premise set $\mathcal{P} = C \setminus \{l\}$ can be extracted by simply removing l from C. The main difference between LBX with PS caching (Algorithm 3) and LBX (Algorithm 2) are lines 6 and 16. Line 6 checks if a previously found PS implying $\neg l$ is contained in $\mathcal{S}$. If this is the case, $\neg l$ is added to the set of known backbone and the call to the solver at line 9 is skipped. Otherwise, the algorithm proceeds LBX, except when the SAT solver report unsatisfiability. In this case, the returned PS is added to the cache for future computation. Notice that the PS is guaranteed to be a new one, since otherwise the call at line 6 would have succeeded. Table 1 shows an example run of Algorithm 3 on the formula in Figure 1.

Algorithm 3: LBX with PS caching

```
1  Function LBX-CACHE (F, μ)
2      (S, U) ← InitPartition(F, μ)
3      (L, B) ← (L(U), ∅)
4      while L ≠ ∅ do
5          l ← RemoveLiteral(L)
6          if Cache.entails(S, ¬l) then
7              B ← B ∪ {¬l}
8              continue
9          (st, μ, C) = SAT(S ∪ B ∪ {l})
10         if st then
11             (S, U) ← UpdateSatClauses(μ, S, U)
12             L ← L ∩ L(U)
13         else
14             B ← B ∪ {¬l}
15             P = C \ {l}
16             Cache.add(P, ¬l)
17     return F \ S                          // MCS of F
```

Example 6 *The first column shows the split $(\mathcal{S}, \mathcal{L})$ produced by the last assignment (the set of falsified clauses $\mathcal{U}$ is omitted for space reasons). The pairs $(\mathcal{S}, \mathcal{L})$ marked with '$*$' represent the splits produced by an initial assignment. Those without represent the splits produced during the iterations. Due to the lack of SAT outcomes for this simple example, `UpdateSatClauses`$(\mu, \mathcal{S}, \mathcal{U})$ is actually never called and the set $\mathcal{S}$ remains the same within the computation of a single MCS. The second column is where the cache is queried in order to check if a literal l is entailed because of a previously discovered PS. If the cache is not able to return a positive answer, in the third column a SAT solver is called in order to test the formula $\mathcal{S} \cup \{l\}$. If the outcome is unsatisfiable, then the premise set implying $\neg l$ is stored into the cache and the literal $\neg l$ is added to the set $\mathcal{B}$ of known backbone literals. During the computation of the first MCS $\mathcal{C}_1$, the premise sets implying $p, \neg r, q$ are collected by the SAT solver (*`SAT`$(\mathcal{S} \cup \{l\})$*) and stored into the cache. While testing the literal $\neg p$, during the extraction of the second MCS, the cache reports that p is known to be implied (due to the previously stored PS $\{c_1, c_2\} \subset \mathcal{S}$) and we can skip a call*

Table 1: Example execution of Algorithm 3.

$(\mathcal{S}, \mathcal{L})$	$Cache \overset{?}{\models} \neg l \ (l \in \mathcal{L})$	$\texttt{SAT}(\mathcal{S} \cup \{l\})$	$\mathcal{P}$	$\mathcal{B}$
$*(\{c_1, c_2, c_4, c_5, c_7, c_8\}, \{\neg p, r, \neg q\})$	$Cache \not\models p$	$\texttt{SAT}(\mathcal{S} \cup \{\neg p\}) \models \emptyset$	$\{c_1, c_2\} \models p$	$\{p\}$
$(\{c_1, c_2, c_4, c_5, c_7, c_8\}, \{r, \neg q\})$	$Cache \not\models \neg r$	$\texttt{SAT}(\mathcal{S} \cup \{r\}) \models \emptyset$	$\{c_4\} \models \neg r$	$\{p, \neg r\}$
$(\{c_1, c_2, c_4, c_5, c_7, c_8\}, \{\neg q\})$	$Cache \not\models q$	$\texttt{SAT}(\mathcal{S} \cup \{\neg q\}) \models \emptyset$	$\{c_7, c_8\} \models q$	$\{p, \neg r, q\}$
$\mathcal{L} = \emptyset, \mathcal{C}_1 = \{c_3, c_6\}$				
$(\mathcal{S}, \mathcal{L})$	$Cache \overset{?}{\models} \neg l \ (l \in \mathcal{L})$	$\texttt{SAT}(\mathcal{S} \cup \{l\})$	$\mathcal{P}$	$\mathcal{B}$
$*(\{c_1, c_2, c_3, c_6, c_7\}, \{\neg r, \neg p, q, \neg s\})$	$Cache \not\models r$	$\texttt{SAT}(\mathcal{S} \cup \{\neg r\}) \models \emptyset$	$\{c_1, c_2, c_3\} \models r$	$\{r\}$
$(\{c_1, c_2, c_3, c_6, c_7\}, \{\neg p, q, \neg s\})$	$Cache \models p$	-	-	$\{r, p\}$
$(\{c_1, c_2, c_3, c_6, c_7\}, \{q, \neg s\})$	$Cache \not\models \neg q$	$\texttt{SAT}(\mathcal{S} \cup \{q\}) \models \emptyset$	$\{c_6\} \models \neg q$	$\{r, p, \neg q\}$
$(\{c_1, c_2, c_3, c_6, c_7\}, \{\neg s\})$	$Cache \not\models s$	$\texttt{SAT}(\mathcal{S} \cup \{\neg s\}) \models \emptyset$	$\{c_6, c_7\} \models s$	$\{r, p, \neg q, s\}$
$\mathcal{L} = \emptyset, \mathcal{C}_2 = \{c_4, c_5, c_8\}$				

to the SAT solver. Notice that we can infer the entailment of p without even having computed a core containing c_5.

Caching of cores for the computation of the second MCS in this particular case would not have been of any help when testing literals in the clause c_5.

Caching of PSes has the potential of making use of the structure of instance when MUSes are likely to share PSes (recall Examples 3 and 4). Intuitively, we expect caching PSes to be preferable to caching cores for instances with many MUSes intersecting with each other, while cores to be better when there are many disjoint MUSes. However, also in cases where cores would be preferable, Proposition 1 allows for obtaining cores from PSes.

Finally, we observed that, in practice, it is not uncommon that the initial partitioning at line 2 of Algorithm 3 splits the formula in two sets $\mathcal{S}$ and $\mathcal{U}$ such that $\mathcal{U}$ is already an MCS. This may be due to a close proximity of MCSes, which would then result in some MCSes being computed without the need of any SAT calls, but instead, only relying on the PSes stored in the cache.

Implementing PS Caching

In contrast to Previti et al. (2017), we develop a dedicated approach to implementing the PS cache, whereas Previti et al. (2017) rely on an external SAT solver for implementing the core cache. In more detail, we propose to implement the PS cache by means of a distinct clause-PS occurrence list for each literal, so that when testing literal l only PSes relative to l are checked. The occurrence list stores for each clause $c \in \mathcal{F}$ a pointer to the set of PSes (if any) that contains that clause. So, if a clause $c \in \mathcal{U}$, we immediately know the set of PSes that do not imply l. If after checking all the clauses in $\mathcal{U}$ there is at least one known PS that did not appear in any of the sets pointed by those clauses in the occurrence list, then the literal l is implied.

Finally, we note that while we presented an extension of LBX with PS caching, the use of PS caching is not limited to LBX. Indeed, PS caching has the advantage of being directly available for a broader range of state-of-the-art MCS algorithms (including CLD and ELS, for example). As a more concrete other example, consider CLD, to which PS caching can be integrated as follows. Suppose that $\mathcal{S} \cup \{D\}$ is unsatisfiable, with D being the D-clause employed in the CLD algorithm and $\mathcal{S}$ satisfiable. Since $\mathcal{S} \cup \{D\}$ is unsatisfiable, $\mathcal{S} \models \neg l$ for all $l \in D$. In order to return a PS for each literal after proving unsatisfiability, clauses can be propagated incrementally one-by-one (through their selector variables on the implementation level). As soon as a specific literal is assigned through unit propagation, the clauses (selector variables) used for making the assignment are part of the PS.

Experiments

We integrated the ideas presented in the preceding sections to the state-of-the-art MCS enumeration algorithm LBX (Mencía, Previti, and Marques-Silva 2015), using Minisat 2.2.0 (Eén and Sörensson 2004) as the underlying SAT solver. The implementation is available at https://www.cs.helsinki.fi/group/coreo/lbx-cache/. We will compare the performance of LBX with caching to that of ELS intrumented with unsatisfiability core caching, as proposed in (Previti et al. 2017), as well as CLD (Marques-Silva et al. 2013). For a cleaner integration of PS caching, our implementation of LBX does not include the additional optimizations proposed in (Mencía, Previti, and Marques-Silva 2015). As a result, we observed that our implementation of LBX shows similar performance as ELS when computing a single MCS. Following Previti et al. (2017), we used as benchmarks 811 instances from (Marques-Silva et al. 2013) originally used for benchmarking algorithms for extracting a single MCS, with the motivation that in terms of the harder task of MCS enumeration, it can be especially beneficial to employ caching on such hard instances. Expectedly, complete enumeration of MCSes is not possible for most of the benchmark. The experiments were run under Linux on 2-GHz machines. A per-instance memory limit of 8 GB and a per-instance time limit of 1800 seconds was enforced. We will compare the number of MCSes enumerated by the different approaches under the time limit.

The results are shown in Figures 2–4, with direct comparisons of LBX using PS caching with LBX without caching (Figure 2), ELS with core caching (Figure 3), and CLD (Figure 4), respectively. Integrating PS caching into LBX notice-

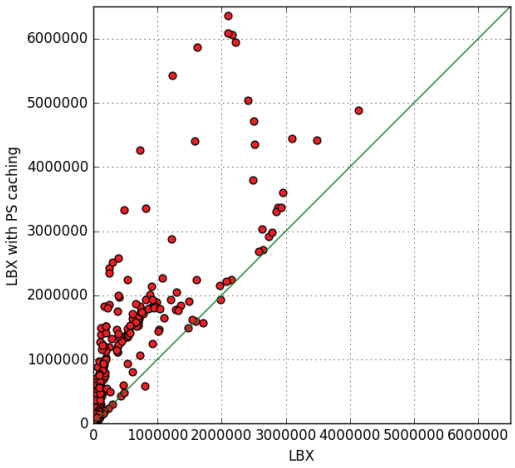

Figure 2: LBX with PS caching vs LBX without caching

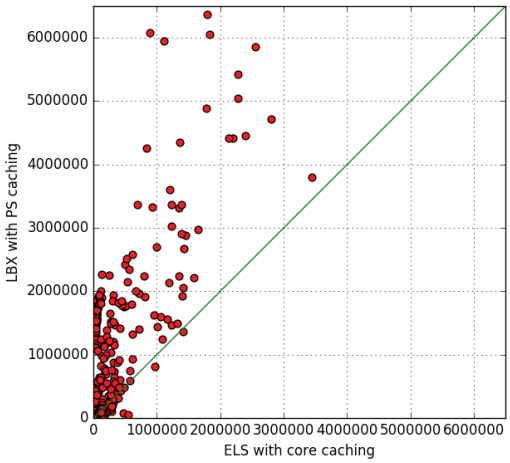

Figure 3: LBX with PS caching vs ELS with core caching

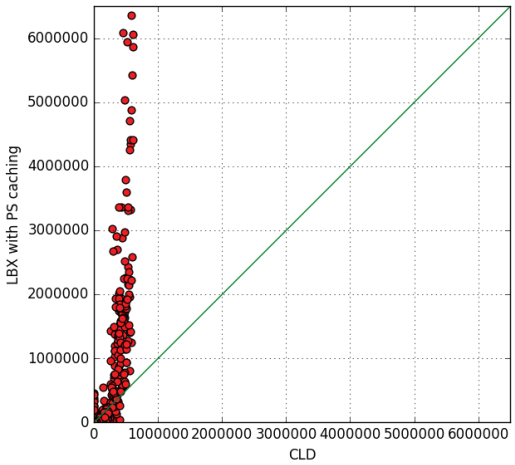

Figure 4: LBX with PS caching vs CLD

ably improves its performance on essentially all instances (Figure 2), the benefits of caching PSes clearly outweighing any possible overheads due to the lightweight PS cache maintenance. The improvement tends to be more significant when the SAT solver calls are expensive and/or when there are many MCSes. Caching of PSes tends to be beneficial also on non-trivial instances with relatively few MCSes, since then the SAT solver calls are more time-consuming. Furthermore, with only few MCSes, querying the cache is less expensive, as it will contain fewer PSes. Similar performance gains can be observed when comparing LBX using PS caching with the recently proposed approach of core caching for ELS (Figure 3): LBX using PS caching performs noticeably better on a great majority of the benchmarks, and there are only a few instances on which ELS using core caching manages to enumerate more MCSes. Finally, we consider the relative performance of LBX using PS caching and CLD (Figure 4). Here we observe that CLD achieves better performance only on instances on which the number of MCSes enumerated by both approaches is small. On a majority of instances, however, LBX using PS caching dominates CLD by several orders of magnitude.

We also observed PS caching is also successful in terms of empirical cache hit rates; on a majority of benchmarks cache hit rates are high, up to 90%. All in all, integrating the proposed PS caching approach to LBX evidently makes MCS enumeration more scalable compared to the enumeration capabilities of these state-of-the-art approaches.

Conclusions

The enumeration of minimal correction subsets is a central task in the analysis of overconstrained systems, with various real-world applications. Complementing recent advances in state-of-the-art algorithms for MCS enumeration, we proposed a novel approach to caching so-called premise sets during the MCS enumeration process. The approach is applicable widely in conjunction with state-of-the-art MCS enumeration algorithms. Empirically, we explained how premise set caching can be integrated into the LBX algorithm for MCS enumeration. Moreover, we showed that premise set caching noticeably improves on the scalability of MCS enumeration of state-of-the-art algorithms. In detail, PS caching noticeably improves the enumeration performance of LBX, making it very competitive against the complementary CLD algorithm, and is noticeably more effective than a recent unsatisfiable core caching approach proposed for the ELS algorithm. In terms of future work, there is potential for obtaining further speed-ups e.g. by developing heuristics for caching both PSes and unsatisfiable cores based on the underlying structural properties of instances.

Acknowledgments

A.P. and M.J. were supported by Academy of Finland (grants 251170 COIN, 276412, 284591, and 312662) and the Research Funds of the University of Helsinki. C.M. was supported by grant TIN2016-79190-R. J.M.S. was supported by FCT funding of LASIGE Research Unit, ref. UID/CEC/00408/2013.

References

Baader, F., and Suntisrivaraporn, B. 2008. Debugging SNOMED CT using axiom pinpointing in the description logic EL+. In *Proc. KR-MED*, volume 410 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Bacchus, F.; Davies, J.; Tsimpoukelli, M.; and Katsirelos, G. 2014. Relaxation search: A simple way of managing optional clauses. In *Proc. AAAI*, 835–841. AAAI Press.

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2009. Solving #SAT and Bayesian inference with backtracking search. *Journal of Artificial Intelligence Research* 34:391–442.

Bailey, J., and Stuckey, P. J. 2005. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proc. PADL*, volume 3350 of *Lecture Notes in Computer Science*, 174–186. Springer.

Beame, P.; Impagliazzo, R.; Pitassi, T.; and Segerlind, N. 2010. Formula caching in DPLL. *Transactions on Computation Theory* 1(3):9:1–9:33.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Birnbaum, E., and Lozinskii, E. L. 2003. Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental and Theoretical Artificial Intelligence* 15(1):25–46.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proc. STOC*, 151–158. ACM.

de la Banda, M. J. G.; Stuckey, P. J.; and Wazny, J. 2003. Finding all minimal unsatisfiable subsets. In *Proc. PPDP*, 32–43. ACM.

Eén, N., and Sörensson, N. 2004. An extensible SAT-solver. In *SAT 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Felfernig, A.; Schubert, M.; and Zehentner, C. 2012. An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 26(1):53–62.

Grégoire, É.; Lagniez, J.; and Mazure, B. 2014. An experimentally efficient method for (MSS, coMSS) partitioning. In *Proc. AAAI*, 2666–2673. AAAI Press.

Jannach, D.; Schmitz, T.; Hofer, B.; and Wotawa, F. 2014. Avoiding, finding and fixing spreadsheet errors - A survey of automated approaches for spreadsheet QA. *Journal of Systems and Software* 94:129–150.

Jose, M., and Majumdar, R. 2011. Cause clue clauses: error localization using maximum satisfiability. In *Proc. PLDI*, 437–446. ACM.

Kilby, P.; Slaney, J. K.; Thiébaux, S.; and Walsh, T. 2005. Backbones and backdoors in satisfiability. In *Proc. AAAI*, 1368–1373. AAAI Press.

Kitching, M., and Bacchus, F. 2007. Symmetric component caching. In *Proc. IJCAI*, 118–124.

Kopp, T.; Singla, P.; and Kautz, H. A. 2016. Toward caching symmetrical subtheories for weighted model counting. In *Proc. AAAI Beyond NP Workshop*, volume WS-16-05 of *AAAI Workshops*. AAAI Press.

Kullmann, O. 2011. Constraint satisfaction problems in clausal form II: minimal unsatisfiability and conflict structure. *Fundamenta Informaticae* 109(1):83–119.

Liffiton, M. H., and Sakallah, K. A. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning* 40(1):1–33.

Marques-Silva, J.; Heras, F.; Janota, M.; Previti, A.; and Belov, A. 2013. On computing minimal correction subsets. In *Proc. IJCAI*, 615–622. AAAI Press.

Mencía, C.; Ignatiev, A.; Previti, A.; and Marques-Silva, J. 2016. MCS extraction with sublinear oracle queries. In *Proc. SAT*, volume 9710 of *Lecture Notes in Computer Science*, 342–360. Springer.

Mencía, C.; Previti, A.; and Marques-Silva, J. 2015. Literal-based MCS extraction. In *Proc. IJCAI*, 1973–1979. AAAI Press.

Nöhrer, A.; Biere, A.; and Egyed, A. 2012. Managing SAT inconsistencies with HUMUS. In *Proc. VaMoS*, 83–91. ACM.

O'Sullivan, B.; Papadopoulos, A.; Faltings, B.; and Pu, P. 2007. Representative explanations for over-constrained problems. In *Proc. AAAI*, 323–328. AAAI Press.

Previti, A.; Mencía, C.; Järvisalo, M.; and Marques-Silva, J. 2017. Improving MCS enumeration via caching. In *Proc. SAT*, volume 10491 of *Lecture Notes in Computer Science*, 184–194. Springer.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1):57–95.

Roychoudhury, A., and Chandra, S. 2016. Formula-based software debugging. *Communications of the ACM* 59(7):68–77.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. In *SAT Online Proceedings*.

Sebastiani, R., and Vescovi, M. 2009. Axiom pinpointing in lightweight description logics via Horn-SAT encoding and conflict analysis. In *Proc. CADE*, volume 5663 of *Lecture Notes in Computer Science*, 84–99. Springer.

Slaney, J. 2014. Set-theoretic duality: A fundamental feature of combinatorial optimisation. In *Proc. ECAI*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 843–848. IOS Press.

Stuckey, P. J.; Sulzmann, M.; and Wazny, J. 2003. Interactive type debugging in Haskell. In *Proc. ACM SIGPLAN Workhop on Haskell*, 72–83. ACM.

Thurley, M. 2006. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. SAT*, volume 4121 of *Lecture Notes in Computer Science*, 424–429. Springer.