On Approaches to Explaining Infeasibility of Sets of Boolean Clauses

Éric Grégoire Bertrand Mazure Cédric Piette
Université Lille-Nord de France, Artois
CRIL-CNRS UMR 8188
F-62307 Lens Cedex, France
{gregoire,mazure,piette}@cril.fr

Abstract

These last years, the issue of locating and explaining contradictions inside sets of propositional clauses has received a renewed attention due to the emergence of very efficient SAT solvers. In case of inconsistency, many such solvers merely conclude that no solution exists or provide an upper approximation of the subset of clauses that are contradictory. However, in most application domains, only knowing that a problem does not admit any solution is not enough informative, and it is important to know which clauses are actually conflicting. In this paper, the focus is on the concept of Minimally Unsatisfiable Subformulas (MUSes), which explain logical inconsistency in terms of minimal sets of contradictory clauses. Specifically, various recent results and computational approaches about MUSes and related concepts are discussed.

1. Introduction

Many issues in the Artificial Intelligence domain amount to representing knowledge using tools that can be related to logic, where the logical satisfiability of the representation translates the feasibility of the problem. For example, some VLSI configuration problems are best modeled as sets of Boolean formulas; proving that no truth-assignment satisfying all formulas does exist guarantees important correctness properties of the circuit (see e.g. [32]). In this paper, the focus is on recent works allowing the infeasibility of a Boolean problem to be explained when no truth-assignment, called model, can be found.

Indeed, the issue of locating and explaining contradictions inside sets of propositional clauses has received a renewed attention these last years, due to the emergence of very efficient SAT solvers [8]. When a set of Boolean clauses is shown satisfiable by a SAT solver, this one delivers a model, which is a certificate of the satisfiability of the problem. On the contrary, in case of inconsistency, these approaches only ensure that no model exists, or provide an upper-approximation of the subset of contradictory clauses. However, such a conclusion is often not enough informative in many application domains. Indeed, users might need to localize in a precise manner a contradictory subpart of the problem that causes its inconsistency since all clauses of the unsatisfiable problem do not necessarily participate to its infeasibility. In some cases, only a subset of them are actually conflicting and make the whole set of clauses unsatisfiable; it can thus be important to pinpoint those contradictory clauses. The concept of Minimally Unsatisfiable Subformulas (MUS) is intended to achieve this objective since it allows logical inconsistency to be explained in terms of minimal sets of contradictory clauses. Accordingly, MUSes of a set of clauses represent the smallest – in terms of set-theoretical inclusion – explanations of the unsatisfiability of the set, and thus point out which precise subpart(s) of a problem should be "repaired" in order for feasibility to be recovered. The goal of this paper is to review various results and computational approaches about MUSes.

2. MUSes and logical background

A propositional or Boolean formula in conjunctive normal form (CNF, for short) is a finite set (interpreted as a conjunction) of clauses, where a clause is a set (interpreted as a disjunction) of literals, a literal being a Boolean variable or its negation. An interpretation is an application that assigns values from $\{true, false\}$ to every Boolean variable. An interpretation is called a *model* of a CNF when it satisfies the CNF, namely when it makes it $true$. SAT is the NP-complete problem which consists in deciding whether a CNF is satisfiable or not, i.e. whether this formula admits at least one model.

SAT is usually solved in practice thanks to variants of the well-known DPLL algorithm [9]. This algorithm consists in assigning a truth value to a variable from the CNF, simplifying the formula and then recursively checking if the simplified formula is satisfiable; if this is the case, the original

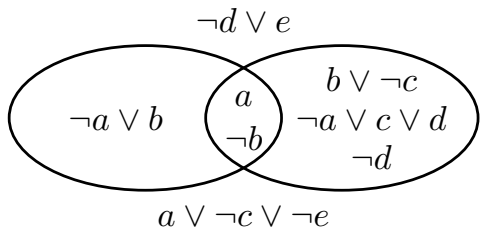


Figure 1. MUSes of example 1

formula is satisfiable; otherwise, the same recursive check is done assuming the opposite truth value. The simplification step essentially removes all clauses which become $true$ under the assignment from the formula, and all literals that become $false$ from the remaining clauses. Moreover, DPLL uses the unit propagation rule: if a clause is a unit clause, i.e. if it contains only a single unassigned literal, this clause can only be satisfied by assigning the necessary value to make this literal $true$. Thus, no choice is necessary. In practice, this often leads to cascades of unit propagations, thus pruning large parts of the search space. Modern exhaustive solvers are all based on this DPLL algorithm and include enhanced techniques such as lazy data structures, learning from conflict analysis, dynamic heuristics or non-chronological backtracking. More details about SAT solving can be found in [4].

Any unsatisfiable CNF exhibits at least one *Minimally Unsatisfiable Subformula*, which is defined as follow:

Definition 1 *A* Minimally Unsatisfiable Subformula *(MUS)* Γ *of a CNF* Σ *is a set of clauses s.t.:*

1. $\Gamma \subseteq \Sigma$
2. Γ *is unsatisfiable*
3. $\forall \Delta \subset \Gamma$, Δ *is satisfiable*

Clearly, an MUS of Σ corresponds to a minimal subset of clauses of Σ that forms a resolution-tree leading to inconsistency [33]. However, most current SAT solvers do not deliver such explanations, or simply deliver sets of conflicting clauses that are not guaranteed to be minimal ones. Unfortunately, extracting an MUS within CNF is a hard computational problem in the worst case since deciding whether a CNF is an MUS or not is DP-complete [31].

Example 1 *Let* $\Sigma = \{\neg d \vee e,\ b \vee \neg c,\ \neg d,\ \neg a \vee b,\ a,\ a \vee \neg c \vee \neg e,\ \neg a \vee c \vee d,\ \neg b\}$ *be a CNF. The set* $\Sigma_M = \{a,\ \neg a \vee b,\ \neg b\}$ *is an MUS of* Σ*. Indeed,* Σ_M *is unsatisfiable and each of its proper subsets is satisfiable.*

A CNF can exhibit several MUSes. Yet, recovering satisfiability requires at least one clause from each MUS of the CNF to be removed. Those MUSes can contain disjoint sets of clauses, or share some of them.

Example 2 *Let* Σ *be the CNF from the previous example.* Σ_M *is not the only MUS of* Σ*. Indeed,* $\Sigma_{M'} = \{b \vee \neg c,\ \neg d,\ a,\ \neg a \vee c \vee d,\ \neg b\}$ *is another MUS of* Σ*. Those two MUSes, are represented with the remaining part of* Σ *in Figure 1. They are the only MUSes of* Σ*. Let us notice that the clauses "*a*" "*$\neg b$*" belong to both MUSes of* Σ*. The removal of one of them from* Σ *is thus sufficient for satisfiability to be recovered, since it would "break" all the sources of inconsistency in* Σ*. On the contrary, the clause "*$\neg d \vee e$*" does not belong to any MUS and does not actually participate to the infeasibility of the CNF.*

Actually, the number of MUSes of a CNF can be exponential w.r.t. the number of clauses in the CNF. More precisely, a CNF composed of n clauses exhibits $C_n^{n/2}$ MUSes in the worst case. More generally, computational problems related to MUSes are typically very hard, and out of reach in the worst case. For instance, checking whether a set of clauses belongs to at least one MUS of an unsatisfiable CNF is Σ_2^p-hard (a consequence of Theorem 8.2 of [12]). Despite these bad worst-case complexity results, efficient techniques have been developped to approximate or compute MUSes for real-world problems. The most recent and efficient ones are reviewed in this paper.

The paper is organized as follows. In Section 3, known polynomial instances of the problem of extracting one MUS are reviewed. Section 4 discusses approaches to approximate one MUS in the general case, whereas main minimization procedures are described in Section 5. Next, exhaustive approaches to explain inconsistency are presented. In Section 7 various techniques to extract the smallest MUS (in terms of the number of clauses) are described. Finally, other important results about MUSes are discussed in Section 8.

3 Polynomial classes for the problem of extracting one MUS

Several fragments of Boolean logic are known to be polynomial time w.r.t. computing one MUS. First, a polynomial algorithm is proposed in [10] for locating an MUS that exhibits a *deficiency* equals to 1, the deficiency being defined as the difference between the number of clauses and the number of variables of a CNF. Clearly enough, any MUS exhibits a positive deficiency [1]. In a more general way, it has been shown that extracting an MUS with a deficiency equals to k (where k is a positive integer) is an NP-hard problem [7]. However, in the same paper, a polynomial algorithm is proposed when the deficiency of the extracted MUS is equal to 2. The complexity of extracting a minimal unsatisfiable formula with deficiency k is $n^{\mathcal{O}(k)}$ [13], where n is the number of variables of the formula.

More recently, new classes of CNFs have been proved polynomial w.r.t. the problem of extracting one of their

MUSes, when they exist. In [6], it is established that it is possible to extract one MUS for any CNF that obeys a so-called "integral property" thanks to a polynomial-time algorithm. Several well-known classes of CNFs obey this property, like (renamable, extended) Horn, Balanced, Matched, for instance. Let us stress that all these classes are also known to be "easy" for SAT. However, all polynomial classes for SAT are not polynomial for the problem of extracting one MUS. For example, CNFs made of binary clauses are polynomial w.r.t. SAT, but they are not polynomial w.r.t. the MUS finding problem.

4 Algorithms for approximating an MUS

Computing an MUS is a hard problem from a worst-case complexity point of view. Thus, complete and exact algorithms to compute MUSes can only be used for formulas that are often smaller than many large CNFs that are encoding real-world problems. Accordingly, the most efficient approaches cannot always guarantee the minimality of the delivered unsatisfiable formula, but deliver "approximations" of an MUS, only.

4.1 Adaptative search for MUSes

A first approach to approximate an MUS, called *adaptative* [5], is based on a preliminary computed *score* about the "difficulty" of clauses, the difficulty here being defined as a weight associated to a clause w.r.t. its estimated importance within the CNF. This is evaluated through an exploration on the search space, the processus recording indications about the "difficulty" to satisfy the various clauses of the CNF.

Next, a set of "difficult" clauses is produced. If it is proved unsatisfiable then it is delivered as an approximation of an MUS. If it is proved satisfiable then this set is *expanded*, and if after a given amount of computing resources no response to the satisfiability test is provided, then it is *contracted*.

The preprocessing step of this approach is crucial, since it enables the clauses to be stratified w.r.t. their difficulty. This approximation technique shows efficient on several families of instances but its accuracy is very dependent on several parameters (like the ratio of difficult clauses to select, etc.) which must be tuned for each CNF in order to find an unsatisfiable subformula of an acceptable size.

4.2 AMUSE

The basic principle of the `AMUSE` [30] method is to create a new variable for each clause of the CNF, introduce it as new disjunct inside the clause, and identify an unsatisfiable subformula thanks to an adapted complete search that gives those auxiliary variables a specific role. Actually, those additional literals are used to mark the corresponding clause; a complete DPLL-oriented search is run on this modified CNF, and when it ends, an unsatisfiable subformula can be found checking the value of those literals.

The `AMUSE` approach appears quite efficient and, on some instances, allows unsatisfiable subformulas to be obtained that are actual MUSes (even if the minimality criterion is not guaranteed). However, this method is efficient only when the MUS is small w.r.t. the size of the formula.

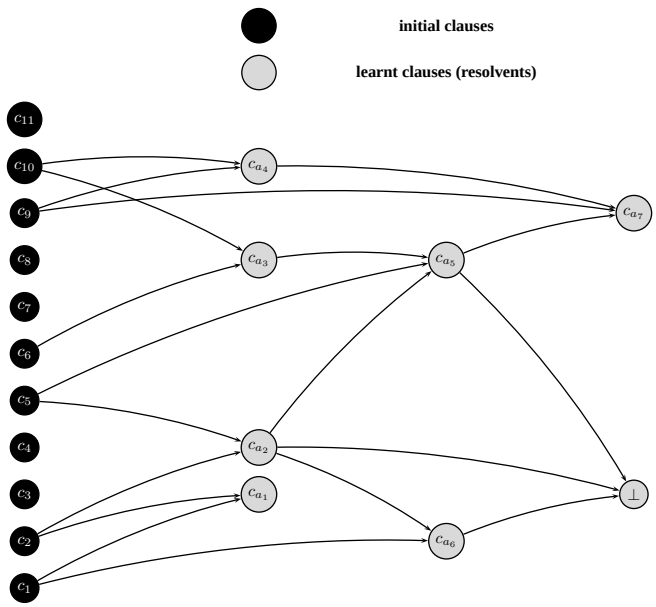
4.3 MUS extraction through learning

Various learning techniques from conflicts have been developped these last years in SAT solvers. Their goal is to reduce the remaining search space to prove (un)satisfiability, taking the information about encountered conflicts into account. Actually, these techniques can also be used to point out which subset of clauses from the CNF has been the source of a discovered conflict, and thus put in light clauses belonging to an unsatisfiable subformula [36].

Roughly, the principle behind such approaches, as implemented in `zCore`, is to keep a trace of the clauses responsible of the production of a new learnt clause, after each conflict. To this end, a Directed Acyclic Graph (DAC) called *resolution graph* is used. Each node of this graph represents a clause, the root nodes being the initial clauses of the CNF, and the internal nodes being the clauses learnt during the search. Edges symbolize resolution: if for instance, an edge starts from a node α to a node β, this means that α is one of the source clauses for learning β. Each clause represented in an internal node is the result of the resolution of each of its father nodes. An example of resolution graph is given in Figure 2.

In order to discriminate clauses that are responsible for yielding the empty clause, it suffices to consider the root nodes which are ancestors of the produced empty clause. Indeed, the other root nodes are not necessary for obtaining the refutation proof. Thus removing them does not invalidate this particular proof. As a consequence, this method ensures that an unsatisfiable formula will be obtained. In addition, once a subformula has been derived, a new complete search can be run with this new formula given in input, in order to produce a new proof that implies a new unsatisfiable subset of clauses. Iterating this procedure leads to compute smaller and smaller (in term of numbers of clauses) subformulas, until a fixed point is reached. However, the minimality of the obtained subformula is not guaranteed.

One main drawback of `zCore` [36] concerns the size of the resolution graph. Indeed, it can be very large, even for a formula of a restricted size. `zCore` solution for this problem consists in recording the resolution graph inside an external file stored on a hard disk. This storage option



In this example, clauses c_1, c_2, c_5, c_6, and c_{10} form an unsatisfiable subformula. Indeed, these clauses are the source of the refutation proof of the formula.

Figure 2. Refutation graph example

has a limited effect on the efficiency of the system since the nodes of the resolution graph are sorted topologically. Indeed, when a clause is generated by resolution, all its source clauses necessarily belong to the graph. However, when the empty clause is produced, the order according to which the graph is explored is reversed, which can lead to a performance problem since exploring a file in the "wrong" order is generally inefficient. To overcome this problem, zCore reverses the graph using a buffer. This technique often proves efficient for extracting an unsatisfiable subformula, but is limited by the required storage of the resolution graph. Recently, a variant of `zCore` has been proposed [14], which implements an analysis of the resolution graph in order to remove some clauses of the computed subformula.

4.4 MUS extraction based on local search

In [27], an original DPLL branching variable heuristic has been introduced. It consists in exploiting the trace of a local search computation to select the next variable to be assigned. The main idea is based on the heuristics that the most often falsified clauses during a local search are expected to be the most difficult ones to satisfy. Branching on such variables can thus lead to trigger many unit propagations, and thus reduce the size of the subsequent subformula.

Furthermore, this incomplete technique can provide a

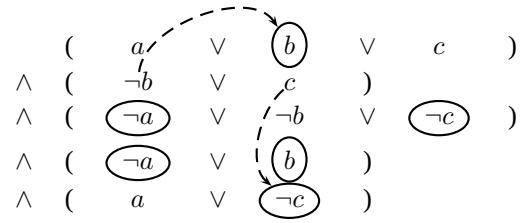


Figure 3. Example of critical clause from example 3

powerful heuristic for determining over-constrainted parts of a CNF. This one is studied in [17], where an algorithm called `AOMUS` is introduced. Based on the empirical findings that some clauses can be often falsified without belonging to any MUS, a refined criterium is investigated. This one is based on the concept of *critical clause*. Intuitively, a clause is critical w.r.t. a given complete interpretation if it is falsified and, at the same time, if the opposite of each of its literals belongs to at least one once-satisfied clause (namely, a clause that is satisfied by exactly one of its literals). Performing a flip[1] in order to satisfy one critical clause leads to falsify at least one other clause of the CNF. Particularly, this concept allows a relevant partial neighborhood of the explored interpretation to be taken into account, in order to decide whether a clause has to be considered by the counting heuristic that is expected to discrimate clauses belonging to MUSes. Interestingly, various properties establishing links between critical clauses and clauses that belong to MUSes can be found in [17]. The concept of critical clause is illustrated by the following example.

Example 3 *Let $\Sigma = \{a \vee b \vee c, \neg b \vee c, \neg a \vee \neg b \vee \neg c, \neg a \vee b, a \vee \neg c\}$ be a CNF. Actually, Σ is an MUS. Let $\omega = \{\neg a, b, \neg c\}$ be an interpretation. Literals that are satisfied by ω are circled in the representation of Σ in Figure 3. ω only falsifies one clause of Σ and once-satisfies two other ones. The falsified clause is critical, since the opposite of each of its literals belongs to a once-satisfied clause (clauses are pointed out by an arrow w.r.t. the "linked" literal). Satisfying this clause by flipping one variable of ω thus leads to falsify another clause, that was satisfied by ω.*

Thus, a local search algorithm is run on the CNF, and delivers a score for each of its clauses expressing the number of times this one has been critical. Clauses with the highest scores are expected to form a good approximation of the MUS(es) of the formula. Lowest-score clauses are then removed, and the resulting CNF is recorded in a FILO structure. Those operations are iterated until the local search

[1] A flip reverses *one* truth value in the current complete interpretation.

finds out a model of the current subformula. In this case, the last subformula from the FILO stack is checked for satisfiability. If it is unsatisfiable, then it is delivered as an MUS approximation, since removing a few clauses heuristically selected suffices to make it consistent. Else, the previously recorded subformulas are considered, until an unsatisfiable one is found, which is then the delivered approximation. This approach performs well on many CNFs, like e.g. random ones or FPGA encoding benchmarks [29].

5 Minimization procedures of unsatisfiable subformulas

The approaches in the last Section cannot ensure that the delivered CNF is minimal: they merely return a subformula that is proved unsatisfiable. However, in some applications, users might prefer getting an MUS, which is an exact cause of inconsistency in the sense that all of its clauses necessarily take part in the contradiction. Clearly enough, methods for computing MUSes in an exact way can be computationaly heavier than simply computing their approximations. In practice, an "approximation" approach as described in last Section is first run, before some kind of minimization step is performed to get an MUS. In this Section, various such minimization techniques are described.

5.1 Transition constraint-based techniques

Several minimization techniques have been proposed independently, in the context of various formal and problems frameworks (Constraints Satisfaction Problems, Mathematical Programming, Boolean logic). They are all based on the *transition constraint* concept.

Definition 2 *Let Σ be an unsatisfiable CNF and $(c_1, \ldots, c_n)$ be an ordering of the n clauses of Σ. There exists a unique clause c_i, called* transition constraint, *s.t. $\{c_1, \ldots, c_{i-1}\}$ is satisfiable and $\{c_1, \ldots, c_i\}$ is unsatisfiable.*

Property 1 *Let c_i be the transition constraint of an unsatisfiable CNF $\Sigma = \{c_1, \ldots, c_n\}$. c_i belongs to all MUSes of the $\{c_1, \ldots, c_i\}$ CNF.*

Computing a transition constraint hence enables one clause c_i of an MUS to be discovered, and the clauses with an index greater to i to be dropped from the list of candidate clauses for belonging to an MUS.

An MUS can be extracted by first computing a transition constraint c_i of the CNF and by reorganizing the ordering $(c_1, \ldots, c_i)$ into $(c_i, c_1, \ldots, c_{i-1})$. A second transition constraint c_j is then computed; the ordering becomes $(c_i, c_j, c_1, \ldots, c_{j-1})$. This process is iterated and stops when the set of transition constraints has been proved unsatisfiable. This set is an MUS.

The minimization techniques differ according to the way of the transition constraint is computed, leading to various efficiency performances. Three main techniques have been proposed to this end.

A first method, called *constructive* [11], starts with an empty CNF and inserts additional clauses in an incremental manner, as long as the resulting CNF remains satisfiable. Whenever the CNF becomes unsatisfiable, the clause introduced in the last place is identified as the transition constraint. A second method for finding the transition constraint is called *destructive* [3]. It considers the whole CNF and removes its clauses incrementally until the resulting CNF becomes satisfiable. The last removed clause is the transition constraint. Finally, a dichotomic technique has been proposed in [21], considering two bounds *min* and *max* (initialized to 1 and $|\Sigma|$, respectively). A satisfiabity test is performed on the CNF $\{c_1, \ldots, c_{(min+max)/2}\}$. If it is satisfiable, then *min*= $\lceil$(*min*+*max*)/2$\rceil$; otherwise, *max*= $\lfloor$(*min*+*max*)/2$\rfloor$. This process is iterated until *min*=*max*, which points out the transition constraint.

Let us notice that the minimization technique of `zCore` [36], called `zminimal`, is a destructive one. This is also the case of `AOMUS`, which also uses a (cheap) incomplete technique to avoid some calls to a complete solver, through the concept of *protected clause* [17]. Main t

Example 4 *Let Σ be a CNF composed of 30 clauses, and containing exactly one MUS made of clauses $1, 3, 8, 12, 17, 20, 21$. Following the numerical ordering, the behavior of the 3 minimization techniques for computing the first transition constraint (which is c_{21}) is illustrated in Figure 4.*

The computational complexity of these minimization techniques can be caracterized by their number of calls to an NP-complete oracle. Let Σ be a CNF, n the number of clauses of Σ, and k the size of the largest MUS of Σ, the time complexity of those techniques is:

- constructive: $\mathcal{O}(n \times k)$
- destructive: $\mathcal{O}(n)$
- dichotomic: $\mathcal{O}(log(n).k)$

Complexity results plead for the use of the dichotomic approach. Actually, it has been empirically observed that the dichotomic one is indeed the more efficient one when applied to approximations of MUSes of a rough quality. However, when the approximation of MUS is already accurate, the destructive approach is more adapted, because the dichotomic method can also need several tests to remove e.g. just one clause. In [18], a specific combination of both

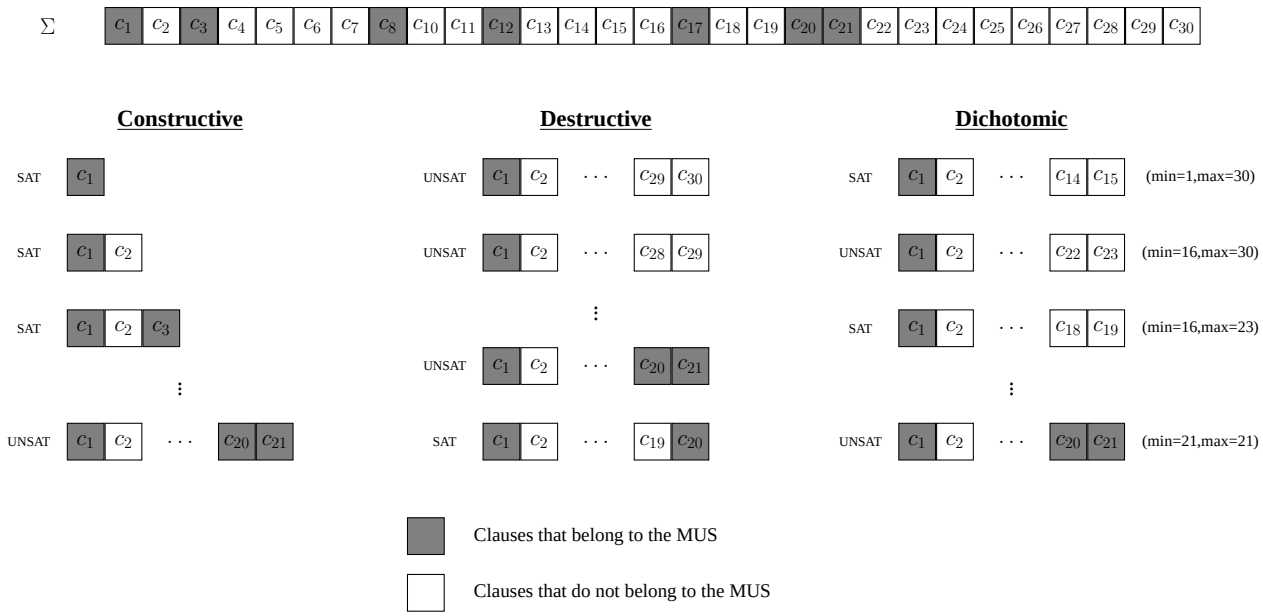


Figure 4. Behavior of the three minimization techniques

destructive and dichotomic approaches has been proposed and proved efficient for extracting an MUS for most families of problems.

5.2 MUP

`MUP` (*Minimal Unsatisfiability Prover*) [22] is a minimization technique that makes use of "clause selector" literals, like the `AMUSE` approach presented in Section 4.2. Unlike AMUSE that adds n variables in a formula containing n clauses, it is here suggested to only augment the formula with $\lceil \log(n+1) \rceil$ variables. In this way, the formula is proved minimally unsatisfiable if and only if its augmented version exhibits exactly n models with different values for the added variables, since the removal of each one of its clause makes the formula recover satisfiability. Thus, this minimality problem is reduced to the counting model one [22]. In addition, `MUP` uses Binary Decision Diagrams (BDD) for eliminating variables. Moreover, if the CNF is not an MUS, then `MUP` is able to provide the clauses to be removed in order for one MUS to be obtained. Nevertheless, the efficiency of this method highly relies on the quality of the initial approximation. Accordingly, it proves computationally more efficient to provide the approach with an approximation computed by a method from Section 4, rather than with the whole initial unsatisfiable CNF.

5.3 MiniUnsat

In [34], another minimization technique is presented, where an MUS is seen as a minimal irredundant subformula, in the specific context of inconsistency. The approach starts with an empty CNF Σ_M, and inserts inside Σ_M the clauses of the original CNF that are not redundant, namely clauses c s.t. $\Sigma_M \not\models c$, which is $true$ iff $\neg c \wedge \Sigma_M$ is satisfiable.

The algorithm is similar to the constructive method since it adds clauses only if they are not redundant with the current growing subformula, which proves generally more efficient. Moreover, `MiniUnsat` uses a so-called "associated assignment" concept for potentially adding several clauses after each complete check, instead of just one, like the classical constructive policy does.

6 Approaches for computing all MUSes

In this Section, complete methods are reviewed, namely methods that are expected to deliver the exhaustive set of MUSes of a CNF. As emphasized earlier, the size of this set is exponential in the worst case. However, in real-world problems, it often remains of a manageable size.

General approaches to compute all minimally inconsistent subsets of a constraints system have been proposed by [3, 20]. These methods correspond to various explorations of a so-called `CS-tree`, which aims at enumerating all subproblems of the formula. Unfortunately, this kind of techniques is limited by the combinatorial explosion of the number of subclauses, and is not very efficient in practice.

6.1 Dualize and Advance

Another approach to compute all minimal unsatisfiable subsets of constraints has not been developped in the

MSS	$\{c_1, c_2, c_3, c_5, c_6\}$	$\{c_2, c_3, c_4, c_6\}$	$\{c_3, c_4, c_5\}$	$\{c_2, c_4, c_5\}$
(1)	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
CoMSS	$\{c_4\}$	$\{c_1, c_5\}$	$\{c_1, c_2, c_6\}$	$\{c_1, c_3, c_6\}$
(2)		$\downarrow$		
MUS	$\{c_2, c_3, c_4, c_5\}$	$\{c_1, c_4\}$	$\{c_4, c_5, c_6\}$	

Figure 5. From MSSes set to MUSes set

Boolean framework, but for Herbrand constraints systems [2, 19]. Particularly, this method, called `Dualize And Advance`, is based on the *Maximal Satisfiable Subformula* concept (MSS), which is defined as follows:

Definition 3 *A Maximal Satisfiable Subformula (MSS)* Γ *of a CNF* Σ *is a set of clauses s.t.:*

1. $\Gamma \subseteq \Sigma$
2. Γ *is satisfiable*
3. $\forall \Delta \subseteq (\Sigma \backslash \Gamma)$ *s.t.* $\Delta \neq \emptyset$, $\Gamma \cup \Delta$ *is unsatisfiable*

The MSS and MUS concepts are dual ones. Indeed, whereas an MUS is a subset of constraints s.t. each of its proper subset is satisfiable, an MSS is a satisfiable subset of constraints s.t. adding any other constraint from the system would make it inconsistent. Actually, the set of MUSes can be deduced from the set of CoMSSes (the complementary set of an MSS), since each set is a "minimal hitting set" of the other one.

Definition 4 *Given a collection of sets* Ω *built on a domain* D*, a* hitting set *of* Ω *is a set of elements of* D *that contains at least one element from each set of* Ω*. Formally:*

H *is a hitting set of* Ω *iff* $H \subseteq D$ *and* $\forall S \in \Omega, H \cap S \neq \emptyset$

Moreover, a hitting set H *of* Ω *is called* minimal *iff for any element* $e \in H$, $H \backslash \{e\}$ *is not a hitting set of* Ω*.*

This relation is also knows as "*hypergraph transversal problem*" (HTP). Hence, it "suffices" to compute all MSSes of a formula to be able to deduce all its MUSes.

Example 5 *Let* Σ *be an unsatisfiable CNF s.t.:*

$$\Sigma = \underset{\mathbf{c_1}}{(x_1)} \wedge \underset{\mathbf{c_2}}{(\neg x_3)} \wedge \underset{\mathbf{c_3}}{(x_2 \vee x_3)} \wedge \underset{\mathbf{c_4}}{(\neg x_1)} \wedge \underset{\mathbf{c_5}}{(x_1 \vee \neg x_2)} \wedge \underset{\mathbf{c_6}}{(x_2)}$$

The set of MSSes of Σ *is given in the first line of Figure 5. The MUSes are computed using a two steps process. First, the complementary of each MSS, called CoMSS, is produced* (1)*. Second, this hitting set is produced, by computing all sets containing exactly one clause of each CoMSS. Those sets are the MUSes of* Σ*.*

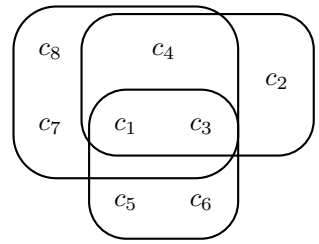


Figure 6. MUSes from example 6

In practice, `DAA` computes MSSes in a straight, incremental, way of growing them. Given a *seed*, under the form of a satisfiable set of constraints (initially the empty set), an MSS is built by trying to add each remaining constraint of the problem, and only keeping the ones which does not trigger a conflict. After having tried each possible clause, an MSS has been obtained, since any other clauses cannot be added without making the set unsatisfiable. When an MSS has been computed, the hitting set is computed to produce an MUS and/or a new seed. Each unsatisfiable element of the hitting set is an MUS, whereas each satisfiable element is a seed for the next iterations, guaranteed to be different from previous ones. Once all MSSes have been computed, the hitting set only produces unsatisfiable sets, which are the set of all MUSes of the CNF.

6.2 CAMUS

`DAA` has been revisited in [25], where a more efficient approach is proposed. This algorithm also builds the set of all MUSes from the set of all MSSes through *HTP*. Nevertheless, both sets are not computed simultaneously but in independent ways in this alternative version.

First, each clause c_i of the CNF is augmented with a new literal $\neg y_i$ called *clause selector*. Indeed, when solving this modified CNF, the assignment of y_i to $false$ desactivates c_i from the instance, by satisfying this clause trivially.

Example 6 *Let* Σ *be the following unsatisfiable CNF:*

$$\begin{array}{lcl}
\Sigma & & \Sigma_y \\
c_1 : \neg c \vee \neg b & & c_1 : \neg c \vee \neg b \vee \neg \mathbf{y_1} \\
c_2 : \neg a \vee \neg c & & c_2 : \neg a \vee \neg c \vee \neg \mathbf{y_2} \\
c_3 : c & & c_3 : c \vee \neg \mathbf{y_3} \\
c_4 : a \vee b & & c_4 : a \vee b \vee \neg \mathbf{y_4} \\
c_5 : \neg c \vee d & \Longrightarrow & c_5 : \neg c \vee d \vee \neg \mathbf{y_5} \\
c_6 : b \vee \neg d & & c_6 : b \vee \neg d \vee \neg \mathbf{y_6} \\
c_7 : \neg a \vee e & & c_7 : \neg a \vee e \vee \neg \mathbf{y_7} \\
c_8 : \neg e & & c_8 : \neg e \vee \neg \mathbf{y_8}
\end{array}$$

Σ *is built on 5 variables and contains 8 clauses. Those clauses form 3 MUS:* $MUS^1_\Sigma = \{c_1, c_2, c_3, c_4\}$, $MUS^2_\Sigma =$

$\{c_1, c_3, c_5, c_6\}$ *and* $MUS^3_\Sigma = \{c_1, c_3, c_4, c_7, c_8\}$*, represented in Figure 6. The first step of* CAMUS changes Σ *into* Σ_y *by adding clause selectors.*

An adapted DPLL algorithm is run on Σ_y. This complete solver allows a given number of clause selectors to be falsified, only. For each value of this bound (initialized to 1 and incremented at each new iteration), an exhaustive search is achieved to compute all models of Σ_y. From these models, CoMSSes of Σ can be deduced: those sets are actually made of the clauses satisfied by one of the added literals. When a model of Σ_y is found, the corresponding CoMSS is recorded.

Example 7 *After the first iteration of* CAMUS *on* Σ_y *from example 6, CoMSSes* $\{c_1\}$ *and* $\{c_3\}$ *are deduced. Blocking clauses* y_1 *and* y_3 *are added to the CNF, which becomes* $\Sigma^1_y = \Sigma_y \wedge \{y_1\} \wedge \{y_3\}$*. In the same way, after the second iteration (where 2 clause selectors are allowed to be falsified), CoMSSes* $\{c_4, c_5\}$ *and* $\{c_4, c_6\}$ *are obtained, and the CNF evolves into* $\Sigma^2_y = \Sigma^1_y \wedge \{y_4 \vee y_5\} \wedge \{y_4 \vee y_6\}$*.*

Without augmenting the CNF with those clauses, during the second iteration of the algorithm the set $C = \{c_3, c_8\}$ *would be computed as a CoMSS. Yet, this one cannot be a CoMSS, since* $\{c_3\}$ *has already been extracted. Accordingly,* C *is not minimal for set-theoretical inclusion, consequently it is not a CoMSS of* Σ*.*

This algorithm enables larger and larger CoMSSes to be derived, by incrementing the bound on the clause selectors. Moreover, after each iteration of the algorithm, one has to check whether the new instance augmented with blocking clauses is still satisfiable without any bound on clause selectors. If the CNF is unsatisfiable, then all CoMSSes have been delivered, since any interpretation falsifies an upper-set of a CoMSS of the CNF. The algorithm is thus stopped. Otherwise, larger CoMSSes exist; the bound is incremented and a new search is run.

Example 8 *After the first iteration of* CAMUS*, a consistency check on* Σ^1_y *is run (without any bound on clause selectors). Clearly, this CNF is satisfiable. After the extraction of CoMSS of size 2, a similar check is performed on* Σ^2_y*. This CNF is consistent: one of its model is for instance* $\omega^2_y = \{a, \neg b, c, \neg d, e, y_1, \neg y_2, y_3, \neg y_5, \neg y_8\}$*. The third iteration is the last one; it enables to extract the CoMSSes* $\{c_2, c_5, c_7\}, \{c_2, c_5, c_8\}, \{c_2, c_6, c_7\}$ *and* $\{c_2, c_6, c_8\}$*, and the CNF* Σ^3_y *is proved unsatisfiable, which entails that all its CoMSS have been extracted. The evolution of* Σ_y *is described in Figure 7.*

6.3 HYCAM

The CAMUS algorithm has been hybridized with an incomplete stochastic local search algorithm (SLS) [16]. The main idea is that SLS often explores interpretations that permit MSSes to be deduced. Indeed, such interpretations represent some forms of minima for the landscape explored by SLS, and it is well-known that SLS methods are often "attracted" towards those minima.

$$\Sigma^3_y \begin{cases} \Sigma^2_y \begin{cases} \Sigma^1_y \begin{cases} \Sigma_y \begin{cases} \neg c \vee \neg b \vee \neg y_1 \\ \neg a \vee \neg c \vee \neg y_2 \\ c \vee \neg y_3 \\ a \vee b \vee \neg y_4 \\ \neg c \vee d \vee \neg y_5 \\ b \vee \neg d \vee \neg y_6 \\ \neg a \vee e \vee \neg y_7 \\ \neg e \vee \neg y_8 \end{cases} \\ y_1 \\ y_3 \end{cases} \\ y_4 \vee y_5 \\ y_4 \vee y_6 \end{cases} \\ y_2 \vee y_5 \vee y_7 \\ y_2 \vee y_5 \vee y_8 \\ y_2 \vee y_6 \vee y_7 \\ y_2 \vee y_6 \vee y_8 \end{cases}$$

Figure 7. Evolution of Σ_y

More precisely, during the exploration of the set of truth-assignments made by a SLS, each set of falsified clauses is considered, and is recorded only if it is not a superset of a previous recorded set of clauses. After a given pre-fixed amount of time, SLS is stopped and the clasical CAMUS algorithm is run, with adding new blocking clauses w.r.t. previously found MSSes. It is shown in [16] that each actual preliminarily discovered MSS allows an NP-complete test together with CoNP-complete tests to be avoided. In practice, this hydrid approach proves very efficient by delivering a significant part of the MSSes of the CNF at a low cost, and in allowing significant gains of efficiency to be obtained.

7 Extracting one smallest MUS

In this Section, the main approaches to compute the smallest -in terms of the number of involved clauses- MUSes, noted SMUS (*Smallest Minimally Unsatisfiable Subformula*), are reviewed.

7.1 Enumerating subformulas

In [26], an algorithms for extracting one SMUS consists of a mere enumeration of subformulas. Roughly, a method that approximates an MUS (see Section 4) is called in order to obtain an unsatisfiable subformula, that provides an upper bound on the size of the smallest MUS. This enables to prune the exploration of a large number of subformu-

las. Then, each strictly smaller subformula of the CNF is checked for satisfiability and the smallest inconsistent one is recorded. Clearly enough, when a CNF is satisfiable, so are all its subformulas. Accordingly, when a subformula is proved satisfiable during the search, the exploration of all its subsets of clauses is pruned. After all subformulas have been tested, the smallest unsatisfiable one is delivered as a SMUS of the CNF. Unfortunately, this approach is limited by the combinatorial blow-up of the number of subformulas. In practice, it can be performed for very small formulas, only.

7.2 A branch & bound algorithm

A more efficient technique for extracting an SMUS is presented in [28]. It is based both on *branch & bound* techniques and on the following property:

Property 2 *Let Σ be a CNF, and ω be an interpretation falsifying the smallest possible number of clauses of Σ, or* MaxSat *solution. Any clause $c \in \Sigma$ falsified by ω belongs to at least one MUS of Σ.*

Accordingly, the technique of [28] requires us to iterate MaxSat solutions to improve a lower bound on the size of the SMUS. Indeed, if for instance two MaxSat solutions falsify disjoint sets of clauses, following Property 2 the size of the SMUS is at least two. This reasoning is generalized to lower-bound the size of the SMUS(es).

An upper bound is next obtained as follows: by considering the set-theoretical union of the disjoint sets of falsified clauses w.r.t. explored MaxSat solutions, one can obtain a subformula that is unsatisfiable. The set of all MUSes of this subformula is computed, and the upper bound is then the size of the smallest MUS of this set.

If the upper bound is equal to the lower bound, then the smallest MUS of the previously computed set of MUSes is delivered as one SMUS. Otherwise, specific subformulas are checked for satisfiability, the search being limited by the known bounds. Furthermore, the approach is enhanced by several additional features to improve the bounds and reduce the subformulas space.

7.3 A greedy genetic approach

Finally, a greedy genetic algorithm has been proposed for extracting one SMUS [35]. This approach uses the relation between Maximum Satisfiability and Minimum Unsatisfiability, like algorithms for computing all MUSes (see Section 6). So, the idea is to compute all MSSes and to derive the SMUS from them. Actually, each MSS is next considered as a *chromosome*, and a genetic approach is run, including the classic crossover, mutation, inversion and selection steps.

The authors show that their approach is faster than [28]'s technique by one order of magnitude in practice. Nevetheless, it remains an incomplete approach which can deliver an MUS that is not the smallest one. Even when one of the actual smallest one is extracted, no warranty can be provided, in opposite to the previously presented branch & bound algorithm that ensures the optimality of the returned set of clauses.

8 Other MUS-related results

In this Section some other results about the computation of MUSes are briefly described.

In [24], a preprocessor for MUSes-computing techniques is introduced. It is based on the *autarky* concept which is defined as a partial assignment that satisfies all clauses that contain an assigned literal. Indeed, within a CNF, clauses satisfied by an autarky cannot be a part of any MUS (and of any CoMSS neither). The authors propose a method to efficiently remove clauses from CNFs thanks to the computation of autarkies before using MUS-finding algorithms. They show that this approach is relevant in practice for the problems of finding out the SMUS and all MUSes of a CNF.

A theoretical study about categorization of clauses w.r.t. their role in inconsistency can be found in [23]. Particularly, the fact that clauses belong to one or several MUSes can determine their classification. Thus, any clause that belongs to all MUSes of a formula is called *necessary clause*. Those clauses belong to any refutation by resolution, and removing one of them allows satisfiability to be obtained. *Potentially necessary clauses* belong to at least one MUS of the formula but not to all of them. Removing one of them is not sufficient to regain satisfiability, but they can become necessary with the removal of appropriate clauses. Clauses that do not belong to any those categories are called *unnecessary*, because they do not belong to any MUS. Removing any combination of unnecessary clauses ensures that the inconsistency of the formula remains, but in this case, the size of the resolution proof can be very large. [23] provides finer-grained categorizations that allow for a better understanding of the various possible roles and properties of clauses within an unsatisfiable CNF.

Finally, the concept of *strict inconsistent covers* (SIC), introduced in [15], locates all disjoint areas of inconsistency in a CNF. This SIC can avoid to compute all MUSes of a CNF while it permits to restore easily the satisfiability of the CNF since the SIC is a set of disjoint MUSes of the CNF s.t. the removing of these MUSes gives a satisfiable CNF.

9 Conclusions

By stressing on minimal subsets of contradictory clauses inside a CNF, MUSes is a useful concept to explain infeasibility of a CNF. The issue of extracting MUSes in CNFs has received much attention these last years, leading to significant practical computational progress, despite bad worst-case complexity results. In this paper, recent approaches to provide users with MUS-based explanations of infeasibility have been presented.

References

[1] R. Aharoni and N. Linial. Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. *Journal of Combinatorial Theory Series A*, 43(2):196–204, 1986.

[2] J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL'05*, pages 174–186, 2005.

[3] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI'93*, volume 1, pages 276–281, 1993.

[4] L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4), 2006.

[5] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130(2):85–100, 2003.

[6] R. Bruni. On exact selection of minimally unsatisfiable subformulae. *Annals of mathematics and artificial intelligence*, 43(1):35–50, 2005.

[7] H. Büning. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 107(1–3):83–98, 2000.

[8] SAT competitions, http://www.satcompetition.org.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] G. Davydov, I. Davydova, and H. Büning. An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF. *Annals of Mathematics and Artificial Intelligence*, 23(3–4):229–245, 1998.

[11] N. de Siqueira and J.-F. Puget. Explanation-based generalisation of failures. In *ECAI'88*, pages 339–344, 1988.

[12] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactual. *Artificial Intelligence*, 57:227–270, 1992.

[13] H. Fleischner, O. Kullman, and S. Szeider. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, 289(1):503–516, 2002.

[14] R. Gershman, M. Koifman, and O. Strichman. Deriving small unsatisfiable cores with dominators. In *CAV'06*, pages 109–122, 2006.

[15] É. Grégoire, B. Mazure, and C. Piette. Tracking MUSes and strict inconsistent covers. In *FMCAD'06*, pages 39–46, 2006.

[16] É. Grégoire, B. Mazure, and C. Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *IJCAI'07*, volume 2, pages 2300–2305, 2007.

[17] É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of MUSes. *Constraints Journal*, 12(3):325–344, 2007.

[18] É. Grégoire, B. Mazure, and C. Piette. On finding minimally unsatisfiable cores of CSPs. *International Journal on Artificial Intelligence Tools (IJAIT)*, 2008. (to appear).

[19] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, 2003.

[20] B. Han and S.-J. Lee. Deriving minimal conflict sets by CS-Trees with mark set in diagnosis from first principles. In *IEEE Transactions on Systems, Man, and Cybernetics*, volume 29, pages 281–286, 1999.

[21] F. Hemery, C. Lecoutre, L. Saïs, and F. Boussemart. Extracting MUCs from constraint networks. In *ECAI'06*, pages 113–117, 2006.

[22] J. Huang. MUP: A minimal unsatisfiability prover. In *ASP-DAC'05*, pages 432–437, 2005.

[23] O. Kullmann, I. Lynce, and J. P. Marques Silva. Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In *SAT'06*, pages 22–35, 2006.

[24] M. Liffiton and K. Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In *SAT'08*, pages 182–195, 2008.

[25] M. H. Liffiton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In *SAT'05*, pages 173–186, 2005.

[26] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *SAT'04*, 2004.

[27] B. Mazure, L. Saïs, and É. Grégoire. Boosting complete techniques thanks to local search methods. *Annals of mathematics and artificial intelligence*, 22:319–331, 1998.

[28] M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. Marques Silva, and K. A. Sakallah. A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In *SAT'05*, pages 467–474, 2005.

[29] G.-J. Nam, F. A. Aloul, K. A. Sakallah, and R. A. Rutenbar. A comparative study of two boolean formulations of FPGA detailed routing constraints. *IEEE Trans. Computers*, 53(6):688–696, 2004.

[30] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.

[31] C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *Journal of Computer and System Sciences*, 37(1):2–13, 1988.

[32] M. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[33] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[34] H. van Maaren and S. Wieringa. Finding guaranteed MUSes fast. In *SAT'08*, pages 291–304, 2008.

[35] J. Zhang, S. Li, and S. Shen. Extracting minimum unsatisfiable cores with a greedy genetic algorithm. In *AI'06*, 2006.

[36] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *SAT'03*, 2003.